Proceedings of the Linux Symposium

Volume One

July 21st–24th, 2004
Ottawa, Ontario
Canada

Contents

TCP Connection Passing

Werner Almesberger
werner@almesberger.net

Abstract

tcpcp is an experimental mechanism that allows cooperating applications to pass ownership of TCP connection endpoints from one Linux host to another one. tcpcp can be used between hosts using different architectures and does not need the other endpoint of the connection to cooperate (or even to know what's going on).

1 Introduction

When designing systems for load-balancing, process migration, or fail-over, there is eventually the point where one would like to be able to "move" a socket from one machine to another one, without losing the connection on that socket, similar to file descriptor passing on a single host. Such a move operation usually involves at least three elements:

1. Moving any application space state related to the connection to the new owner. E.g. in the case of a Web server serving large static files, the application state could simply be the file name and the current position in the file.

2. Making sure that packets belonging to the connection are sent to the new owner of the socket. Normally this also means that the previous owner should no longer receive them.

3. Last but not least, creating compatible network state in the kernel of the new connection owner, such that it can resume the communication where the previous owner left off.

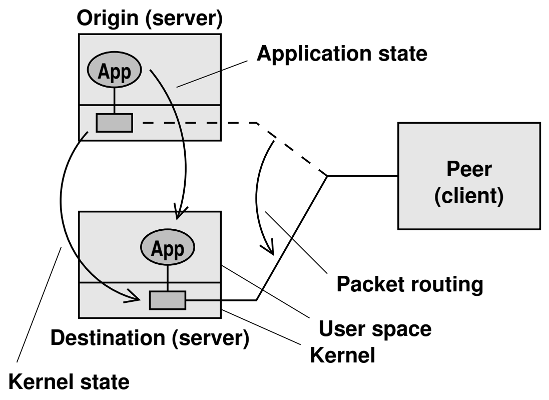


Figure 1: Passing one end of a TCP connection from one host to another.

Figure 1 illustrates this for the case of a client-server application, where one server passes ownership of a connection to another server. We shall call the host from which ownership of the connection endpoint is taken the *origin*, the host to which it is transferred the *destination*, and the host on the other end of the connection (which does not change) the *peer*.

Details of moving the application state are beyond the scope of this paper, and we will only sketch relatively simple examples. Similarly, we will mention a few ways for how the redirection in the network can be accomplished, but without going into too much detail.

The complexity of the kernel state of a network connection, and the difficulty of moving this state from one host to another, varies greatly with the transport protocol being used. Among the two major transport protocols of the Internet, UDP [1] and TCP [2], the latter clearly presents more of a challenge in this regard. Nevertheless, some issues also apply to UDP.

tcpcp (TCP Connection Passing) is a proof of concept implementation of a mechanism that allows applications to transport the kernel state of a TCP endpoint from one host to another, while the connection is established, and without requiring the peer to cooperate in any way. tcpcp is not a complete process migration or load-balancing solution, but rather a building block that can be integrated into such systems. tcpcp consists of a kernel patch (at the time of writing for version 2.6.4 of the Linux kernel) that implements the operations for dumping and restoring the TCP connection endpoint, a library with wrapper functions (see Section 3), and a few applications for debugging and demonstration.

The project's home page is at `http://tcpcp.sourceforge.net/`

The remainder of this paper is organized as follows: this section continues with a description of the context in which connection passing exists. Section 2 explains the connection passing operation in detail. Section 3 introduces the APIs tcpcp provides. The information that defines a TCP connection and its state is described in Section 4. Sections 5 and 6 discuss congestion control and the limitations TCP imposes on checkpointing. Security implications of the availability and use of tcpcp are examined in Section 7. We conclude with an outlook on future direction the work on tcpcp will take in Section 8, and the conclusions in Section 9.

The excellent "TCP/IP Illustrated" [3] is recommended for readers who wish to refresh their memory of TCP/IP concepts and terminology.

1.1 There is more than one way to do it

tcpcp is only one of several possible methods for passing TCP connections among hosts. Here are some alternatives:

In some cases, the solution is to avoid passing the "live" TCP connection, but to terminate the connection between the origin and the peer, and rely on higher protocol layers to re-establish a new connection between the destination and the peer. Drawbacks of this approach include that those higher layers need to know that they have to re-establish the connection, and that they need to do this within an acceptable amount of time. Furthermore, they may only be able to do this at a few specific points during a communication.

The use of HTTP redirection [4] is a simple example of connection passing above the transport layer.

Another approach is to introduce an intermediate layer between the application and the kernel, for the purpose of handling such redirection. This approach is fairly common in process migration solutions, such as Mosix [5], MIGSOCK [6], etc. It requires that the peer be equipped with the same intermediate layer.

1.2 Transparency

The key feature of tcpcp is that the peer can be left completely unaware that the connection is passed from one host to another. In detail, this means:

- The peer's networking stack can be used "as is," without modification and without requiring non-standard functionality
- The connection is not interrupted

- The peer does not have to stop sending
- No contradictory information is sent to the peer
- These properties apply to all protocol layers visible to the peer

Furthermore, tcpcp allows the connection to be passed at any time, without needing to synchronize the data stream with the peer.

The kernels of the hosts between which the connection is passed both need to support tcpcp, and the application(s) on these hosts will typically have to be modified to perform the connection passing.

1.3 Various uses

Application scenarios in which the functionality provided by tcpcp could be useful include load balancing, process migration, and fail-over.

In the case of load balancing, an application can send connections (and whatever processing is associated with them) to another host if the local one gets overloaded. Or, one could have a host acting as a dispatcher that may perform an initial dialog and then assigns the connection to a machine in a farm.

For process migration, tcpcp would be invoked when moving a file descriptor linked to a socket. If process migration is implemented in the kernel, an interface would have to be added to tcpcp to allow calling it in this way.

Fail-over is tricker, because there is normally no prior indication when the origin will become unavailable. We discuss the issues arising from this in more detail in Section 6.

2 Passing the connection

Figure 2 illustrates the connection passing procedure in detail.

1. The application at the origin initiates the procedure by requesting retrieval of what we call the *Internal Connection Information* (ICI) of a socket. The ICI contains all the information the kernel needs to re-create a TCP connection endpoint
2. As a side-effect of retrieving the ICI, tcpcp *isolates* the connection: all incoming packets are silently discarded, and no packets are sent. This is accomplished by setting up a per-socket filter, and by changing the output function. Isolating the socket ensures that the state of the connection being passed remains stable at either end.
3. The kernel copies all relevant variables, plus the contents of the out-of-order and send/retransmit buffers to the ICI. The out-of-order buffer contains TCP segments that have not been acknowledged yet, because an earlier segment is still missing.
4. After retrieving the ICI, the application empties the receive buffer. It can either process this data directly, or send it along with the other information, for the destination to process.
5. The origin sends the ICI and any relevant application state to the destination. The application at the origin keeps the socket open, to ensure that it stays isolated.
6. The destination opens a new socket. It may then bind it to a new port (there are other choices, described below).

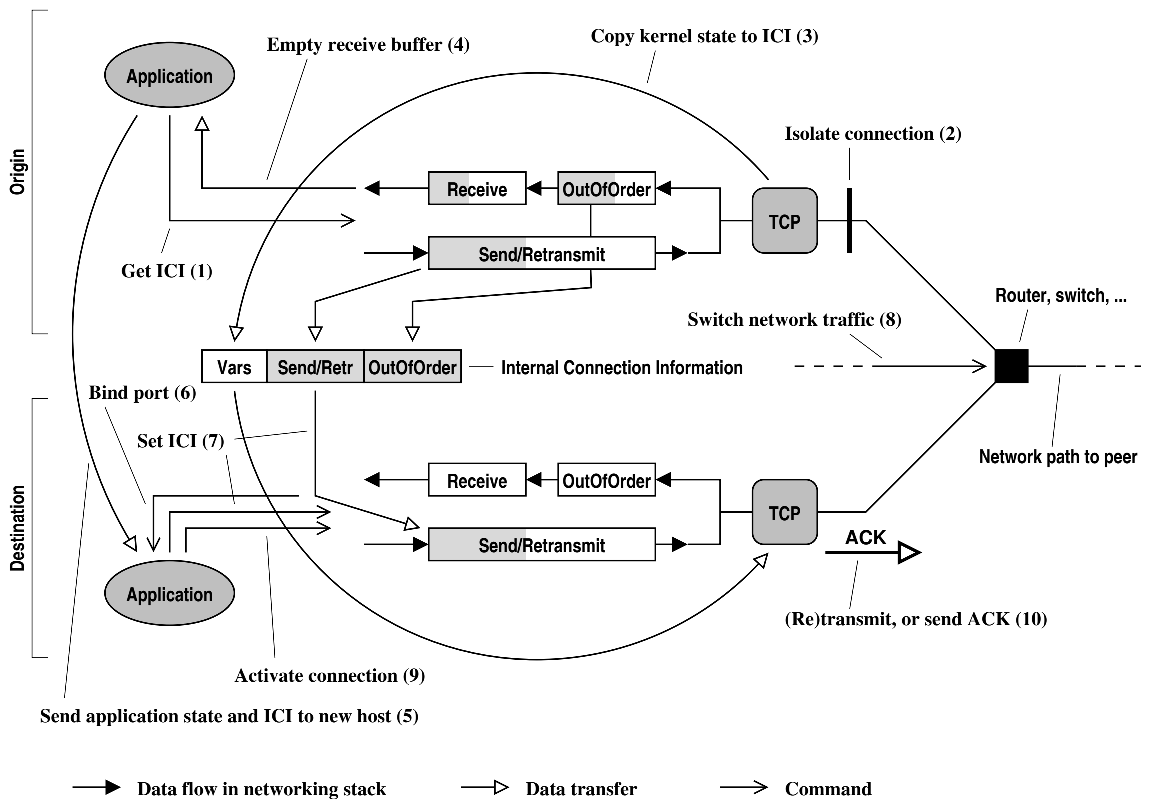


Figure 2: Passing a TCP connection endpoint in ten easy steps.

7. The application at the destination now sets the ICI on the socket. The kernel creates and populates the necessary data structures, but does not send any data yet. The current implementation makes no use of the out-of-order data.

8. Network traffic belonging to the connection is redirected from the origin to the destination host. Scenarios for this are described in more detail below. The application at the origin can now close the socket.

9. The application at the destination makes a call to *activate* the connection.

10. If there is data to transmit, the kernel will do so. If there is no data, an otherwise empty ACK segment (like a window probe) is sent to wake up the peer.

Note that, at the end of this procedure, the socket at the destination is a perfectly normal TCP endpoint. In particular, this endpoint can be passed to another host (or back to the original one) with tcpcp.

2.1 Local port selection

The local port at the destination can be selected in three ways:

- The destination can simply try to use the same port as the origin. This is necessary if no address translation is performed on the connection.

- The application can bind the socket before setting the ICI. In this case, the port in the ICI is ignored.

- The application can also clear the port information in the ICI, which will cause the socket to be bound to any available port. Compared to binding the socket before setting the ICI, this approach has the advantage of using the local port number space much more efficiently.

The choice of the port selection method depends on how the environment in which tcpcp operates is structured. Normally, either the first or the last method would be used.

2.2 Switching network traffic

There are countless ways for redirecting IP packets from one host to another, without help from the transport layer protocol. They include redirecting part of the link layer, ingenious modifications of how link and network layer interact [7], all kinds of tunnels, network address translation (NAT), etc.

Since many of the techniques are similar to network-based load balancing, the Linux Virtual Server Project [8] is a good starting point for exploring these issues.

While a comprehensive study of this topic if beyond the scope of this paper, we will briefly sketch an approach using a static route, because this is conceptually straightforward and relatively easy to implement.

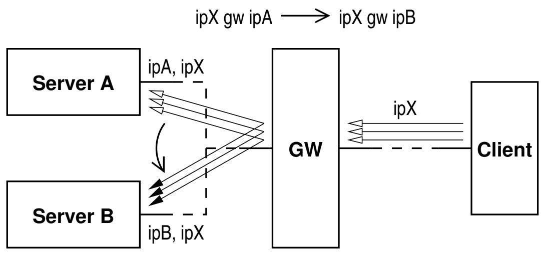


Figure 3: Redirecting network traffic using a static route.

The scenario shown in Figure 3 consists of two servers **A** and **B**, with interfaces with the IP addresses **ipA** and **ipB**, respectively. Each server also has a virtual interface with the address **ipX**. **ipA**, **ipB**, and **ipX** are on the same subnet, and also the gateway machine has an interface on this subnet.

At the gateway, we create a static route as follows:

`route add` *ipX* `gw` *ipA*

When the client connects to the address **ipX**, it reaches host **A**. We can now pass the connection to host **B**, as outlined in Section 2. In Step 8, we change the static route on the gateway as follows:

`route del` *ipX*
`route add` *ipX* `gw` *ipB*

One major limitation of this approach is of course that this routing change affects all connections to **ipX**, which is usually undesirable. Nevertheless, this simple setup can be used to demonstrate the operation of tcpcp.

3 APIs

The API for tcpcp consists of a low-level part that is based on getting and setting socket options, and a high-level library that provides convenient wrappers for the low-level API.

We mention only the most important aspects of both APIs here. They are described in more detail in the documentation that is included with tcpcp.

3.1 Low-level API

The ICI is retrieved by getting the `TCP_ICI` socket option. As a side-effect, the connection is isolated, as described in Section 2. The application can determine the maximum ICI size for the connection in question by getting the TCP_MAXICISIZE socket option.

Example:

```
void *buf;
int ici_size;
size_t size = sizeof(int);

getsockopt(s,SOL_TCP,TCP_MAXICISIZE,
    &ici_size,&size);
buf = malloc(ici_size);
size = ici_size;
getsockopt(s,SOL_TCP,TCP_ICI,
    buf,&size);
```

The connection endpoint at the destination is created by setting the TCP_ICI socket option, and the connection is activated by "setting" the TCP_CP_FN socket option to the value TCPCP_ACTIVATE.[1]

Example:

```
int sub_function = TCPCP_ACTIVATE;

setsockopt(s,SOL_TCP,TCP_ICI,
    buf,size);
/* ... */
setsockopt(s,SOL_TCP,TCP_CP_FN,
    &sub_function,
    sizeof(sub_function));
```

3.2 High-level API

These are the most important functions provided by the high-level API:

```
void *tcpcp_get(int s);
int tcpcp_size(const void *ici);
int tcpcp_create(const void *ici);
int tcpcp_activate(int s);
```

[1]The use of a multiplexed socket option is admittedly ugly, although convenient during development.

tcpcp_get allocates a buffer for the ICI, and retrieves that ICI (isolating the connection as a side-effect). The amount of data in the ICI can be queried by calling tcpcp_size on it.

tcpcp_create sets an ICI on a socket, and tcpcp_activate activates the connection.

4 Describing a TCP endpoint

In this section, we describe the parameters that define a TCP connection and its state. tcpcp collects all the information it needs to re-create a TCP connection endpoint in a data structure we call *Internal Connection Information* (ICI).

The ICI is portable among systems supporting tcpcp, irrespective of their CPU architecture.

Besides this data, the kernel maintains a large number of additional variables that can either be reset to default values at the destination (such as congestion control state), or that are only rarely used and not essential for correct operation of TCP (such as statistics).

4.1 Connection identifier

Each TCP connection in the global Internet or any private internet [9] is uniquely identified by the IP addresses of the source and destination host, and the port numbers used at both ends.

tcpcp currently only supports IPv4, but can be extended to support IPv6, should the need arise.

4.2 Fixed data

A few parameters of a TCP connection are negotiated during the initial handshake, and remain unchanged during the life time of the connection. These parameters include whether window scaling, timestamps, or selective acknowledgments are used, the number of bits by

Connection identifier	
`ip.v4.ip_src`	IPv4 address of the host on which the ICI was recorded (source)
`ip.v4.ip_dst`	IPv4 address of the peer (destination)
`tcp_sport`	Port at the source host
`tcp_dport`	Port at the destination host
Fixed at connection setup	
`tcp_flags`	TCP flags (window scale, SACK, ECN, etc.)
`snd_wscale`	Send window scale
`rcv_wscale`	Receive window scale
`snd_mss`	Maximum Segment Size at the source host
`rcv_mss`	MSS at the destination host
Connection state	
`state`	TCP connection state (e.g. ESTABLISHED)
Sequence numbers	
`snd_nxt`	Sequence number of next new byte to send
`rcv_nxt`	Sequence number of next new byte expected to receive
Windows (flow-control)	
`snd_wnd`	Window received from peer
`rcv_wnd`	Window advertised to peer
Timestamps	
`ts_gen`	Current value of the timestamp generator
`ts_recent`	Most recently received timestamp

Table 1: TCP variables recorded in tcpcp's Internal Connection Information (ICI) structure.

which the window is shifted, and the maximum segment sizes (MSS).

These parameters are used mainly for sanity checks, and to determine whether the destination host is able to handle the connection. The received MSS continues of course to limit the segment size.

4.3 Sequence numbers

The sequence numbers are used to synchronize all aspects of a TCP connection.

Only the sequence numbers we expect to see in the network, in either direction, are needed when re-creating the endpoint. The kernel uses several variables that are derived from these sequence numbers. The values of these variables either coincide with `snd_nxt` and `rcv_nxt` in the state we set up, or they can be calculated by examining the send buffer.

4.4 Windows (flow-control)

The (flow-control) window determines how much more data can be sent or received without overrunning the receiver's buffer.

The window the origin received from the peer is also the window we can use after re-creating the endpoint.

The window the origin advertised to the peer defines the minimum receive buffer size at the destination.

4.5 Timestamps

TCP can use timestamps to detect old segments with wrapped sequence numbers [10]. This mechanism is called *Protect Against Wrapped Sequence numbers* (PAWS).

Linux uses a global counter (`tcp_time_stamp`) to generate local timestamps. If a moved connection were to use the counter at the new host, local round-trip-time calculation may be confused when receiving timestamp replies from the previous connection, and the peer's PAWS algorithm will discard segments if timestamps appear to have jumped back in time.

Just turning off timestamps when moving the connection is not an acceptable solution, even though [10] seems to allow TCP to just stop sending timestamps, because doing so would bring back the problem PAWS tries to solve in the first place, and it would also reduce the accuracy of round-trip-time estimates, possibly degrading the throughput of the connection.

A more satisfying solution is to synchronization the local timestamp generator. This is accomplished by introducing a per-connection timestamp offset that is added to the value of `tcp_time_stamp`. This calculation is hidden in the macro `tp_time_stamp(tp)`, which just becomes `tcp_time_stamp` if the kernel is configured without tcpcp.

The addition of the timestamp offset is the only major change tcpcp requires in the existing TCP/IP stack.

4.6 Receive buffers

There are two buffers at the receiving side: the buffer containing segments received out-of-order (see Section 2), and the buffer with data that is ready for retrieval by the application.

tcpcp currently ignores both buffers: the out-of-order buffer is copied into the ICI, but not used when setting up the new socket. Any data in the receive buffer is left for the application to read and process.

4.7 Send buffer

The send and retransmit buffer contains data that is no longer accessible through the socket API, and that cannot be discarded. It is therefore placed in the ICI, and used to populate the send buffer at the destination.

4.8 Selective acknowledgments

In Section 5 of [11], the use of inbound SACK information is left optional. tcpcp takes advantage of this, and neither preserves SACK information collected from inbound segments, nor the history of SACK information sent to the peer.

Outbound SACKs convey information about the receiver's out-of-order queue. Fortunately, [11] declares this information as purely advisory. In particular, if reception of data has been acknowledged with a SACK, this does not imply that the receiver has to remember having done so. First, it can request retransmission of this data, and second, when constructing new SACKs, the receiver is encouraged to include information from previous SACKs, but is under no obligation to do so.

Therefore, while [11] discourages losing SACK information, doing so does not violate its requirements.

Losing SACK information may temporarily degrade the throughput of the TCP connection. This is currently of little concern, because tcpcp forces the connection into slow start, which has even more drastic performance implications.

SACK recovery may need to be reconsidered once tcpcp implements more sophisticated congestion control.

4.9 Other data

The TCP connection state is currently always ESTABLISHED. It may be useful to also allow passing connections in earlier states, e.g. SYN_RCVD. This is for further study.

Congestion control data and statistics are currently omitted. The new connection starts with slow-start, to allow TCP to discover the characteristics of the new path to the peer.

5 Congestion control

Most of the complexity of TCP is in its congestion control. tcpcp currently avoids touching congestion control almost entirely, by setting the destination to slow start.

This is a highly conservative approach that is appropriate if knowing the characteristics of the path between the origin and the peer does not give us any information on the characteristics of the path between the destination and the peer, as shown in the lower part of Figure 4.

However, if the characteristics of the two paths can be expected to be very similar, e.g. if the hosts passing the connection are on the same LAN, better performance could be achieved by allowing tcpcp to resume the connection at or nearly at full speed.

Re-establishing congestion control state is for further study. To avoid abuse, such an operation can be made available only to sufficiently trusted applications.

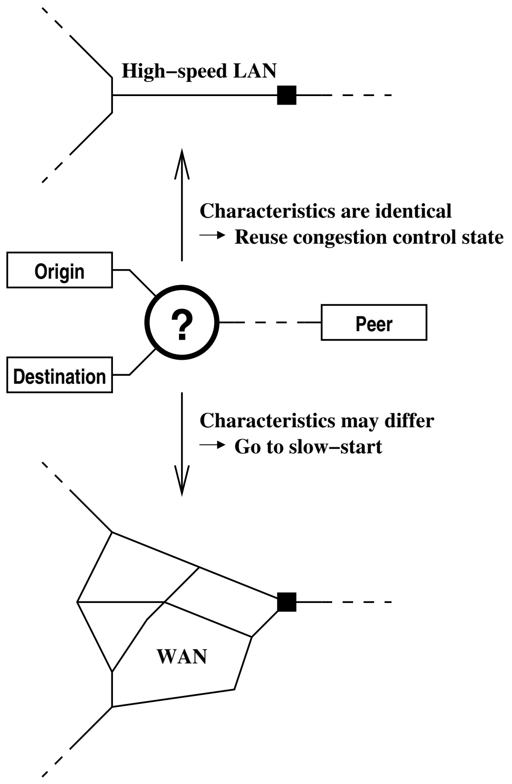

High−speed LAN
Characteristics are identical
→ Reuse congestion control state
Origin
?
Peer
Destination
Characteristics may differ
→ Go to slow−start
WAN

Figure 4: Depending on the structure of the network, the congestion control state of the original connection may or may not be reused.

6 Checkpointing

tcpcp is primarily designed for scenarios, where the old and the new connection owner are both functional during the process of connection passing.

A similar usage scenario would if the node owning the connection occasionally retrieves (“checkpoints”) the momentary state of the connection, and after failure of the connection owner, another node would then use the checkpoint data to resurrect the connection.

While apparently similar to connection passing, checkpointing presents several problems which we discuss in this section. Note that this is speculative and that the current implementation of tcpcp does not support any of the extensions discussed here.

We consider the send and receive flow of the TCP connection separately, and we assume that sequence numbers can be directly translated to application state (e.g. when transferring a file, application state consists only of the actual file position, which can be trivially mapped to and from TCP sequence numbers). Furthermore, we assume the connection to be in ESTABLISHED state at both ends.

6.1 Outbound data

One or more of the following events may occur between the last checkpoint and the moment the connection is resurrected:

- the sender may have enqueued more data
- the receiver may have acknowledged more data
- the receiver may have retrieved more data, thereby growing its window

Assuming that no additional data has been received from the peer, the new sender can simply re-transmit the last segment. (Alternatively, `tcp_xmit_probe_skb` might be useful for the same purpose.) In this case, the following protocol violations can occur:

- The sequence number may have wrapped. This can be avoided by making sure that a checkpoint is never older than the Maximum Segment Lifetime (MSL)[2], and that less than 2^{31} bytes are sent between checkpoints.
- If using PAWS, the timestamp may be below the last timestamp sent by the old sender. The best solution for avoiding this is probably to tightly synchronize clock on the old and the new connection owner, and to make a conservative estimate of the number of ticks of the local timestamp clock that have passed since taking the checkpoint. This assumes that the timestamp clock ticks roughly in real time.

[2][2] specifies a MSL of two minutes.

Since new data in the segment sent after resurrecting the connection cannot exceed the receiver's window, the only possible outcomes are that the segment contains either new data, or only old data. In either case, the receiver will acknowledge the segment.

Upon reception of an acknowledgment, either in response to the retransmitted segment, or from a packet in flight at the time when the connection was resurrected, the sender knows how far the connection state has advanced since the checkpoint was taken.

If the sequence number from the acknowledgment is below `snd_nxt`, no special action is necessary. If the sequence number is above `snd_nxt`, the sender would exceptionally treat this as a valid acknowledgment.[3]

As a possible performance improvement, the sender may notify the application once a new sequence number has been received, and the application could then skip over unnecessary data.

6.2 Inbound data

The main problem with checkpointing of incoming data is that TCP will acknowledge data that has not yet been retrieved by the application. Therefore, checkpointing would have to delay outbound acknowledgments until the application has actually retrieved them, and has

[3]Note that this exceptional condition does not necessarily have to occur with the first acknowledgment received.

checkpointed the resulting state change.

To intercept all types of ACKs, `tcp_transmit_skb` would have to be changed to send `tp->copied_seq` instead of `tp->rcv_nxt`. Furthermore, a new API function would be needed to trigger an explicit acknowledgment after the data has been stored or processed.

Putting acknowledges under application control would change their timing. This may upset the round-trip time estimation of the peer, and it may also cause it to falsely assume changes in the congestion level along the path.

7 Security

tcpcp bypasses various sets of access and consistency checks normally performed when setting up TCP connections. This section analyzes the overall security impact of tcpcp.

7.1 Two lines of defense

When setting TCP_ICI, the kernel has no means of verifying that the connection information actually originates from a compatible system. Users may therefore manipulate connection state, copy connection state from arbitrary other systems, or even synthesize connection state according to their wishes. tcpcp provides two mechanisms to protect against intentional or accidental mis-uses:

1. tcpcp only takes as little information as possible from the user, and re-generates as much of the state related to the TCP connection (such as neighbour and destination data) as possible from local information. Furthermore, it performs a number of sanity checks on the ICI, to ensure its integrity, and compatibility with constraints of the local system (such as buffer size limits and kernel capabilities).

2. Many manipulations possible through tcpcp can be shown to be available through other means if the application has the `CAP_NET_RAW` capability. Therefore, establishing a new TCP connection with tcpcp also requires this capability. This can be relaxed on a host-wide basis.

7.2 Retrieval of sensitive kernel data

Getting `TCP_ICI` may retrieve information from the kernel that one would like to hide from unprivileged applications, e.g. details about the state of the TCP ISN generator. Since the equally unprivileged `TCP_INFO` already gives access to most TCP connection metadata, tcpcp does not create any new vulnerabilities.

7.3 Local denial of service

Setting `TCP_ICI` could be used to introduce inconsistent data in the TCP stack, or the kernel in general. Preventing this relies on the correctness and completeness of the sanity checks mentioned before.

tcpcp can be used to accumulate stale data in the kernel. However, this is not very different from e.g. creating a large number of unused sockets, or letting buffers fill up in TCP connections, and therefore poses no new security threat.

tcpcp can be used to shutdown connections belonging to third party applications, provided that the usual access restrictions grant access to copies of their socket descriptors. This is similar to executing `shutdown` on such sockets, and is therefore believed to pose no new threat.

7.4 Restricted state transitions

tcpcp could be used to advance TCP connection state past boundaries imposed by internal or external control mechanisms. In particular, conspiring applications may create TCP connections without ever exchanging SYN packets, bypassing SYN-filtering firewalls. Since SYN-filtering firewalls can already be avoided by privileged applications, sites depending on SYN-filtering firewalls should therefore use the default setting of tcpcp, which makes its use also a privileged operation.

7.5 Attacks on remote hosts

The ability to set `TCP_ICI` makes it easy to commit all kinds of of protocol violations. While tcpcp may simplify implementing such attacks, this type of abuses has always been possible for privileged users, and therefore, tcpcp poses no new security threat to systems properly resistant against network attacks.

However, if a site allows systems where only trusted users may be able to communicate with otherwise shielded systems with known remote TCP vulnerabilities, tcpcp could be used for attacks. Such sites should use the default setting, which makes setting `TCP_ICI` a privileged operation.

7.6 Security summary

To summarize, the author believes that the design of tcpcp does not open any new exploits if tcpcp is used in its default configuration.

Obviously, some subtleties have probably been overlooked, and there may be bugs inadvertently leading to vulnerabilities. Therefore, tcpcp should receive public scrutiny before being considered fit for regular use.

8 Future work

To allow faster connection passing among hosts that share the same, or a very similar path to the peer, tcpcp should try to avoid going to slow start. To do so, it will have to pass more congestion control information, and integrate it properly at the destination.

Although not strictly part of tcpcp, the redirection apparatus for the network should be further extended, in particular to allow individual connections to be redirected at that point too, and to include some middleware that coordinates the redirecting with the changes at the hosts passing the connection.

It would be very interesting if connection passing could also be used for checkpointing. The analysis in Section 6 suggests that at least limited checkpointing capabilities should be feasible without interfering with regular TCP operation.

The inner workings of TCP are complex and easily disturbed. It is therefore important to subject tcpcp to thorough testing, in particular in transient states, such as during recovery from lost segments. The umlsim simulator [12] allows to generate such conditions in a deterministic way, and will be used for these tests.

9 Conclusion

tcpcp is a proof of concept implementation that successfully demonstrates that an endpoint of a TCP connection can be passed from one host to another without involving the host at the opposite end of the TCP connection. tcpcp also shows that this can be accomplished with a relatively small amount of kernel changes.

tcpcp in its present form is suitable for experimental use as a building block for load balancing and process migration solutions. Future work will focus on improving the performance of tcpcp, on validating its correctness, and on exploring checkpointing capabilities.

References

[1] RFC768; Postel, Jon. *User Datagram Protocol*, IETF, August 1980.

[2] RFC793; Postel, Jon. *Transmission Control Protocol*, IETF, September 1981.

[3] Stevens, W. Richard. *TCP/IP Illustrated, Volume 1 – The Protocols*, Addison-Wesley, 1994.

[4] RFC2616; Fielding, Roy T.; Gettys, James; Mogul, Jeffrey C.; Frystyk Nielsen, Henrik; Masinter, Larry; Leach, Paul J.; Berners-Lee, Tim. *Hypertext Transfer Protocol – HTTP/1.1*, IETF, June 1999.

[5] Bar, Moshe. *OpenMosix*, Proceedings of the 10th International Linux System Technology Conference (Linux-Kongress 2003), pp. 94–102, October 2003.

[6] Kuntz, Bryan; Rajan, Karthik. *MIGSOCK – Migratable TCP Socket in Linux*, CMU, M.Sc. Thesis, February 2002. `http://www-2.cs.cmu.edu/~softagents/migsock/MIGSOCK.pdf`

[7] Leite, Fábio Olivé. *Load-Balancing HA Clusters with No Single Point of Failure*, Proceedings of the 9th International Linux System Technology Conference (Linux-Kongress 2002), pp. 122–131, September 2002. `http://www.linux-kongress.org/2002/papers/lk2002-leite.html`

[8] *Linux Virtual Server Project*, `http://www.linuxvirtualserver.org/`

[9] RFC1918; Rekhter, Yakov; Moskowitz, Robert G.; Karrenberg, Daniel; de Groot, Geert Jan; Lear, Eliot. *Address Allocation for Private Internets*, IETF, February 1996.

[10] RFC1323; Jacobson, Van; Braden, Bob; Borman, Dave. *TCP Extensions for High Performance*, IETF, May 1992.

[11] RFC2018; Mathis, Matt; Mahdavi, Jamshid; Floyd, Sally; Romanow, Allyn. *TCP Selective Acknowledgement Options*, IETF, October 1996.

[12] Almesberger, Werner. *UML Simulator*, Proceedings of the Ottawa Linux Symposium 2003, July 2003. `http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Almesberger-OLS2003.pdf`

Cooperative Linux

Dan Aloni
`da-x@colinux.org`

Abstract

In this paper I'll describe Cooperative Linux, a port of the Linux kernel that allows it to run as an unprivileged lightweight virtual machine in kernel mode, on top of another OS kernel. It allows Linux to run under any operating system that supports loading drivers, such as Windows or Linux, after minimal porting efforts. The paper includes the present and future implementation details, its applications, and its comparison with other Linux virtualization methods. Among the technical details I'll present the CPU-complete context switch code, hardware interrupt forwarding, the interface between the host OS and Linux, and the management of the VM's pseudo physical RAM.

1 Introduction

Cooperative Linux utilizes the rather underused concept of a Cooperative Virtual Machine (CVM), in contrast to traditional VMs that are unprivileged and being under the complete control of the host machine.

The term **Cooperative** is used to describe two entities working in parallel, e.g. coroutines [2]. In that sense the most plain description of Cooperative Linux is turning two operating system kernels into two big coroutines. In that mode, each kernel has its own complete CPU context and address space, and each kernel decides when to give control back to its partner.

However, only one of the two kernels has control on the physical hardware, where the other is provided only with virtual hardware abstraction. From this point on in the paper I'll refer to these two kernels as the host operating system, and the guest Linux VM respectively. The host can be every OS kernel that exports basic primitives that provide the Cooperative Linux portable driver to run in CPL0 mode (ring 0) and allocate memory.

The special CPL0 approach in Cooperative Linux makes it significantly different than traditional virtualization solutions such as VMware, plex86, Virtual PC, and other methods such as Xen. All of these approaches work by running the guest OS in a less privileged mode than of the host kernel. This approach allowed for the extensive simplification of Cooperative Linux's design and its short early-beta development cycle which lasted only one month, starting from scratch by modifying the vanilla Linux 2.4.23-pre9 release until reaching to the point where KDE could run.

The only downsides to the CPL0 approach is stability and security. If it's unstable, it has the potential to crash the system. However, measures can be taken, such as cleanly shutting it down on the first internal Oops or panic. Another disadvantage is security. Acquiring root user access on a Cooperative Linux machine can potentially lead to root on the host machine if the attacker loads specially crafted kernel module or uses some very elaborated exploit in case which the Cooperative Linux kernel was compiled without module support.

Most of the changes in the Cooperative Linux patch are on the i386 tree—the only supported architecture for Cooperative at the time of this writing. The other changes are mostly additions of virtual drivers: cobd (block device), conet (network), and cocon (console). Most of the changes in the i386 tree involve the initialization and setup code. It is a goal of the Cooperative Linux kernel design to remain as close as possible to the standalone i386 kernel, so all changes are localized and minimized as much as possible.

2 Uses

Cooperative Linux in its current early state can already provide some of the uses that User Mode Linux[1] provides, such as virtual hosting, kernel development environment, research, and testing of new distributions or buggy software. It also enabled new uses:

- **Relatively effortless migration path from Windows.** In the process of switching to another OS, there is the choice between installing another computer, dual-booting, or using a virtualization software. The first option costs money, the second is tiresome in terms of operation, but the third can be the most quick and easy method—especially if it's free. This is where Cooperative Linux comes in. It is already used in workplaces to convert Windows users to Linux.
- **Adding Windows machines to Linux clusters.** The Cooperative Linux patch is minimal and can be easily combined with others such as the MOSIX or Open-MOSIX patches that add clustering capabilities to the kernel. This work in progress allows to add Windows machines to super-computer clusters, where one illustration could tell about a secretary workstation computer that runs Cooperative Linux as a screen saver—when the secretary goes home at the end of the day and leaves the computer unattended, the office's cluster gets more CPU cycles for free.
- **Running an otherwise-dual-booted Linux system from the other OS.** The Windows port of Cooperative Linux allows it to mount real disk partitions as block devices. Numerous people are using this in order to access, rescue, or just run their Linux system from their ext3 or reiserfs file systems.
- **Using Linux as a Windows firewall on the same machine.** As a likely competitor to other out-of-the-box Windows firewalls, iptables along with a stripped-down Cooperative Linux system can potentially serve as a network firewall.
- **Linux kernel development / debugging / research and study on another operating systems.**

  Digging inside a running Cooperative Linux kernel, you can hardly tell the difference between it and a standalone Linux. All virtual addresses are the same—Oops reports look familiar and the architecture dependent code works in the same manner, excepts some transparent conversions, which are described in the next section in this paper.
- **Development environment for porting to and from Linux.**

3 Design Overview

In this section I'll describe the basic methods behind Cooperative Linux, which include complete context switches, handling of hardware interrupts by forwarding, physical address translation and the pseudo physical memory RAM.

3.1 Minimum Changes

To illustrate the minimal effect of the Cooperative Linux patch on the source tree, here is a diffstat listing of the patch on Linux 2.4.26 as of May 10, 2004:

```
CREDITS                           |    6
Documentation/devices.txt         |    7
Makefile                          |    8
arch/i386/config.in               |   30
arch/i386/kernel/Makefile         |    2
arch/i386/kernel/cooperative.c    |  181 +++++
arch/i386/kernel/head.S           |    4
arch/i386/kernel/i387.c           |    8
arch/i386/kernel/i8259.c          |  153 ++++
arch/i386/kernel/ioport.c         |   10
arch/i386/kernel/process.c        |   28
arch/i386/kernel/setup.c          |   61 +
arch/i386/kernel/time.c           |  104 +++
arch/i386/kernel/traps.c          |    9
arch/i386/mm/fault.c              |    4
arch/i386/mm/init.c               |   37 +
arch/i386/vmlinux.lds             |   82 +-
drivers/block/Config.in           |    4
drivers/block/Makefile            |    1
drivers/block/cobd.c              |  334 ++++++++++
drivers/block/ll_rw_blk.c         |    2
drivers/char/Makefile             |    4
drivers/char/colx_keyb.c          | 1221 ++++++++++++++*
drivers/char/mem.c                |    8
drivers/char/vt.c                 |    8
drivers/net/Config.in             |    4
drivers/net/Makefile              |    1
drivers/net/conet.c               |  205 ++++++
drivers/video/Makefile            |    4
drivers/video/cocon.c             |  484 +++++++++++++++
include/asm-i386/cooperative.h    |  175 +++++
include/asm-i386/dma.h            |    4
include/asm-i386/io.h             |   27
include/asm-i386/irq.h            |    6
include/asm-i386/mc146818rtc.h    |    7
include/asm-i386/page.h           |   30
include/asm-i386/pgalloc.h        |    7
include/asm-i386/pgtable-2level.h |    8
include/asm-i386/pgtable.h        |    7
include/asm-i386/processor.h      |   12
include/asm-i386/system.h         |    8
include/linux/console.h           |    1
include/linux/cooperative.h       |  317 +++++++++
include/linux/major.h             |    1
init/do_mounts.c                  |    3
init/main.c                       |    9
kernel/Makefile                   |    2
kernel/cooperative.c              |  254 +++++++
kernel/panic.c                    |    4
kernel/printk.c                   |    6
50 files changed, 3828 insertions(+), 74 deletions(-)
```

3.2 Device Driver

The device driver port of Cooperative Linux is used for accessing kernel mode and using the kernel primitives that are exported by the host OS kernel. Most of the driver is OS-independent code that interfaces with the OS dependent primitives that include page allocations, debug printing, and interfacing with user space.

When a Cooperative Linux VM is created, the driver loads a kernel image from a vmlinux file that was compiled from the patched kernel with `CONFIG_COOPERATIVE`. The vmlinux file doesn't need any cross platform tools in order to be generated, and the same vmlinux file can be used to run a Cooperative Linux VM on several OSes of the same architecture.

The VM is associated with a per-process resource—a file descriptor in Linux, or a device handle in Windows. The purpose of this association makes sense: if the process running the VM ends abnormally in any way, all resources are cleaned up automatically from a callback when the system frees the per-process resource.

3.3 Pseudo Physical RAM

In Cooperative Linux, we had to work around the Linux MM design assumption that the entire physical RAM is bestowed upon the kernel on startup, and instead, only give Cooperative Linux a fixed set of physical pages, and then only do the translations needed for it to work transparently in that set. All the memory which Cooperative Linux considers as physical is in that allocated set, which we call the Pseudo Physical RAM.

The memory is allocated in the host OS using the appropriate kernel function—`alloc_pages()` in Linux and `MmAllocatePagesForMdl()` in Windows—so it is not mapped in any address space on the host for not wasting PTEs. The allocated pages are always resident and not freed until the VM is downed. Page tables

```
--- linux/include/asm-i386/pgtable-2level.h     2004-04-20 08:04:01.000000000 +0300
+++ linux/include/asm-i386/pgtable-2level.h     2004-05-09 16:54:09.000000000 +0300
@@ -58,8 +58,14 @@
 }
 #define ptep_get_and_clear(xp) __pte(xchg(&(xp)->pte_low, 0))
 #define pte_same(a, b)         ((a).pte_low == (b).pte_low)
-#define pte_page(x)            (mem_map+((unsigned long)(((x).pte_low >> PAGE_SHIFT))))
 #define pte_none(x)            (!(x).pte_low)
+
+#ifndef CONFIG_COOPERATIVE
+#define pte_page(x)            (mem_map+((unsigned long)(((x).pte_low >> PAGE_SHIFT))))
 #define __mk_pte(page_nr,pgprot) __pte(((page_nr) << PAGE_SHIFT) | pgprot_val(pgprot))
+#else
+#define pte_page(x)            CO_VA_PAGE((x).pte_low)
+#define __mk_pte(page_nr,pgprot) __pte((CO_PA(page_nr) & PAGE_MASK) | pgprot_val(pgprot))
+#endif

 #endif /* _I386_PGTABLE_2LEVEL_H */
```

Table 1: Example of MM architecture dependent changes

are created for mapping the allocated pages in the VM's kernel virtual address space. The VM's address space resembles the address space of a regular kernel—the normal RAM zone is mapped contiguously at 0xc0000000.

The VM address space also has its own special fixmaps—the page tables themselves are mapped at 0xfef00000 in order to provide an O(1) ability for translating PPRAM (Psuedo-Physical RAM) addresses to physical addresses when creating PTEs for user space and `vmalloc()` space. On the other way around, a special physical-to-PPRAM map is allocated and mapped at 0xff000000, to speed up handling of events such as pages faults which require translation of physical addresses to PPRAM address. This bi-directional memory address mapping allows for a negligible overhead in page faults and user space mapping operations.

Very few changes in the i386 MMU macros were needed to facilitate the PPRAM. An example is shown in Table 1. Around an #ifdef of `CONFIG_COOPERATIVE` the `__mk_pte()` low level MM macro translates a PPRAM struct page to a PTE that maps the real physical page. Respectively, `pte_page()` takes a PTE that was generated by `__mk_pte()` and returns the corresponding struct page for it. Other macros such as `pmd_page()` and `load_cr3()` were also changed.

3.4 Context Switching

The Cooperative Linux VM uses only one host OS process in order to provide a context for itself and its processes. That one process, named colinux-daemon, can be called a Super Process since it frequently calls the kernel driver to perform a context switch from the host kernel to the guest Linux kernel and back. With the frequent (HZ times a second) host kernel entries, it is able able to completely control the CPU and MMU without affecting anything else in the host OS kernel.

On the Intel 386 architecture, a complete context switch requires that the top page directory table pointer register—CR3—is changed. However, it is not possible to easily change both the instruction pointer (EIP) and CR3 in one instruction, so it implies that the code that changes CR3 must be mapped in both contexts for the change to be possible. It's problematic to map that code at the same virtual address in both contexts due to design limitations—the two contexts can divide the kernel and user address space differently, such that one virtual address can contain a kernel mapped page in one OS and a user mapped page in another.

In Cooperative Linux the problem was solved by using an intermediate address space during the switch (referred to as the 'passage page,' see Figure 1). The intermediate address space is defined by a specially created page tables in both the guest and host contexts and maps the same code that is used for the switch (passage code) at both of the virtual addresses that are involved. When a switch occurs, first CR3 is changed to point to the intermediate address space. Then, EIP is relocated to the other mapping of the passage code using a jump. Finally, CR3 is changed to point to the top page table directory of the other OS.

The single MMU page that contains the passage page code, also contains the saved state of one OS while the other is executing. Upon the beginning of a switch, interrupts are turned off, and a current state is saved to the passage page by the passage page code. The state includes all the general purpose registers, the segment registers, the interrupt descriptor table register (IDTR), the global descriptor table (GDTR), the local descriptor register (LTR), the task register (TR), and the state of the FPU / MMX / SSE registers. In the middle of the passage page code, it restores the state of the other OS and interrupts are turned back on. This process is akin to a "normal" process to process context switch.

Since control is returned to the host OS on every hardware interrupt (described in the following section), it is the responsibility of the host OS scheduler to give time slices to the Cooperative Linux VM just as if it was a regular process.

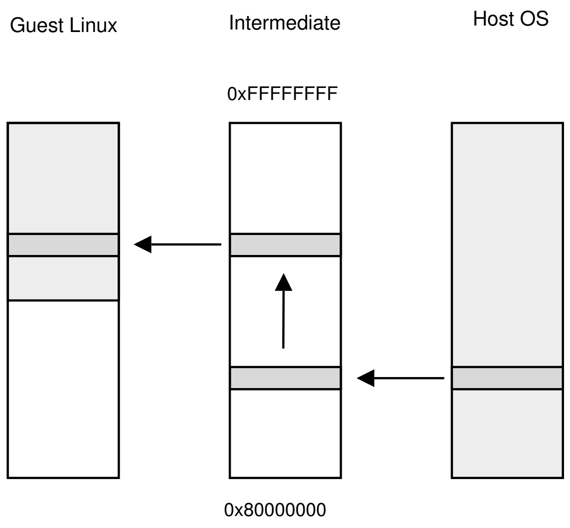


Figure 1: Address space transition during an OS cooperative kernel switch, using an intermapped page

3.5 Interrupt Handling and Forwarding

Since a complete MMU context switch also involves the IDTR, Cooperative Linux must set an interrupt vector table in order to handle the hardware interrupts that occur in the system during its running state. However, Cooperative Linux only forwards the invocations of interrupts to the host OS, because the latter needs to know about these interrupts in order to keep functioning and support the colinux-daemon process itself, regardless to the fact that external hardware interrupts are meaningless to the Cooperative Linux virtual machine.

The interrupt vectors for the internal processor exceptions (0x0–0x1f) and the system call vector (0x80) are kept like they are so that Cooperative Linux handles its own page faults and other exceptions, but the other interrupt vectors point to special proxy ISRs (interrupt service routines). When such an ISR is invoked during the Cooperative Linux context by an external hardware interrupt, a context switch is made to the host OS using the passage code. On the other side, the address of the relevant ISR of the host OS is determined by looking at its IDT. An interrupt call stack is forged and a jump occurs to that address. Between the invocation of the ISR in the Linux side and the handling of the interrupt in the host side, the interrupt flag is disabled.

The operation adds a tiny latency to interrupt handling in the host OS, but it is quite neglectable. Considering that this interrupt forwarding technique also involves the hardware timer interrupt, the host OS cannot detect that its CR3 was hijacked for a moment and therefore no exceptions in the host side would occur as a result of the context switch.

To provide interrupts for the virtual device drivers of the guest Linux, the changes in the arch code include a virtual interrupt controller which receives messages from the host OS on the occasion of a switch and invokes `do_IRQ()` with a forged `struct pt_args`. The interrupt numbers are virtual and allocated on a per-device basis.

4 Benchmarks And Performance

4.1 Dbench results

This section shows a comparison between User Mode Linux and Cooperative Linux. The machine which the following results were generated on is a 2.8GHz Pentium 4 with HT enabled, 512GB RAM, and a 120GB SATA Maxtor hard-drive that hosts ext3 partitions. The comparison was performed using the dbench 1.3-2 package of Debian on all setups.

The host machine runs the Linux 2.6.6 kernel patched with SKAS support. The UML kernel is Linux 2.6.4 that runs with 32MB of RAM, and is configured to use SKAS mode. The Cooperative Linux kernel is a Linux 2.4.26 kernel and it is configured to run with 32MB of RAM, same as the UML system. The root file-system of both UML and Cooperative Linux machines is the same host Linux file that contains an ext3 image of a 0.5GB minimized Debian system.

The commands ‘dbench 1’, ‘dbench 3’, and ‘dbench 10’ were run in 3 consecutive runs for each command, on the host Linux, on UML, and on Cooperative Linux setups. The results are shown in Table 2, Table 3, and Table 4.

System	Throughput	Netbench
Host	43.813	54.766
	50.117	62.647
	44.128	55.160
UML	10.418	13.022
	9.408	11.760
	9.309	11.636
coLinux	10.418	13.023
	12.574	15.718
	12.075	15.094

Table 2: output of dbench 10 (units are in MB/sec)

System	Throughput	Netbench
Host	43.287	54.109
	41.383	51.729
	59.965	74.956
UML	11.857	14.821
	15.143	18.929
	14.602	18.252
coLinux	24.095	30.119
	32.527	40.659
	36.423	45.528

Table 3: output of dbench 3 (units are in MB/sec)

4.2 Understanding the results

From the results in these runs, ‘dbench 10’, ‘dbench 3’, and ‘dbench 1’ show 20%, 123%, and 303% increase respectively, compared to UML. These numbers relate to the number

System	Throughput	Netbench
Host	158.205	197.756
	182.191	227.739
	179.047	223.809
UML	15.351	19.189
	16.691	20.864
	16.180	20.226
coLinux	45.592	56.990
	72.452	90.565
	106.952	133.691

Table 4: output of dbench 1 (units are in MB/sec)

of dbench threads, which is a result of the synchronous implementation of cobd[1]. Yet, neglecting the versions of the kernels compared, Cooperative Linux achieves much better probably because of low overhead with regard to context switching and page faulting in the guest Linux VM.

The current implementation of the cobd driver is synchronous file reading and writing directly from the kernel of the host Linux—No user space of the host Linux is involved, therefore less context switching and copying. About copying, the specific implementation of cobd in the host Linux side benefits from the fact that `filp->f_op->read()` is called directly on the cobd driver's request buffer after mapping it using `kmap()`. Reimplementing this driver as asynchronous on both the host and guest—can improve performance.

Unlike UML, Cooperative Linux can benefit in the terms of performance from the implementation of kernel-to-kernel driver bridges such as cobd. For example, currently virtual Ethernet in Cooperative Linux is done similar to UML—i.e., using user space daemons with tuntap on the host. If instead we create a kernel-to-kernel implementation with no user space daemons in between, Cooperative Linux has the potential to achieve much better in benchmarking.

[1]ubd UML equivalent

5 Planned Features

Since Cooperative Linux is a new project (2004–), most of its features are still waiting to be implemented.

5.1 Suspension

Software-suspending Linux is a challenge on standalone Linux systems, considering the entire state of the hardware needs to be saved and restored, along with the space that needs to be found for storing the suspended image. On User Mode Linux suspending [3] is easier—only the state of a few processes needs saving, and no hardware is involved.

However, in Cooperative Linux, it will be even easier to implement suspension, because it will involve its internal state almost entirely. The procedure will involve serializing the pseudo physical RAM by enumerating all the page table entries that are used in Cooperative Linux, either by itself (for user space and vmalloc page tables) or for itself (the page tables of the pseudo physical RAM), and change them to contain the pseudo value instead of the real value.

The purpose of this suspension procedure is to allow no notion of the real physical memory to be contained in any of the pages allocated for the Cooperative Linux VM, since Cooperative Linux will be given a different set of pages when it will resume at a later time. At the suspended state, the pages can be saved to a file and the VM could be resumed later. Resuming will involve loading that file, allocating the memory, and fix-enumerate all the page tables again so that the values in the page table entries point to the newly allocated memory.

Another implementation strategy will be to just dump everything on suspension as it is, but on resume—enumerate all the page table entries and adjust between the values of the old RPPFNs[2] and new RPPFNs.

Note that a suspended image could be created under one host OS and be resumed in another host OS of the same architecture. One could carry a suspended Linux on a USB memory device and resume/suspend it on almost any computer.

5.2 User Mode Linux[1] inside Cooperative Linux

The possibility of running UML inside Cooperative Linux is not far from being immediately possible. It will allow to bring UML with all its glory to operating systems that cannot support it otherwise because of their user space APIs. Combining UML and Cooperative Linux cancels the security downside that running Cooperative Linux could incur.

5.3 Live Cooperative Distributions

Live-CD distributions like KNOPPIX can be used to boot on top of another operating system and not only as standalone, reaching a larger sector of computer users considering the host operating system to be Windows NT/2000/XP.

5.4 Integration with ReactOS

ReactOS, the free Windows NT clone, will be incorporating Cooperative Linux as a POSIX subsystem.

5.5 Miscellaneous

- Virtual frame buffer support.
- Incorporating features from User Mode Linux, e.g. humfs [3].
- Support for more host operating systems such as FreeBSD.

[2]real physical page frame numbers

6 Conclusions

We have discussed how Cooperative Linux works and its benefits—apart from being a BSKH[4], Cooperative Linux has the potential to become an alternative to User Mode Linux that enhances on portability and performance, rather than on security.

Moreover, the implications that Cooperative Linux has on what is the media defines as 'Linux on the Desktop'—are massive, as the world's most dominant albeit proprietary desktop OS supports running Linux distributions for free, as another software, with the aimed-for possibility that the Linux newbie would switch to the standalone Linux. As user-friendliness of the Windows port will improve, the exposure that Linux gets by the average computer user can increase tremendously.

7 Thanks

Muli Ben Yehuda, IBM

Jun Okajima, Digital Infra

Kuniyasu Suzaki, AIST

References

[1] Jeff Dike. User Mode Linux. `http://user-mode-linux.sf.net`.

[3]A recent addition to UML that provides an host FS implementation that uses files in order to store its VFS metadata

[4]Big Scary Kernel Hack

[2] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, Massachusetts, 1997. Describes coroutines in their pure sense.

[3] Richard Potter. Scrapbook for User Mode Linux. `http://sbuml.sourceforge.net/`.

Build your own Wireless Access Point

Erik Andersen
Codepoet Consulting
`andersen@codepoet.org`

Abstract

This presentation will cover the software, tools, libraries, and configuration files needed to construct an embedded Linux wireless access point. Some of the software available for constructing embedded Linux systems will be discussed, and selection criteria for which tools to use for differing embedded applications will be presented. During the presentation, an embedded Linux wireless access point will be constructed using the Linux kernel, the uClibc C library, BusyBox, the syslinux bootloader, iptables, etc. Emphasis will be placed on the more generic aspects of building an embedded Linux system using BusyBox and uClibc. At the conclusion of the presentation, the presenter will (with luck) boot up the newly constructed wireless access point and demonstrate that it is working perfectly. Source code, build system, cross compilers, and detailed instructions will be made available.

1 Introduction

When I began working on embedded Linux, the question of whether or not Linux was small enough to fit inside a particular device was a difficult problem. Linux distributions[1] have historically been designed for server and desktop systems. As such, they deliver a full-featured, comprehensive set of tools for just about every purpose imaginable. Most Linux distributions, such as Red Hat, Debian, or SuSE, provide hundreds of separate software packages adding up to several gigabytes of software. The goal of server or desktop Linux distributions has been to provide as much value as possible to the user; therefore, the large size is quite understandable. However, this has caused the Linux operating system to be much larger then is desirable for building an embedded Linux system such as a wireless access point. Since embedded devices represent a fundamentally different target for Linux, it became apparent to me that embedded devices would need different software than what is commonly used on desktop systems. I knew that Linux has a number of strengths which make it extremely attractive for the next generation of embedded devices, yet I could see that developers would need new tools to take advantage of Linux within small, embedded spaces.

I began working on embedded Linux in the middle of 1999. At the time, building an 'embedded Linux' system basically involved copying binaries from an existing Linux distribution to a target device. If the needed software did not fit into the required amount of flash memory, there was really nothing to be done about it except to add more flash or give up on the project. Very little effort had been made to develop smaller application programs and li-

[1]The term "distribution" is used by the Linux community to refer to a collection of software, including the Linux kernel, application programs, and needed library code, which makes up a complete running system. Sometimes, the term "Linux" or "GNU/Linux" is also used to refer to this collection of software.

braries designed for use in embedded Linux.

As I began to analyze how I could save space, I decided that there were three main areas that could be attacked to shrink the footprint of an embedded Linux system: the kernel, the set of common application programs included in the system, and the shared libraries. Many people doing Linux kernel development were at least talking about shrinking the footprint of the kernel. For the past five years, I have focused on the latter two areas: shrinking the footprint of the application programs and libraries required to produce a working embedded Linux system. This paper will describe some of the software tools I've worked on and maintained, which are now available for building very small embedded Linux systems.

2 The C Library

Let's take a look at an embedded Linux system, the Linux Router Project, which was available in 1999. `http://www.linuxrouter.org/` The Linux Router Project, begun by Dave Cinege, was and continues to be a very commonly used embedded Linux system. Its self-described tagline reads "A networking-centric micro-distribution of Linux" which is "small enough to fit on a single 1.44MB floppy disk, and makes building and maintaining routers, access servers, thin servers, thin clients, network appliances, and typically embedded systems next to trivial." First, let's download a copy of one of the Linux Router Project's "idiot images." I grabbed my copy from the mirror site at `ftp://sunsite.unc.edu/pub/Linux/distributions/linux-router/dists/current/idiot-image_1440KB_FAT_2.9.8_Linux_2.2.gz`.

Opening up the idiot-image there are several very interesting things to be seen.

```
# gunzip \
  idiot-image_1440KB_FAT_2.9.8_Linux_2.2.gz
# mount \
  idiot-image_1440KB_FAT_2.9.8_Linux_2.2 \
  /mnt -o loop

# du -ch /mnt/*
34K     /mnt/etc.lrp
6.0K    /mnt/ldlinux.sys
512K    /mnt/linux
512     /mnt/local.lrp
1.0K    /mnt/log.lrp
17K     /mnt/modules.lrp
809K    /mnt/root.lrp
512     /mnt/syslinux.cfg
1.0K    /mnt/syslinux.dpy
1.4M    total

# mkdir test
# cd test
# tar -xzf /mnt/root.lrp

# du -hs
2.2M    .
2.2M    total

# du -ch bin root sbin usr var
460K    bin
8.0K    root
264K    sbin
12K     usr/bin
304K    usr/sbin
36K     usr/lib/ipmasqadm
40K     usr/lib
360K    usr
56K     var/lib/lrpkg
60K     var/lib
4.0K    var/spool/cron/crontabs
8.0K    var/spool/cron
12K     var/spool
76K     var
1.2M    total

# du -ch lib
24K     lib/POSIXness
1.1M    lib
1.1M    total

# du -h lib/libc-2.0.7.so
644K    lib/libc-2.0.7.so
```

Taking a look at the software contained in this embedded Linux system, we quickly notice that in a software image totaling 2.2 Megabytes, the libraries take up over half the space. If we look even closer at the set of libraries, we quickly find that the largest single component in the entire system is the GNU C library, in this case occupying nearly 650k. What is more, this is a very old version of the C library; newer versions of GNU glibc, such as version 2.3.2, are over 1.2 Megabytes all by themselves! There are tools available from Linux vendors and in the Open Source community which can reduce the footprint of the GNU C library considerably by stripping unwanted symbols; however, using such tools precludes adding additional software at a later date. Even when these tools are appropriate, there are limits to the amount of size which can be reclaimed from the GNU C library in this way.

The prospect of shrinking a single library that takes up so much space certainly looked like low hanging fruit. In practice, however, replacing the GNU C library for embedded Linux systems was not easy task.

3 The origins of uClibc

As I despaired over the large size of the GNU C library, I decided that the best thing to do would be to find another C library for Linux that would be better suited for embedded systems. I spent quite a bit of time looking around, and after carefully evaluating the various Open Source C libraries that I knew of[2], I sadly found that none of them were suitable replacements for glibc. Of all the Open Source C libraries, the library closest to what I imagined an embedded C library should be was called `uC-libc` and was being used for uClinux systems. However, it also had many problems at the time—not the least of which was that uC-libc had no central maintainer. The only mechanism being used to support multiple architectures was a complete source tree fork, and there had already been a few such forks with plenty of divergant code. In short, uC-libc was a mess of twisty versions, all different. After spending some time with the code, I decided to fix it, and in the process changed the name to `uClibc` (no hyphen).

With the help of D. Jeff Dionne, one of the creators of uClinux[3], I ported uClibc to run on Intel compatible x86 CPUs. I then grafted in the header files from glibc 2.1.3 to simplify software ports, and I cleaned up the resulting breakage. The header files were later updated again to generally match glibc 2.3.2. This effort has made porting software from glibc to uClibc extremely easy. There were, however, many functions in uClibc that were either broken or missing and which had to be re-written or created from scratch. When appropriate, I sometimes grafted in bits of code from the current GNU C library and libc5. Once the core of the library was reasonably solid, I began adding a platform abstraction layer to allow uClibc to compile and run on different types of CPUs. Once I had both the ARM and x86 platforms basically running, I made a few small announcements to the Linux community. At that point, several people began to make regular contributions. Most notably was Manuel Novoa III, who began contributing at that time. He has continued working on uClibc and is responsible for significant portions of uClibc such as the stdio and internationalization code.

After a great deal of effort, we were able to build the first shared library version of uClibc in January 2001. And earlier this year we were able to compile a Debian Woody system using uClibc[4], demonstrating the library is now able

[2]The Open Source C libraries I evaluated at the time included Al's Free C RunTime library (no longer on the Internet); dietlibc available from `http://www.fefe.de/dietlibc/`; the minix C library available from `http://www.cs.vu.nl/cgi-bin/raw/pub/minix/`; the newlib library available from `http://sources.redhat.com/newlib/`; and the eCos C library available from `ftp://ecos.sourceware.org/pub/ecos/`.

[3]uClinux is a port of Linux designed to run on microcontrollers which lack Memory Management Units (MMUs) such as the Motorolla DragonBall or the ARM7TDMI. The uClinux web site is found at `http://www.uclinux.org/`.

[4]`http://www.uclibc.org/dists/`

to support a complete Linux distribution. People now use uClibc to build versions of Gentoo, Slackware, Linux from Scratch, rescue disks, and even live Linux CDs[5]. A number of commercial products have also been released using uClibc, such as wireless routers, network attached storage devices, DVD players, etc.

4 Compiling uClibc

Before we can compile uClibc, we must first grab a copy of the source code and unpack it so it is ready to use. For this paper, we will just grab a copy of the daily uClibc snapshot.

```
# SITE=http://www.uclibc.org/downloads
# wget -q $SITE/uClibc-snapshot.tar.bz2

# tar -xjf uClibc-snapshot.tar.bz2
# cd uClibc
```

uClibc requires a configuration file, `.config`, that can be edited to change the way the library is compiled, such as to enable or disable features (i.e. whether debugging support is enabled or not), to select a cross-compiler, etc. The preferred method when starting from scratch is to run `make defconfig` followed by `make menuconfig`. Since we are going to be targeting a standard Intel compatible x86 system, no changes to the default configuration file are necessary.

5 The Origins of BusyBox

As I mentioned earlier, the two components of an embedded Linux that I chose to work towards reducing in size were the shared libraries and the set common application programs. A typical Linux system contains a variety of command-line utilities from numerous different organizations and independent programmers. Among the most prominent of these utilities were GNU shellutils, fileutils, textutils (now combined to form GNU coreutils), and similar programs that can be run within a shell (commands such as `sed`, `grep`, `ls`, etc.). The GNU utilities are generally very high-quality programs, and are almost without exception very, very feature-rich. The large feature set comes at the cost of being quite large—prohibitively large for an embedded Linux system. After some investigation, I determined that it would be more efficient to replace them rather than try to strip them down, so I began looking at alternatives.

Just as with alternative C libraries, there were several choices for small shell utilities: BSD has a number of utilities which could be used. The Minix operating system, which had recently released under a free software license, also had many useful utilities. Sash, the stand alone shell, was also a possibility. After quite a lot of research, the one that seemed to be the best fit was BusyBox. It also appealed to me because I was already familiar with BusyBox from its use on the Debian boot floppies, and because I was acquainted with Bruce Perens, who was the maintainer. Starting approximately in October 1999, I began enhancing BusyBox and fixing the most obvious problems. Since Bruce was otherwise occupied and was no longer actively maintaining BusyBox, Bruce eventually consented to let me take over maintainership.

Since that time, BusyBox has gained a large following and attracted development talent from literally the whole world. It has been used in commercial products such as the IBM Linux wristwatch, the Sharp Zaurus PDA, and Linksys wireless routers such as the WRT54G, with many more products being released all the time. So many new features and applets have been added to BusyBox, that the biggest chal-

[5]Puppy Linux available from `http://www.goosee.com/puppy/` is a live linux CD system built with uClibc that includes such favorites as XFree86 and Mozilla.

lenge I now face is simply keeping up with all of the patches that get submitted!

6 So, How Does It Work?

BusyBox is a multi-call binary that combines many common Unix utilities into a single executable. When it is run, BusyBox checks if it was invoked via a symbolic link (a `symlink`), and if the name of the symlink matches the name of an applet that was compiled into BusyBox, it runs that applet. If BusyBox is invoked as `busybox`, then it will read the command line and try to execute the applet name passed as the first argument. For example:

```
# ./busybox date
Wed Jun 2 15:01:03 MDT 2004

# ./busybox echo "hello there"
hello there

# ln -s ./busybox uname
# ./uname
Linux
```

BusyBox is designed such that the developer compiling it for an embedded system can select exactly which applets to include in the final binary. Thus, it is possible to strip out support for unneeded and unwanted functionality, resulting in a smaller binary with a carefully selected set of commands. The customization granularity for BusyBox even goes one step further: each applet may contain multiple features that can be turned on or off. Thus, for example, if you do not wish to include large file support, or you do not need to mount NFS filesystems, you can simply turn these features off, further reducing the size of the final BusyBox binary.

7 Compiling Busybox

Let's walk through a normal compile of BusyBox. First, we must grab a copy of the BusyBox source code and unpack it so it is ready to use. For this paper, we will just grab a copy of the daily BusyBox snapshot.

```
# SITE=http://www.busybox.net/downloads
# wget -q $SITE/busybox-snapshot.tar.bz2
# tar -xjf busybox-snapshot.tar.bz2
# cd busybox
```

Now that we are in the BusyBox source directory we can configure BusyBox so that it meets the needs of our embedded Linux system. This is done by editing the file `.config` to change the set of applets that are compiled into BusyBox, to enable or disable features (i.e. whether debugging support is enabled or not), and to select a cross-compiler. The preferred method when starting from scratch is to run `make defconfig` followed by `make menuconfig`. Once BusyBox has been configured to taste, you just need to run `make` to compile it.

8 Installing Busybox to a Target

If you then want to install BusyBox onto a target device, this is most easily done by typing: `make install`. The installation script automatically creates all the required directories (such as `/bin`, `/sbin`, and the like) and creates appropriate symlinks in those directories for each applet that was compiled into the BusyBox binary.

If we wanted to install BusyBox to the directory /mnt, we would simply run:

```
# make PREFIX=/mnt install

[--installation text omitted--]
```

9 Let's build something that works!

Now that I have certainly bored you to death, we finally get to the fun part, building our own embedded Linux system. For hardware, I will be using a Soekris 4521 system[6] with an 133 Mhz AMD Elan CPU, 64 MB main memory, and a generic Intersil Prism based 802.11b card that can be driven using the `hostap`[7] driver. The root filesystem will be installed on a compact flash card.

To begin with, we need to create toolchain with which to compile the software for our wireless access point. This requires we first compile GNU binutils[8], then compile the GNU compiler collection—gcc[9], and then compile uClibc using the newly created gcc compiler. With all those steps completed, we must finally recompile gcc using using the newly built uClibc library so that `libgcc_s` and `libstdc++` can be linked with uClibc.

Fortunately, the process of creating a uClibc toolchain can be automated. First we will go to the uClibc website and obtain a copy of the uClibc `buildroot` by going here:

```
http://www.uclibc.org/cgi-bin/
cvsweb/buildroot/
```

and clicking on the "Download tarball" link[10]. This is a simple GNU make based build system which first builds a uClibc toolchain, and then builds a root filesystem using the newly built uClibc toolchain.

For the root filesystem of our wireless access point, we will need a Linux kernel, uClibc, BusyBox, pcmcia-cs, iptables, hostap, wtools, bridgeutils, and the dropbear ssh server. To compile these programs, we will first edit the buildroot Makefile to enable each of these items. Figure 1 shows the changes I made to the buildroot Makefile:

Running `make` at this point will download the needed software packages, build a toolchain, and create a minimal root filesystem with the specified software installed.

On my system, with all the software packages previously downloaded and cached locally, a complete build took 17 minutes, 19 seconds. Depending on the speed of your network connection and the speed of your build system, now might be an excellent time to take a lunch break, take a walk, or watch a movie.

10 Checking out the new Root Filesystem

We now have our root filesystem finished and ready to go. But we still need to do a little more work before we can boot up our newly built embedded Linux system. First, we need to compress our root filesystem so it can be loaded as an initrd.

```
# gzip -9 root_fs_i386
# ls -sh root_fs_i386.gz
1.1M root_fs_i386.gz
```

Now that our root filesystem has been compressed, it is ready to install on the boot media. To make things simple, I will install the Compact Flash boot media into a USB card reader device, and copy files using the card reader.

```
# ms-sys -s /dev/sda
Public domain master boot record
successfully written to /dev/sda
```

[6] `http://www.soekris.com/net4521.htm`

[7] `http://hostap.epitest.fi/`

[8] `http://sources.redhat.com/binutils/`

[9] `http://gcc.gnu.org/`

[10] `http://www.uclibc.org/cgi-bin/cvsweb/buildroot.tar.gz?view=tar`

```
--- Makefile
+++ Makefile
@@ -140,6 +140,6 @@
 # Unless you want to build a kernel, I recommend just using
 # that...
-TARGETS+=kernel-headers
-#TARGETS+=linux
+#TARGETS+=kernel-headers
+TARGETS+=linux
 #TARGETS+=system-linux

@@ -150,5 +150,5 @@
 #TARGETS+=zlib openssl openssh
 # Dropbear sshd is much smaller than openssl + openssh
-#TARGETS+=dropbear_sshd
+TARGETS+=dropbear_sshd

 # Everything needed to build a full uClibc development system!
@@ -175,5 +175,5 @@

 # Some stuff for access points and firewalls
-#TARGETS+=iptables hostap wtools dhcp_relay bridge
+TARGETS+=iptables hostap wtools dhcp_relay bridge
 #TARGETS+=iproute2 netsnmp
```

Figure 1: Changes to the buildroot Makefile

```
# mkdosfs /dev/sda1
mkdosfs 2.10 (22 Sep 2003)

# syslinux /dev/sda1

# cp root_fs_i386.gz /mnt/root_fs.gz

# cp build_i386/buildroot-kernel /mnt/linux
```

So we now have a copy of our root filesystem and Linux kernel on the compact flash disk. Finally, we need to configure the bootloader. In case you missed it a few steps ago, we are using the syslinux bootloader for this example. I happen to have a ready to use syslinux configuration file, so I will now install that to the compact flash disk as well:

```
# cat  syslinux.cfg
TIMEOUT 0
PROMPT 0
DEFAULT linux
LABEL linux
 KERNEL linux
 APPEND initrd=root_fs.gz \
        console=ttyS0,57600 \
        root=/dev/ram0 boot=/dev/hda1,msdos rw

# cp syslinux.cfg /mnt
```

And now, finally, we are done. Our embedded Linux system is complete and ready to boot. And you know what? It is very, very small. Take a look at Table 1.

With a carefully optimized Linux kernel (which this kernel unfortunately isn't) we could expect to have even more free space. And remember, every bit of space we save is money that embedded Linux developers don't have to spend on expensive flash memory. So now comes the final test; it is now time to boot from our compact flash disk. Here is what you should see.

```
[----kernel boot messages snipped--]
```

```
# ll /mnt
total 1.9M
drwxr-r-     2  root  root   16K  Jun  2  16:39  ./
drwxr-xr-x  22  root  root  4.0K  Feb  6  07:40  ../
-r-xr-r-     1  root  root  7.7K  Jun  2  16:36  ldlinux.sys*
-rwxr-r-     1  root  root  795K  Jun  2  16:36  linux*
-rwxr-r-     1  root  root  1.1M  Jun  2  16:36  root_fs.gz*
-rwxr-r-     1  root  root   170  Jun  2  16:39  syslinux.cfg*
```

Table 1: Output of `ls -lh /mnt`.

```
Freeing unused kernel memory: 64k freed

Welcome to the Erik's wireless access point.

uclibc login: root

BusyBox v1.00-pre10 (2004.06.02-21:54+0000)
Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# du -h / | tail -n 1
2.6M

#
```

And there you have it—your very own wireless access point. Some additional configuration will be necessary to start up the wireless interface, which will be demonstrated during my presentation.

11 Conclusion

The two largest components of a standard Linux system are the utilities and the libraries. By replacing these with smaller equivalents a much more compact system can be built. Using BusyBox and uClibc allows you to customize your embedded distribution by stripping out unneeded applets and features, thus further reducing the final image size. This space savings translates directly into decreased cost per unit as less flash memory will be required. Combine this with the cost savings of using Linux, rather than a more expensive proprietary OS, and the reasons for using Linux become very compelling. The example Wireless Access point we created is a simple but useful example. There are thousands of other potential applications that are only waiting for you to create them.

Run-time testing of LSB Applications

Stuart Anderson
Free Standards Group
anderson@freestandards.org

Matt Elder
University of South Caroilina
happymutant@sc.rr.com

Abstract

The dynamic application test tool is capable of checking API usage at run-time. The LSB defines only a subset of all possible parameter values to be valid. This tool is capable of checking these value while the application is running.

This paper will explain how this tool works, and highlight some of the more interesting implementation details such as how we managed to generate most of the code automatically, based on the interface descriptions contained in the LSB database.

Results to date will be presented, along with future plans and possible uses for this tool.

1 Introduction

The Linux Standard Base (LSB) Project began in 1998, when the Linux community came together and decided to take action to prevent GNU/Linux based operating systems from fragmenting in the same way UNIX operating systems did in the 1980s and 1990s. The LSB defines the Application Binary Interface (ABI) for the core part of a GNU/Linux system. As an ABI, the LSB defines the interface between the operating system and the applications. A complete set of tests for an ABI must be capable of measuring the interface from both sides.

Almost from the beginning, testing has been a cornerstone of the project. The LSB was originally organized around 3 components: the written specification, a sample implementation, and the test suites. The written specification is the ultimate definition of the LSB. Both the sample implementation, and the test suites yield to the authority of the written specification.

The sample implementation (SI) is a minimal subset of a GNU/Linux system that provides a runtime that implements the LSB, and as little else as possible. The SI is neither intended to be a minimal distribution, nor the basis for a distribution. Instead, it is used as both a proof of concept and a testing tool. Applications which are seeking certification are required to prove they execute correctly using the SI and two other distributions. The SI is also used to validate the runtime test suites.

The third component is testing. One of the things that strengthens the LSB is its ability to measure, and thus prove, conformance to the standard. Testing is achieved with an array of different test suites, each of which measures a different aspect of the specification.

LSB Runtime

- `cmdchk`

  This test suite is a simple existence test that ensures the required LSB commands and utilities are found on an LSB conforming system.

- `libchk`

  This test suite checks the libraries required by the LSB to ensure they contain the interfaces and symbol versions as specified by the LSB.

- `runtimetests`

  This test suite measures the behavior of the interfaces provided by the GNU/Linux system. This is the largest of the test suites, and is actually broken down into several components, which are referred to collectively as the runtime tests. These tests are derived from the test suites used by the Open Group for UNIX branding.

LSB Packaging

- `pkgchk`

  This test examines an RPM format package to ensure it conforms to the LSB.

- `pkginstchk`

  This test suite is used to ensure that the package management tool provided by a GNU/Linux system will correctly install LSB conforming packages. This suite is still in early stages of development.

LSB Application

- `appchk`

  This test performs a static analysis of an application to ensure that it only uses libraries and interfaces specified by the LSB.

- `dynchk`

  This test is used to measure an applications use of the LSB interfaces during its execution, and is the subject of this paper.

2 The database

The LSB Specification contains over 6600 interfaces, each of which is associated with a library and a header file, and may have parameters. Because of the size and complexity of the data describing these interfaces, a database is used to maintain this information.

It is impractical to try and keep the specification, test suites and development libraries and headers synchronized for this much data. Instead, portions of the specification and tests, and all of the development headers and libraries are generated from the database. This ensures that as changes are made to the database, the changes are propagated to the other parts of the project as well.

Some of the relevant data components in this DB are Libraries, Headers, Interfaces, and Types. There are also secondary components and relations between all of the components. A short description of some of these is needed before moving on to how the dynchk test is constructed.

2.1 Library

The LSB specifies 17 shared libraries, which contains the 6600 interfaces. The interfaces in each library are grouped into logical units called a LibGroup. The LibGroups help to organize the interfaces, which is very useful in the written specification, but isn't used much elsewhere.

2.2 Interface

An Interface represents a globally visible symbol, such as a function, or piece of data. Interfaces have a Type, which is either the type of the global data or the return type of the function. If the Interface is a function, then it will have zero or more Parameters, which form a

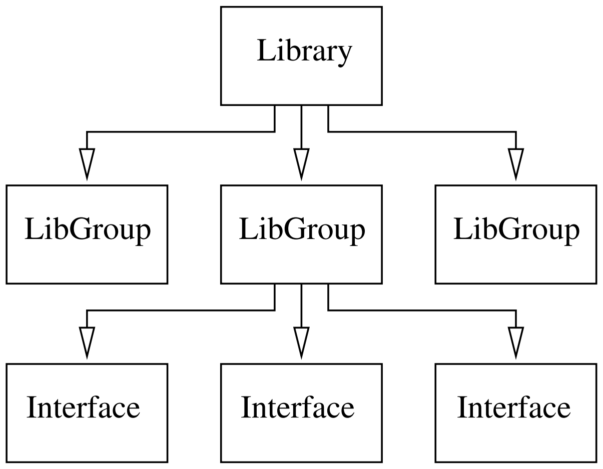


Figure 1: Relationship between Library, LibGroup and Interface

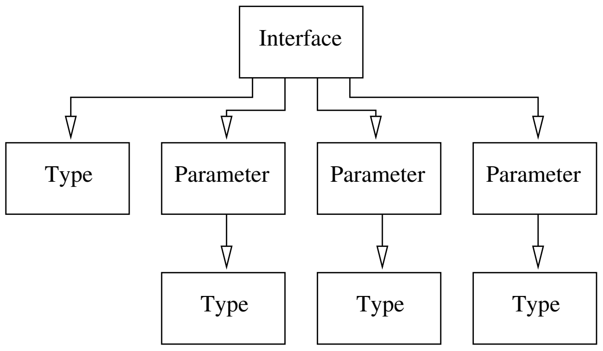


Figure 2: Relationship between Interface, Type and Parameter

set of Types ordered by their position in the parameter list.

2.3 Type

As mentioned above, the database contains enough information to be able to generate header files which are a part of the LSB development tools. This means that the database must be able to represent Clanguage types. The Type and TypeMember tables provide these. These tables are used recursively. If a Type is defined in terms of another type, then it will have a base type that points to that other type.

For structs and unions, the TypeMemeber table

Tid	Ttype	Tname	Tbasetype
1	Intrinsic	int	0
2	Pointer	int *	1

Table 1: Example of recursion in Type table for int *

```
struct foo {
  int    a;
  int   *b;
}
```

Figure 3: Sample struct

is used to hold the ordered list of members. Entries in the TypeMember table point back to the Type table to describe the type of each member. For enums, the TypeMember table is also used to hold the ordered list of values.

Tid	Ttype	Tname	Tbasetype
1	Intrinsic	int	0
2	Pointer	int *	1
3	Struct	foo	0

Table 2: Contents of Type table

The structure shown in Figure 3 is represented by the entries in the Type table in Table 2 and the TypeMember table in Table 3.

2.4 Header

Headers, like Libraries, have their contents arranged into logical groupings known a HeaderGroups. Unlike Libraries, these HeaderGroups are ordered so that the proper sequence of definitions within a header file can be maintained. HeaderGroups contain Constant definitions (i.e. #define statements) and Type definitions. If you examine a few well designed header files, you will notice a pattern of a comment followed by related constant definitions and type definitions. The entire header file can be viewed as a repeating sequence of this pat-

Tmid	TMname	TMtypeid	TMposition	TMmemberof
10	a	1	0	3
11	b	2	1	3

Table 3: Contents of TypeMember

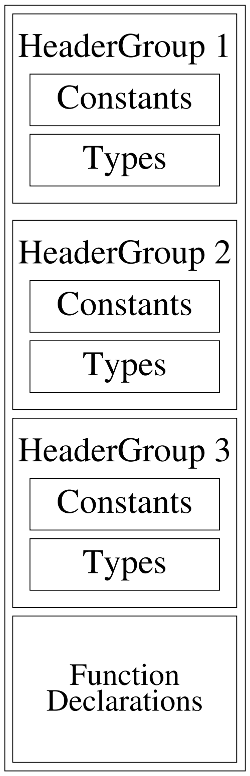


Figure 4: Organization of Headers

tern. This pattern is the basis for the HeaderGroup concept.

2.5 TypeType

One last construct in our database should be mentioned. While we are able to represent a syntactic description of interfaces and types in the database, this is not enough to automatically generate meaningful test cases. We need to add some semantic information that better describes how the types in structures and parameters are used. As an example, `struct sockaddr` contains a member, `sa_family`, of type unsigned short. The compiler will of course ensure that only values between 0 and $2^{16} - 1$ will be used, but only a few of those values have any meaning in this context. By adding the semantic information that this member holds a socket family value, the test generator can cause the value found in `sa_family` to be tested against the legal socket families values (`AF_INET`, `AF_INET6`, etc), instead of just ensuring the value falls between 0 and $2^{16} - 1$, which is really just a noop test.

Example TypeType entries

- `RWaddress`

  An address from the process space that must be both readable and writable.

- `Rdaddress`

  An address from the process space that must be at least readable.

- `filedescriptor`

  A small integer value greater than or equal to 0, and less than the maximum file descriptor for the process.

- `pathname`

  The name of a file or directory that should be compared against the Filesystem Hierarchy Standard.

2.6 Using this data

As mentioned above, the data in the database is used to generate different portions of the LSB project. This strategy was adopted to ensure these different parts would always be in sync, without having to depend on human intervention.

The written specification contains tables of interfaces, and data definitions (constants and types). These are all generated from the database.

The LSB development environment[1] consists of stub libraries and header files that contain only the interfaces defined by the LSB. This development environment helps catch the use of non-LSB interfaces during the development or porting of an application instead of being surprised by later test results. Both the stub libraries and headers are produced by scripts pulling data from the database.

Some of the test suites described previously have components which are generated from the database. `Cmdchk` and `libchk` have lists of commands and interfaces respectively which are extracted from the database. The static application test tool, `appchk`, also has a list of interfaces that comes from the database. The dynamic application test tool, `dynchk`, has the majority of its code generated from information in the database.

3 The Dynamic Checker

The static application checker simply examines an executable file to determine if it is using interfaces beyond those allowed by the LSB. This is very useful to determine if an application has been built correctly. However, is unable to determine if the interfaces are used correctly when the application is executed. A different kind of test is required to be able to perform this level of checking. This new test must interact with the application while it is running, without interfering with the execution of the application.

This new test has two major components: a mechanism for hooking itself into an application, and a collection of functions to perform the tests for all of the interfaces. These components can mostly be developed independently of each other.

3.1 The Mechanism

The mechanism for interacting with the application must be transparent and noninterfering to the application. We considered the approach used by 3 different tools: abc, ltrace, and fakeroot.

- `abc`—This tool was the inspiration for our new dynamic checker. `abc` was developed as part of the SVR4 ABI test tools. `abc` works by modifying the target application. The application's executable is modified to load a different version of the shared libraries and to call a different version of each interface. This is accomplished by changing the strings in the symbol table and `DT_NEEDED` records. For example, `libc.so.1` is changed to `LiBc.So.1`, and `fread()` is changed to `FrEaD()`. The test set is then located in `/usr/lib/LiBc.So.1`, which in turns loads the original `/usr/lib/libc.so.1`. This mechanism works, but the requirement to modify the executable file is undesirable.
- `ltrace`—This tool is similar to `strace`, except that it traces calls into shared libraries instead of calls into the kernel. `ltrace` uses the ptrace interface to control the application's process. With this approach, the test sets are located in a separate program and are invoked by stopping the application upon

[1]See the May Issue of *Linux Journal* for more information on the LSB Development Environment.

entry to the interface being tested. This approach has two drawbacks: first, the code required to decode the process stack and extract the parameters is unique to each architecture, and second, the tests themselves are more complicated to write since the parameters have to be fetched from the application's process.

- `fakeroot`—This tool is used to create an environment where an unprivileged process appears to have root privileges. `fakeroot` uses `LD_PRELOAD` to load an additional shared library before any of the shared libraries specified by the `DT_NEEDED` records in the executable. This extra library contains a replacement function for each file manipulation function. The functions in this library will be selected by the dynamic linker instead of the normal functions found in the regular libraries. The test sets themselves will perform tests of the parameters, and then call the original version of the functions.

We chose to use the `LD_PRELOAD` mechanism because we felt it was the simplest to use. Based on this mechanism, a sample test case looks like Figure 5.

One problem that must be avoided when using this mechanism is recursion. If the above function just called `read()` at the end, it would end up calling itself again. Instead, the `RTLD_NEXT` flag passed to `dlsym()` tells the dynamic linker to look up the symbol on one of the libraries loaded after the current library. This will get the original version of the function.

3.2 Test set organization

The test set functions are organized into 3 layers. The top layer contains the functions that are test stubs for the LSB interfaces. These functions are implemented by calling the functions in layers 2 and 3. An example of a function in the first layer was given in Figure 5.

The second layer contains the functions that test data structures and types which are passed in as parameters. These functions are also implemented by calling the functions in layer 3 and other functions in layer 2. A function in the second layer looks like Figure 6.

The third layer contains functions that test the types which have been annotated with additional semantic information. These functions often have to perform nontrivial operations to test the assertion required for these supplemental types. Figure 7 is an example of a layer 3 function.

Presently, there are 3056 functions in layer 1 (tests for `libstdc++` are not yet being generated), 106 functions in layer 2, and just a few in layer 3. We estimate that the total number of functions in layer 3 upon completion of the test tool will be on the order of several dozen. The functions in the first two layers are automatically generated based on the information in the database. Functions in layer 3 are hand coded.

3.3 Automatic generation of the tests

In Table 4, is a summary of the size of the test tool so far. As work progresses, these numbers will only get larger. Most of the code in the test is very repetitive, and prone to errors when edited manually. The ability to automate the process of creating this code is highly desirable.

Let's take another look at the sample function from layer 1. This time, however, lets replace some of the code with a description of the information it represents. See Figure 8 for this parameterized version.

All of the occurrences of the string *read* are

```
ssize_t read (int arg0, void *arg1, size_t arg2) {
  if (!funcptr)
      funcptr = dlsym(RTLD_NEXT, "read");
  validate_filedescriptor(arg0, "read");
  validate_RWaddress(arg1, "read");
  validate_size_t(arg2, "read");
  return funcptr(arg0, arg1, arg2);
}
```

Figure 5: Test case for read() function

```
void validate_struct_sockaddr_in(struct sockaddr_in *input,
                                 char *name) {
  validate_socketfamily(input->sin_family,name);
  validate_socketport(input->sin_port,name);
  validate_IPv4Address((input->sin_addr), name);
}
```

Figure 6: Test case for validating `struct sockaddr_in`

Module	Files	Lines of Code
libc	752	19305
libdl	5	125
libgcc_s	13	262
libGL	450	11046
libICE	49	1135
libm	281	6568
libncurses	266	6609
libpam	13	335
libpthread	82	2060
libSM	37	865
libX11	668	16112
libXext	113	2673
libXt	288	7213
libz	39	973
structs	106	1581

Table 4: Summary of generated code

actually just the function name, and could have been replaced also.

The same thing can be done for the sample function from layer 2 as is seen in Figure 9.

These two examples, now represent templates that can be used to create the functions for layers 1 and 2. From the previous description of the database, you can see that there is enough information available to be able to instantiate these templates for each interfaces, and structure used by the LSB.

The automation is implemented by 2 perl scripts: `gen_lib.pl` and `gen_tests.pl`. These scripts generate the code for layers 1 and 2 respectively.

Overall, these scripts work well, but we have run into a few interesting situations along the way.

3.4 Handling the exceptions

So far, we have come up with an overall architecture for the test tool, selected a mechanism that allows us to hook the tests into the running application, discovered the pattern in the test functions so that we could create a template for

```
void validate_filedescriptor(const int fd, const char *name) {
   if (fd >= lsb_sysconf(_SC_OPEN_MAX))
       ERROR("fd too big");
   else if (fd < 0)
       ERROR("fd negative");
   }
```

Figure 7: Test case for validating a filedescriptor

```
 return-type read (list of parameters) {
     if (!funcptr)
         funcptr = dlsym(RTLD_NEXT, "read");
     validate_parameter1 type(arg0, "read");
     validate_parameter2 type(arg1, "read");
     validate_parameter3 type(arg2, "read");
     return funcptr(arg0, arg1, arg2);
 }
```

Figure 8: Parameterized test case for a function

automatically generating the code, and implemented the scripts to generate all of the tests cases. The only problem is that now we run into the real world, where things don't always follow the rules.

Here are a few of the interesting situations we have encountered

- `Variadic Functions`

  Of the 725 functions in libc, 25 of them take a variable number of parameters. This causes problems in the generation of the code for the test case, but most importantly it affects our ability to know how to process the arguments. These function have to be written by hand to handle the special needs of these functions. For the functions in the `exec`, `printf` and `scanf` families, the test cases can be implemented by calling the varargs form of the function (`execl()` can be implemented using `execv()`).

- `open()`

  In addition to the problems of being a variadic function, the third parameter to `open()` and `open64()` is only valid if the `O_CREAT` flag is set in the second parameter to these functions. This simple exception requires a small amount of manual intervention, so these function have to be maintained by hand.

- `memory allocation`

  One of the recursion problems we ran into is that memory will be allocated within the `dlsym()` function call, so the implementation of one test case ends up invoking the test case for one of the memory allocation routines, which by default would call `dlsym()`, creating the recursion. This cycle had to be broken by having the test cases for these routines call libc private interfaces to memory allocation.

- `changing memory map`

```
void validate_struct_structure name(struct structure name
   *input, char *name) {
     validate_type of member 1(input->name of member 1, name);
     validate_type of member 2(input->name of member 2, name);
     validate_type of member 3((input->name of member 3), name);
}
```

Figure 9: Parameterized test case for a struct

Pointers are validated by making sure they contain an address that is valid for the process. `/proc/self/maps` is read to obtain the memory map of the current process. These results are cached, for performance reasons, but usually, the memory map of the process will change over time. Both the stack and the heap will grow, resulting in valid pointers being checked against a cached copy of the memory map. In the event a pointer is found to be invalid, the memory map is re-read, and the pointer checked again. The `mmap()` and `munmap()` test cases are also maintained by hand so that they can also cause the memory map to be re-read.

- `hidden ioctl()s`

  By design, the LSB specifies interfaces at the highest possible level. One example of this, is the use of the termio functions, instead of specifying the underlying `ioctl()` interface. It turns out that this tool catches the underlying `ioctl()` calls anyway, and flags it as an error. The solution is for the termio functions the set a flag indicating that the `ioctl()` test case should skip its tests.

- `Optionally NULL parameters`

  Many interfaces have parameters which may be NULL. This triggerred lots of warnings for many programs. The solution was to add a flag that indicated that the Parameter may be NULL, and to not try to validate the pointer, or the data being pointed to.

No doubt, there will be more interesting situations to have to deal with before this tool is completed.

4 Results

As of the deadline for this paper, results are preliminary, but encouraging. The tool is initially being tested against simple commands such as ls and vi, and some X Windows clients such as xclock and xterm. The tool is correctly inserting itself into the application under test, and we are getting some interesting results that will be examined more closely.

One example is vi passes a NULL to `__strtol_internal` several times during startup.

The tool was designed to work across all architectures. At present, it has been built and tested on only the IA32 and IA64 architectures. No significant problems are anticipate on other architectures.

Additional results and experience will be presented at the conference.

5 Future Work

There is still much work to be done. Some of the outstanding tasks are highlighted here.

- Additional `TypeTypes`

  Semantic information needs to be added for additional parameters and structures. The additional layer 3 tests that correspond to this information must also be implemented.

- Architecture-specific interfaces

  As we found in the LSB, there are some interfaces, and types that are unique to one or more architectures. These need to be handled properly so they are not part of the tests when built on an architecture for which they don't apply.

- Unions

  Although Unions are represented in the database in the same way as structures, the database does not contain enough information to describe how to interpret or test the contents of a union. Test cases that involve unions may have to be written by hand.

- Additional libraries

  The information in the database for the graphics libraries and for `libstdc++` is incomplete, therefore, it is not possible to generate all of the test cases for those libraries. Once the data is complete, the test cases will also be complete.

Linux Block IO—present and future

Jens Axboe
SuSE
`axboe@suse.de`

Abstract

One of the primary focus points of 2.5 was fixing up the bit rotting block layer, and as a result 2.6 now sports a brand new implementation of basically anything that has to do with passing IO around in the kernel, from producer to disk driver. The talk will feature an in-depth look at the IO core system in 2.6 comparing to 2.4, looking at performance, flexibility, and added functionality. The rewrite of the IO scheduler API and the new IO schedulers will get a fair treatment as well.

No 2.6 talk would be complete without 2.7 speculations, so I shall try to predict what changes the future holds for the world of Linux block I/O.

1 2.4 Problems

One of the most widely criticized pieces of code in the 2.4 kernels is, without a doubt, the block layer. It's bit rotted heavily and lacks various features or facilities that modern hardware craves. This has led to many evils, ranging from code duplication in drivers to massive patching of block layer internals in vendor kernels. As a result, vendor trees can easily be considered forks of the 2.4 kernel with respect to the block layer code, with all of the problems that this fact brings with it: 2.4 block layer code base may as well be considered dead, no one develops against it. Hardware vendor drivers include many nasty hacks and `#ifdef's` to work in all of the various 2.4 kernels that are out there, which doesn't exactly enhance code coverage or peer review.

The block layer fork didn't just happen for the fun of it of course, it was a direct result of the various problem observed. Some of these are added features, others are deeper rewrites attempting to solve scalability problems with the block layer core or IO scheduler. In the next sections I will attempt to highlight specific problems in these areas.

1.1 IO Scheduler

The main 2.4 IO scheduler is called `elevator_linus`, named after the benevolent kernel dictator to credit him for some of the ideas used. `elevator_linus` is a one-way scan elevator that always scans in the direction of increasing LBA. It manages latency problems by assigning sequence numbers to new requests, denoting how many new requests (either merges or inserts) may pass this one. The latency value is dependent on data direction, smaller for reads than for writes. Internally, `elevator_linus` uses a double linked list structure (the kernels `struct list_head`) to manage the request structures. When queuing a new IO unit with the IO scheduler, the list is walked to find a suitable insertion (or merge) point yielding an **O(N)** runtime. That in itself is suboptimal in presence of large amounts of IO and to make matters even worse, we repeat this scan if the request free list was empty when we entered the IO scheduler. The latter is not an error condition, it will happen all the time for even moderate amounts of write back against a queue.

1.2 struct buffer_head

The main IO unit in the 2.4 kernel is the `struct buffer_head`. It's a fairly unwieldy structure, used at various kernel layers for different things: caching entity, file system block, and IO unit. As a result, it's suboptimal for either of them.

From the block layer point of view, the two biggest problems is the size of the structure and the limitation in how big a data region it can describe. Being limited by the file system *one block* semantics, it can at most describe a `PAGE_CACHE_SIZE` amount of data. In Linux on x86 hardware that means 4KiB of data. Often it can be even worse: raw io typically uses the soft sector size of a queue (default 1KiB) for submitting io, which means that queuing eg 32KiB of IO will enter the io scheduler 32 times. To work around this limitation and get at least to a page at the time, a 2.4 hack was introduced. This is called `vary_io`. A driver advertising this capability acknowledges that it can manage `buffer_head's` of varying sizes at the same time. File system read-ahead, another frequent user of submitting larger sized io, has no option but to submit the read-ahead window in units of the page size.

1.3 Scalability

With the limit on `buffer_head` IO size and `elevator_linus` runtime, it doesn't take a lot of thinking to discover obvious scalability problems in the Linux 2.4 IO path. To add insult to injury, the entire IO path is guarded by a single, global lock: `io_request_lock`. This lock is held during the entire IO queuing operation, and typically also from the other end when a driver subtracts requests for IO submission. A single global lock is a big enough problem on its own (bigger SMP systems will suffer immensely because of cache line bouncing), but add to that long runtimes and you have a really huge IO scalability problem.

Linux vendors have long shipped lock scalability patches for quite some time to get around this problem. The adopted solution is typically to make the queue lock a pointer to a driver local lock, so the driver has full control of the granularity and scope of the lock. This solution was adopted from the 2.5 kernel, as we'll see later. But this is another case where driver writers often need to differentiate between vendor and vanilla kernels.

1.4 API problems

Looking at the block layer as a whole (including both ends of the spectrum, the producers and consumers of the IO units going through the block layer), it is a typical example of code that has been hacked into existence without much thought to design. When things broke or new features were needed, they had been grafted into the existing mess. No well defined interface exists between file system and block layer, except a few scattered functions. Controlling IO unit flow from IO scheduler to driver was impossible: 2.4 exposes the IO scheduler data structures (the `->queue_head` linked list used for queuing) directly to the driver. This fact alone makes it virtually impossible to implement more clever IO scheduling in 2.4. Even the recently (in the 2.4.20's) added lower latency work was horrible to work with because of this lack of boundaries. Verifying correctness of the code is extremely difficult; peer review of the code likewise, since a reviewer must be intimate with the block layer structures to follow the code.

Another example on lack of clear direction is the partition remapping. In 2.4, it's the driver's responsibility to resolve partition mappings. A given request contains a device and sector offset (i.e. `/dev/hda4`, sector 128) and the driver must map this to an absolute device offset before sending it to the hardware. Not only does this cause duplicate code in the drivers, it also means the IO scheduler has no knowledge of the real device mapping of a particular request. This adversely impacts IO scheduling whenever partitions aren't laid out in strict ascending disk order, since it causes the io scheduler to make the wrong decisions when ordering io.

2 2.6 Block layer

The above observations were the initial kick off for the 2.5 block layer patches. To solve some of these issues the block layer needed to be turned inside out, breaking basically anything-io along the way.

2.1 bio

Given that `struct buffer_head` was one of the problems, it made sense to start from scratch with an IO unit that would be agreeable to the upper layers as well as the drivers. The main criteria for such an IO unit would be something along the lines of:

1. Must be able to contain an arbitrary amount of data, as much as the hardware allows. Or as much that makes *sense* at least, with the option of easily pushing this boundary later.
2. Must work equally well for pages that have a virtual mapping as well as ones that do not.
3. When entering the IO scheduler and driver, IO unit must point to an absolute location on disk.
4. Must be able to stack easily for IO stacks such as raid and device mappers. This includes full redirect stacking like in 2.4, as well as partial redirections.

Once the primary goals for the IO structure were laid out, the `struct bio` was born. It was decided to base the layout on a scatter-gather type setup, with the `bio` containing a map of pages. If the map count was made flexible, items 1 and 2 on the above list were already solved. The actual implementation involved splitting the data container from the `bio` itself into a `struct bio_vec` structure. This was mainly done to ease allocation of the structures so that `sizeof(struct bio)` was always constant. The `bio_vec` structure is simply a tuple of `{page, length, offset}`, and the `bio` can be allocated with room for anything from 1 to `BIO_MAX_PAGES`. Currently Linux defines that as 256 pages, meaning we can support up to 1MiB of data in a single `bio` for a system with 4KiB page size. At the time of implementation, 1MiB was a good deal beyond the point where increasing the IO size further didn't yield better performance or lower CPU usage. It also has the added bonus of making the `bio_vec` fit inside a single page, so we avoid higher order memory allocations (`sizeof(struct bio_vec)` == 12 on 32-bit, 16 on 64-bit) in the IO path. This is an important point, as it eases the pressure on the memory allocator. For swapping or other low memory situations, we ideally want to stress the allocator as little as possible.

Different hardware can support different sizes of io. Traditional parallel ATA can do a maximum of 128KiB per request, qlogicfc SCSI doesn't like more than 32KiB, and lots of high end controllers don't impose a significant limit on max IO size but may restrict the maximum number of segments that one IO may be composed of. Additionally, software raid or device mapper stacks may like special alignment of IO or the guarantee that IO won't cross stripe boundaries. All of this knowledge is either impractical or impossible to statically advertise to submitters of io, so an easy interface for populating a `bio` with pages was essential if supporting large IO was to become practical. The current solution is `int bio_add_page()` which attempts to add a single page (full or partial) to a `bio`. It returns the amount of bytes successfully added. Typical users of this function continue adding pages to a `bio` until it fails—then it is submitted for IO through `submit_bio()`, a new `bio` is allocated and populated until all data has gone out. `int bio_add_page()` uses statically defined parameters inside the request queue to determine how many pages can be added, and attempts to query a registered `merge_bvec_fn` for dynamic limits that the block layer cannot know about.

Drivers hooking into the block layer before the IO scheduler[1] deal with `struct bio` directly, as opposed to the `struct request` that are output after the IO scheduler. Even though the page addition API guarantees that they never need to be able to deal with a `bio` that is too big, they still have to manage local splits at sub-page granularity. The API was defined that way to make it easier for IO submitters to manage, so they don't have to deal with sub-page splits. 2.6 block layer defines two ways to deal with this situation—the first is the general clone interface. `bio_clone()` returns a clone of a `bio`. A clone is defined as a private copy of the `bio` itself, but with a shared `bio_vec` page map list. Drivers can modify the cloned `bio` and submit it to a different device without duplicating the data. The second interface is tailored specifically to single page splits and was written by kernel raid maintainer Neil Brown. The main function is `bio_split()` which returns a `struct bio_pair` describing the two parts of the original `bio`. The two `bio`'s can then be submitted separately by the driver.

[1]Also known as at `make_request` time.

2.2 Partition remapping

Partition remapping is handled inside the IO stack before going to the driver, so that both drivers and IO schedulers have immediate full knowledge of precisely where data should end up. The device unfolding is done automatically by the same piece of code that resolves full `bio` redirects. The worker function is `blk_partition_remap()`.

2.3 Barriers

Another feature that found its way to some vendor kernels is IO barriers. A barrier is defined as a piece of IO that is guaranteed to:

- Be on platter (or safe storage at least) when completion is signaled.
- Not proceed any previously submitted io.
- Not be proceeded by later submitted io.

The feature is handy for journalled file systems, fsync, and any sort of cache bypassing IO[2] where you want to provide guarantees on data order and correctness. The 2.6 code isn't even complete yet or in the Linus kernels, but it has made its way to Andrew Morton's -mm tree which is generally considered a staging area for features. This section describes the code so far.

The first type of barrier supported is a soft barrier. It isn't of much use for data integrity applications, since it merely implies ordering inside the IO scheduler. It is signaled with the `REQ_SOFTBARRIER` flag inside `struct request`. A stronger barrier is the

[2]Such types of IO include `O_DIRECT` or raw.

hard barrier. From the block layer and IO scheduler point of view, it is identical to the soft variant. Drivers need to know about it though, so they can take appropriate measures to correctly honor the barrier. So far the ide driver is the only one supporting a full, hard barrier. The issue was deemed most important for journalled desktop systems, where the lack of barriers and risk of crashes / power loss coupled with ide drives generally always defaulting to write back caching caused significant problems. Since the ATA command set isn't very intelligent in this regard, the ide solution adopted was to issue pre- and post flushes when encountering a barrier.

The hard and soft barrier share the feature that they are both tied to a piece of data (a `bio`, really) and cannot exist outside of data context. Certain applications of barriers would really like to issue a disk flush, where finding out which piece of data to attach it to is hard or impossible. To solve this problem, the 2.6 barrier code added the `blkdev_issue_flush()` function. The block layer part of the code is basically tied to a queue hook, so the driver issues the flush on its own. A helper function is provided for SCSI type devices, using the generic SCSI command transport that the block layer provides in 2.6 (more on this later). Unlike the queued data barriers, a barrier issued with `blkdev_issue_flush()` works on all interesting drivers in 2.6 (IDE, SCSI, SATA). The only missing bits are drivers that don't belong to one of these classes—things like **CISS** and **DAC960**.

2.4 IO Schedulers

As mentioned in section 1.1, there are a number of known problems with the default 2.4 IO scheduler and IO scheduler interface (or lack thereof). The idea to base latency on a unit of data (sectors) rather than a time based unit is hard to tune, or requires auto-tuning at runtime and this never really worked out. Fixing the runtime problems with `elevator_linus` is next to impossible due to the data structure exposing problem. So before being able to tackle any problems in that area, a neat API to the IO scheduler had to be defined.

2.4.1 Defined API

In the spirit of avoiding over-design[3], the API was based on initial adaption of `elevator_linus`, but has since grown quite a bit as newer IO schedulers required more entry points to exploit their features.

The core function of an IO scheduler is, naturally, insertion of new io units and extraction of ditto from drivers. So the first 2 API functions are defined, `next_req_fn` and `add_req_fn`. If you recall from section 1.1, a new IO unit is first attempted merged into an existing request in the IO scheduler queue. And if this fails and the newly allocated request has raced with someone else adding an adjacent IO unit to the queue in the mean time, we also attempt to merge `struct requests`. So 2 more functions were added to cater to these needs, `merge_fn` and `merge_req_fn`. Cleaning up after a successful merge is done through `merge_cleanup_fn`. Finally, a defined IO scheduler can provide init and exit functions, should it need to perform any duties during queue init or shutdown.

The above described the IO scheduler API as of 2.5.1, later on more functions were added to further abstract the IO scheduler away from the block layer core. More details may be found in the `struct elevator_s` in `<linux/elevator.h>` kernel include file.

[3]Some might, rightfully, claim that this is worse than no design

2.4.2 deadline

In kernel 2.5.39, `elevator_linus` was finally replaced by something more appropriate, the *deadline* IO scheduler. The principles behind it are pretty straight forward — new requests are assigned an expiry time in milliseconds, based on data direction. Internally, requests are managed on two different data structures. The sort list, used for inserts and front merge lookups, is based on a red-black tree. This provides **O(log n)** runtime for both insertion and lookups, clearly superior to the doubly linked list. Two FIFO lists exist for tracking request expiry times, using a double linked list. Since strict FIFO behavior is maintained on these two lists, they run in **O(1)** time. For back merges it is important to maintain good performance as well, as they dominate the total merge count due to the layout of files on disk. So **deadline** added a merge hash for back merges, ideally providing **O(1)** runtime for merges. Additionally, **deadline** adds a one-hit merge cache that is checked even before going to the hash. This gets surprisingly good hit rates, serving as much as 90% of the merges even for heavily threaded io.

Implementation details aside, **deadline** continues to build on the fact that the fastest way to access a single drive, is by scanning in the direction of ascending sector. With its superior runtime performance, **deadline** is able to support very large queue depths without suffering a performance loss or spending large amounts of time in the kernel. It also doesn't suffer from latency problems due to increased queue sizes. When a request expires in the FIFO, **deadline** jumps to that disk location and starts serving IO from there. To prevent accidental seek storms (which would further cause us to miss deadlines), **deadline** attempts to serve a number of requests from that location before jumping to the next expired request. This means that the assigned request deadlines are soft, not a specific hard target that must be met.

2.4.3 Anticipatory IO scheduler

While **deadline** works very well for most workloads, it fails to observe the natural dependencies that often exist between synchronous reads. Say you want to list the contents of a directory—that operation isn't merely a single sync read, it consists of a number of reads where only the completion of the final request will give you the directory listing. With **deadline**, you could get decent performance from such a workload in presence of other IO activities by assigning very tight read deadlines. But that isn't very optimal, since the disk will be serving other requests in between the dependent reads causing a potentially disk wide seek every time. On top of that, the tight deadlines will decrease performance on other io streams in the system.

Nick Piggin implemented an anticipatory IO scheduler [Iyer] during 2.5 to explore some interesting research in this area. The main idea behind the anticipatory IO scheduler is a concept called *deceptive idleness*. When a process issues a request and it completes, it might be ready to issue a new request (possibly close by) immediately. Take the directory listing example from above—it might require 3–4 IO operations to complete. When each of them completes, the process[4] is ready to issue the next one almost instantly. But the traditional io scheduler doesn't pay any attention to this fact, the new request must go through the IO scheduler and wait its turn. With **deadline**, you would have to typically wait 500 milliseconds for each read, if the queue is held busy by other processes. The result is poor interactive performance for each process, even though overall throughput might be acceptable or even good.

[4]Or the kernel, on behalf of the process.

Instead of moving on to the next request from an unrelated process immediately, the anticipatory IO scheduler (hence forth known as **AS**) opens a small window of opportunity for that process to submit a new IO request. If that happens, **AS** gives it a new chance and so on. Internally it keeps a decaying histogram of IO *think times* to help the anticipation be as accurate as possible.

Internally, **AS** is quite like **deadline**. It uses the same data structures and algorithms for sorting, lookups, and FIFO. If the think time is set to 0, it is very close to **deadline** in behavior. The only differences are various optimizations that have been applied to either scheduler allowing them to diverge a little. If **AS** is able to reliably predict when waiting for a new request is worthwhile, it gets phenomenal performance with excellent interactiveness. Often the system throughput is sacrificed a little bit, so depending on the workload **AS** might not be the best choice always. The IO storage hardware used, also plays a role in this—a non-queuing ATA hard drive is a much better fit than a SCSI drive with a large queuing depth. The SCSI firmware reorders requests internally, thus often destroying any accounting that **AS** is trying to do.

2.4.4 CFQ

The third new IO scheduler in 2.6 is called **CFQ**. It's loosely based on the ideas on stochastic fair queuing (SFQ [McKenney]). **SFQ** is fair as long as its hashing doesn't collide, and to avoid that, it uses a continually changing hashing function. Collisions can't be completely avoided though, frequency will depend entirely on workload and timing. **CFQ** is an acronym for completely fair queuing, attempting to get around the collision problem that **SFQ** suffers from. To do so, **CFQ** does away with the fixed number of buckets that processes can be placed in. And using regular hashing technique to find the appropriate bucket in case of collisions, fatal collisions are avoided.

CFQ deviates radically from the concepts that **deadline** and **AS** is based on. It doesn't assign deadlines to incoming requests to maintain fairness, instead it attempts to divide bandwidth equally among classes of processes based on some correlation between them. The default is to hash on thread group id, tgid. This means that bandwidth is attempted distributed equally among the processes in the system. Each class has its own request sort and hash list, using red-black trees again for sorting and regular hashing for back merges. When dealing with writes, there is a little catch. A process will almost never be performing its own writes—data is marked dirty in context of the process, but write back usually takes place from the *pdflush* kernel threads. So **CFQ** is actually dividing read bandwidth among processes, while treating each pdflush thread as a separate process. Usually this has very minor impact on write back performance. Latency is much less of an issue with writes, and good throughput is very easy to achieve due to their inherent asynchronous nature.

2.5 Request allocation

Each block driver in the system has at least one `request_queue_t` request queue structure associated with it. The recommended setup is to assign a queue to each logical spindle. In turn, each request queue has a `struct request_list` embedded which holds free `struct request` structures used for queuing io. 2.4 improved on this situation from 2.2, where a single global free list was available to add one per queue instead. This free list was split into two sections of equal size, for reads and writes, to prevent either direction from starving the other[5]. 2.4 statically allocated a big chunk of requests for each queue, all residing in the precious low memory of a machine. The combination of **O(N)** runtime and statically allocated request structures firmly prevented any real world experimentation with large queue depths on 2.4 kernels.

2.6 improves on this situation by dynamically allocating request structures on the fly instead. Each queue still maintains its request free list like in 2.4. However it's also backed by a memory pool[6] to provide deadlock free allocations even during swapping. The more advanced io schedulers in 2.6 usually back each request by its own private request structure, further increasing the memory pressure of each request. Dynamic request allocation lifts some of this pressure as well by pushing that allocation inside two hooks in the IO scheduler API—`set_req_fn` and `put_req_fn`. The latter handles the later freeing of that data structure.

2.6 Plugging

For the longest time, the Linux block layer has used a technique dubbed *plugging* to increase IO throughput. In its simplicity, plugging works sort of like the plug in your tub drain—when IO is queued on an initially empty queue, the queue is plugged. Only when someone asks for the completion of some of the queued IO is the plug yanked out, and io is allowed to drain from the queue. So instead of submitting the first immediately to the driver, the block layer allows a small buildup of requests. There's nothing wrong with the principle of plugging, and it has been shown to work well for a number of workloads. However, the block layer maintains a global list of plugged queues inside the `tq_disk` task queue. There are three main problems with this approach:

1. It's impossible to go backwards from the file system and find the specific queue to unplug.

2. Unplugging one queue through `tq_disk` unplugs all plugged queues.

3. The act of plugging and unplugging touches a global lock.

All of these adversely impact performance. These problems weren't really solved until late in 2.6, when Intel reported a huge scalability problem related to unplugging [Chen] on a 32 processor system. 93% of system time was spent due to contention on `blk_plug_lock`, which is the 2.6 direct equivalent of the 2.4 `tq_disk` embedded lock. The proposed solution was to move the plug lists to a per-CMU structure. While this would solve the contention problems, it still leaves the other 2 items on the above list unsolved.

So work was started to find a solution that would fix all problems at once, and just generally Feel Right. 2.6 contains a link between the block layer and write out paths which is embedded inside the queue, a `struct backing_dev_info`. This structure holds information on read-ahead and queue congestion state. It's also possible to go from a `struct page` to the backing device, which may or may not be a block device. So it would seem an obvious idea to move to a backing device unplugging scheme instead, getting rid of the global `blk_run_queues()` unplugging. That solution would fix all three issues at once—there would be no global way to unplug all devices, only target specific unplugs, and the backing device gives us a mapping from page to queue. The code was rewritten to do just that, and provide unplug functionality going from a specific `struct block_device`, page, or backing device. Code and interface was much superior to the existing code base,

[5] In reality, to prevent writes for consuming all requests.

[6] `mempool_t` interface from Ingo Molnar.

and results were truly amazing. Jeremy Higdon tested on an 8-way IA64 box [Higdon] and got 75–80 thousand IOPS on the stock kernel at 100% CPU utilization, 110 thousand IOPS with the per-CPU Intel patch also at full CPU utilization, and finally 200 thousand IOPS at merely 65% CPU utilization with the backing device unplugging. So not only did the new code provide a huge speed increase on this machine, it also went from being CPU to IO bound.

2.6 also contains some additional logic to unplug a given queue once it reaches the point where waiting longer doesn't make much sense. So where 2.4 will always wait for an explicit unplug, 2.6 can trigger an unplug when one of two conditions are met:

1. The number of queued requests reach a certain limit, `q->unplug_thresh`. This is device tweak able and defaults to 4.
2. When the queue has been idle for `q->unplug_delay`. Also device tweak able, and defaults to 3 milliseconds.

The idea is that once a certain number of requests have accumulated in the queue, it doesn't make much sense to continue waiting for more—there is already an adequate number available to keep the disk happy. The time limit is really a last resort, and should rarely trigger in real life. Observations on various work loads have verified this. More than a handful or two timer unplugs per minute usually indicates a kernel bug.

2.7 SCSI command transport

An annoying aspect of CD writing applications in 2.4 has been the need to use ide-scsi, necessitating the inclusion of the entire SCSI stack for only that application. With the clear majority of the market being ATAPI hardware, this becomes even more silly. ide-scsi isn't without its own class of problems either—it lacks the ability to use DMA on certain writing types. CDDA audio ripping is another application that thrives with ide-scsi, since the native uniform cdrom layer interface is less than optimal (put mildly). It doesn't have DMA capabilities at all.

2.7.1 Enhancing struct request

The problem with 2.4 was the lack of ability to generically send SCSI "like" commands to devices that understand them. Historically, only file system read/write requests could be submitted to a driver. Some drivers made up faked requests for other purposes themselves and put then on the queue for their own consumption, but no defined way of doing this existed. 2.6 adds a new request type, marked by the `REQ_BLOCK_PC` bit. Such a request can be either backed by a `bio` like a file system request, or simply has a data and length field set. For both types, a SCSI command data block is filled inside the request. With this infrastructure in place and appropriate update to drivers to understand these requests, it's a cinch to support a much better direct-to-device interface for burning.

Most applications use the SCSI sg API for talking to devices. Some of them talk directly to the `/dev/sg*` special files, while (most) others use the `SG_IO` ioctl interface. The former requires a yet unfinished driver to transform them into block layer requests, but the latter can be readily intercepted in the kernel and routed directly to the device instead of through the SCSI layer. Helper functions were added to make burning and ripping even faster, providing DMA for all applications and without copying data between kernel and user space at all. So the zero-copy DMA burning was possible, and this even without changing most applications.

3 Linux-2.7

The 2.5 development cycle saw the most massively changed block layer in the history of Linux. Before 2.5 was opened, Linus had clearly expressed that one of the most important things that needed doing, was the block layer update. And indeed, the very first thing merged was the complete bio patch into 2.5.1-pre2. At that time, no more than a handful drivers compiled (let alone worked). The 2.7 changes will be nowhere as severe or drastic. A few of the possible directions will follow in the next few sections.

3.1 IO Priorities

Prioritized IO is a very interesting area that is sure to generate lots of discussion and development. It's one of the missing pieces of the complete resource management puzzle that several groups of people would very much like to solve. People running systems with many users, or machines hosting virtual hosts (or completed virtualized environments) are dying to be able to provide some QOS guarantees. Some work was already done in this area, so far nothing complete has materialized. The CKRM [CKRM] project spear headed by IBM is an attempt to define global resource management, including io. They applied a little work to the **CFQ** IO scheduler to provide equal bandwidth between resource management classes, but at no specific priorities. Currently I have a **CFQ** patch that is 99% complete that provides full priority support, using the IO contexts introduced by **AS** to manage fair sharing over the full time span that a process exists[7]. This works well enough, but only works for that specific IO scheduler. A nicer solution would be to create a scheme that works independently of the io scheduler used. That would require a rethinking of the IO scheduler API.

[7]CFQ currently tears down class structures as soon as it is empty, it doesn't persist over process life time.

3.2 IO Scheduler switching

Currently Linux provides no less than 4 IO schedulers—the 3 mentioned, plus a forth dubbed **noop**. The latter is a simple IO scheduler that does no request reordering, no latency management, and always merges whenever it can. Its area of application is mainly highly intelligent hardware with huge queue depths, where regular request reordering doesn't make sense. Selecting a specific IO scheduler can either be done by modifying the source of a driver and putting the appropriate calls in there at queue init time, or globally for any queue by passing the `elevator=xxx` boot parameter. This makes it impossible, or at least very impractical, to benchmark different IO schedulers without many reboots or recompiles. Some way to switch IO schedulers per queue and on the fly is desperately needed. Freezing a queue and letting IO drain from it until it's empty (pinning new IO along the way), and then shutting down the old io scheduler and moving to the new scheduler would not be so hard to do. The queues expose various sysfs variables already, so the logical approach would simply be to:

```
# echo deadline > \
    /sys/block/hda/queue/io_scheduler
```

A simple but effective interface. At least two patches doing something like this were already proposed, but nothing was merged at that time.

4 Final comments

The block layer code in 2.6 has come a long way from the rotted 2.4 code. New features bring it more up-to-date with modern hardware, and completely rewritten from scratch core provides much better scalability, performance, and memory usage benefiting any machine from small to really huge. Going back a few years, I heard constant complaints about the block layer and how much it sucked and how outdated it was. These days I rarely hear anything about the current state of affairs, which usually means that it's doing pretty well indeed. 2.7 work will mainly focus on feature additions and driver layer abstractions (our concept of IDE layer, SCSI layer etc will be severely shook up). Nothing that will wreak havoc and turn everything inside out like 2.5 did. Most of the 2.7 work mentioned above is pretty light, and could easily be back ported to 2.6 once it has been completed and tested. Which is also a good sign that nothing really radical or risky is missing. So things are settling down, a sign of stability.

References

[Iyer] Sitaram Iyer and Peter Druschel, *Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O*, 18th ACM Symposium on Operating Systems Principles, `http://www.cs.rice.edu/~ssiyer/r/antsched/antsched.ps.gz`, 2001

[McKenney] Paul E. McKenney, *Stochastic Fairness Queuing*, INFOCOM `http://rdrop.com/users/paulmck/paper/sfq.2002.06.04.pdf`, 1990

[Chen] Kenneth W. Chen, *per-cpu blk_plug_list*, Linux kernel mailing list `http://www.ussg.iu.edu/hypermail/linux/kernel/0403.0/0179.html`, 2004

[Higdon] Jeremy Higdon, *Re: [PATCH] per-backing dev unplugging #2*, Linux kernel mailing list `http://marc.theaimsgroup.com/?l=linux-kernel&m=107941470424309&w=2`, 2004

[CKRM] IBM, *Class-based Kernel Resource Management (CKRM)*, `http://ckrm.sf.net`, 2004

[Bhattacharya] Suparna Bhattacharya, *Notes on the Generic Block Layer Rewrite in Linux 2.5*, General discussion, `Documentation/block/biodoc.txt`, 2002

Linux AIO Performance and Robustness for Enterprise Workloads

Suparna Bhattacharya, IBM `(suparna@in.ibm.com)`
John Tran, IBM `(jbtran@ca.ibm.com)`
Mike Sullivan, IBM `(mksully@us.ibm.com)`
Chris Mason, SUSE `(mason@suse.com)`

1 Abstract

In this paper we address some of the issues identified during the development and stabilization of Asynchronous I/O (AIO) on Linux 2.6.

We start by describing improvements made to optimize the throughput of streaming buffered filesystem AIO for microbenchmark runs. Next, we discuss certain tricky issues in ensuring data integrity between AIO Direct I/O (DIO) and buffered I/O, and take a deeper look at synchronized I/O guarantees, concurrent I/O, write-ordering issues and the improvements resulting from radix-tree based write-back changes in the Linux VFS.

We then investigate the results of using Linux 2.6 filesystem AIO on the performance metrics for certain enterprise database workloads which are expected to benefit from AIO, and mention a few tips on optimizing AIO for such workloads. Finally, we briefly discuss the issues around workloads that need to combine asynchronous disk I/O and network I/O.

2 Introduction

AIO enables a single application thread to overlap processing with I/O operations for better utilization of CPU and devices. AIO can improve the performance of certain kinds of I/O intensive applications like databases, webservers and streaming-content servers. The use of AIO also tends to help such applications adapt and scale more smoothly to varying loads.

2.1 Overview of kernel AIO in Linux 2.6

The Linux 2.6 kernel implements in-kernel support for AIO. A low-level native AIO system call interface is provided that can be invoked directly by applications or used by library implementations to build POSIX/SUS semantics. All discussion hereafter in this paper pertains to the native kernel AIO interfaces.

Applications can submit one or more I/O requests asynchronously using the `io_submit()` system call, and obtain completion notification using the `io_getevents()` system call. Each I/O request specifies the operation (typically read/write), the file descriptor and the parameters for the operation (e.g., file offset, buffer). I/O requests are associated with the completion queue (ioctx) they were submitted against. The results of I/O are reported as completion events on this queue, and reaped using `io_getevents()`.

The design of AIO for the Linux 2.6 kernel has been discussed in [1], including the motivation behind certain architectural choices, for example:

- Sharing a common code path for AIO and regular I/O
- A retry-based model for AIO continuations across blocking points in the case of buffered filesystem AIO (currently implemented as a set of patches to the Linux 2.6 kernel) where worker threads take on the caller's address space for executing retries involving access to user-space buffers.

2.2 Background on retry-based AIO

The retry-based model allows an AIO request to be executed as a series of non-blocking iterations. Each iteration retries the remaining part of the request from where the last iteration left off, re-issuing the corresponding AIO filesystem operation with modified arguments representing the remaining I/O. The retries are "kicked" via a special AIO waitqueue callback routine, `aio_wake_function()`, which replaces the default waitqueue entry used for blocking waits.

The high-level retry infrastructure is responsible for running the iterations in the address space context of the caller, and ensures that only one retry instance is active at a given time. This relieves the fops themselves from having to deal with potential races of that sort.

2.3 Overview of the rest of the paper

In subsequent sections of this paper, we describe our experiences in addressing several issues identified during the optimization and stabilization efforts related to the kernel AIO implementation for Linux 2.6, mainly in the area of disk- or filesystem-based AIO.

We observe, for example, how I/O patterns generated by the common VFS code paths used by regular and retry-based AIO could be non-optimal for streaming AIO requests, and we describe the modifications that address this finding. A different set of problems that has seen some development activity are the races, exposures and potential data-integrity concerns between direct and buffered I/O, which become especially tricky in the presence of AIO. Some of these issues motivated Andrew Morton's modified page-writeback design for the VFS using tagged radix-tree lookups, and we discuss the implications for the AIO `O_SYNC` write implementation. In general, disk-based filesystem AIO requirements for database workloads have been a guiding consideration in resolving some of the trade-offs encountered, and we present some initial performance results for such workloads. Lastly, we touch upon potential approaches to allow processing of disk-based AIO and communications I/O within a single event loop.

3 Streaming AIO reads

3.1 Basic retry pattern for single AIO read

The retry-based design for buffered filesystem AIO read works by converting each blocking wait for read completion on a page into a *retry exit*. The design queues an asynchronous notification callback and returns the number of bytes for which the read has completed so far without blocking. Then, when the page becomes up-to-date, the callback kicks off a retry continuation in task context. This retry continuation invokes the same filesystem read operation again using the caller's address space, but this time with arguments modified to reflect the remaining part of the read request.

For example, given a 16KB read request starting at offset 0, where the first 4KB is already in cache, one might see the following sequence of retries (in the absence of readahead):

```
first time:
   fop->aio_read(fd, 0, 16384) = 4096
and when read completes for the second page:
   fop->aio_read(fd, 4096, 12288) = 4096
and when read completes for the third page:
   fop->aio_read(fd, 8192, 8192) = 4096
and when read completes for the fourth page:
   fop->aio_read(fd, 12288, 4096) = 4096
```

3.2 Impact of readahead on single AIO read

Usually, however, the readahead logic attempts to batch read requests in advance. Hence, more I/O would be seen to have completed at each retry. The logic attempts to predict the optimal readahead window based on state it maintains about the sequentiality of past read requests on the same file descriptor. Thus, given a maximum readahead window size of 128KB, the sequence of retries would appear to be more like the following example, which results in significantly improved throughput:

```
first time:
   fop->aio_read(fd, 0, 16384) = 4096,
     after issuing readahead
     for 128KB/2 = 64KB
and when read completes for the above I/O:
   fop->aio_read(fd, 4096, 12288) = 12288
```

Notice that care is taken to ensure that readaheads are not repeated during retries.

3.3 Impact of readahead on streaming AIO reads

In the case of streaming AIO reads, a sequence of AIO read requests is issued on the same file descriptor, where subsequent reads are submitted without waiting for previous requests to complete (contrast this with a sequence of synchronous reads).

Interestingly, we encountered a significant throughput degradation as a result of the interplay of readahead and streaming AIO reads. To see why, consider the retry sequence for streaming random AIO read requests of 16KB, where `o1, o2, o3, ...` refer to the random offsets where these reads are issued:

```
first time:
  fop->aio_read(fd, o1, 16384) = -EIOCBRETRY,
    after issuing readahead for 64KB
    as the readahead logic sees the first page
    of the read
  fop->aio_read(fd, o2, 16384) = -EIOCBRETRY,
    after issuing readahead for 8KB (notice
    the shrinkage of the readahead window
    because of non-sequentiality seen by the
    readahead logic)
  fop->aio_read(fd, o3, 16384) = -EIOCBRETRY,
    after maximally shrinking the readahead
    window, turning off readahead and issuing
    4KB read in the slow path
  fop->aio_read(fd, o4, 16384) = -EIOCBRETRY,
    after issuing 4KB read in the slow path
.
.
and when read completes for o1
  fop->aio_read(fd, o1, 16384) = 16384
and when read completes for o2
  fop->aio_read(fd, o2, 16384) = 8192
and when read completes for o3
  fop->aio_read(fd, o3, 16384) = 4096
and when read completes for o4
  fop->aio_read(fd, o3, 16384) = 4096
.
.
```

In steady state, this amounts to a maximally-shrunk readahead window with 4KB reads at random offsets being issued serially one at a time on a slow path, causing seek storms and driving throughputs down severely.

3.4 Upfront readahead for improved streaming AIO read throughputs

To address this issue, we made the readahead logic aware of the sequentiality of all pages in a single read request upfront—before submitting the next read request. This resulted in a more desirable outcome as follows:

```
  fop->aio_read(fd, o1, 16384) = -EIOCBRETRY,
    after issuing readahead for 64KB
    as the readahead logic sees all the 4
    pages for the read
  fop->aio_read(fd, o2, 16384) = -EIOCBRETRY,
    after issuing readahead for 20KB, as the
    readahead logic sees all 4 pages of the
    read (the readahead window shrinks to
    4+1=5 pages)
  fop->aio_read(fd, o3, 16384) = -EIOCBRETRY,
    after issuing readahead for 20KB, as the
    readahead logic sees all 4 pages of the
    read (the readahead window is maintained
    at 4+1=5 pages)
        .
        .
and when read completes for o1
  fop->aio_read(fd, o1, 16384) = 16384
and when read completes for o2
  fop->aio_read(fd, o2, 16384) = 16384
and when read completes for o3
  fop->aio_read(fd, o3, 16384) = 16384
        .
        .
```

3.5 Upfront readahead and sendfile regressions

At first sight it appears that upfront readahead is a reasonable change for all situations, since it immediately passes to the readahead logic the entire size of the request. However, it has the unintended, potential side-effect of losing pipelining benefits for really large reads, or operations like sendfile which involve post processing I/O on the contents just read. One way to address this is to clip the maximum size of upfront readahead to the maximum readahead setting for the device. To see why even that may not suffice for certain situations, let us take a look at the following sequence for a webserver that uses non-blocking sendfile to serve a large (2GB) file.

```
  sendfile(fd, 0, 2GB, fd2) = 8192,
    tells readahead about up to 128KB
    of the read
  sendfile(fd, 8192, 2GB - 8192, fd2) = 8192,
    tells readahead about 8KB - 132KB
    of the read
  sendfile(fd, 16384, 2GB - 16384, fd2) = 8192,
    tells readahead about 16KB-140KB
    of the read
       ...
```

This confuses the readahead logic about the I/O pattern which appears to be 0–128K, 8K–132K, 16K–140K instead of clear sequentiality from 0–2GB that is really appropriate.

To avoid such unanticipated issues, upfront readahead required a special case for AIO alone, limited to the maximum readahead setting for the device.

3.6 Streaming AIO read microbenchmark comparisons

We explored streaming AIO throughput improvements with the retry-based AIO implementation and optimizations discussed above, using a custom microbenchmark called aio-stress [2]. aio-stress issues a stream of AIO requests to one or more files, where one can vary several parameters including I/O unit size, total I/O size, depth of iocbs submitted at a time, number of concurrent threads, and type and pattern of I/O operations, and reports the overall throughput attained.

The hardware included a 4-way 700MHz Pentium® III machine with 512MB of RAM and a 1MB L2 cache. The disk subsystem used for the I/O tests consisted of an Adaptec AIC7896/97 Ultra2 SCSI controller connected to a disk enclosure with six 9GB disks, one of which was configured as an ext3 filesystem with a block size of 4KB for testing.

The runs compared aio-stress throughputs for streaming random buffered I/O reads (i.e., without `O_DIRECT`), with and without the previously described changes. All the runs were for the case where the file was not already cached in memory. The above graph summarizes how the results varied across individual request sizes of 4KB to 64KB, where I/O was targeted to a single file of size 1GB, the depth of iocbs outstanding at a time being 64KB. A third run was performed to find out how the results compared with equivalent runs using AIO-DIO.

With the changes applied, the results showed an approximate 2x improvement across all block sizes, bringing throughputs to levels that match the corresponding results using AIO-DIO.

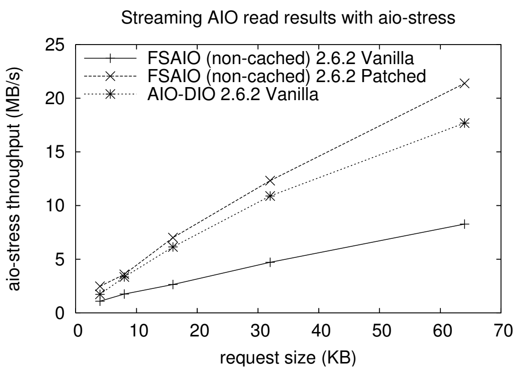


Figure 1: Comparisons of streaming random AIO read throughputs

4 AIO DIO vs cached I/O integrity issues

4.1 DIO vs buffered races

Stephen Tweedie discovered several races between DIO and buffered I/O to the same file [3]. These races could lead to potential stale-data exposures and even data-integrity issues. Most instances were related to situations when in-core meta-data updates were visible before actual instantiation or resetting of corresponding data blocks on disk. Problems could also arise when meta-data updates were not visible to other code paths that could simultaneously update meta-data as well. The races mainly affected sparse files due to the lack of atomicity between the file flush in the DIO paths and actual data block accesses.

The solution that Stephen Tweedie came up with, and which Badari Pulavarty reported to Linux 2.6, involved protecting block lookups and meta-data updates with the inode semaphore (`i_sem`) in DIO paths for both read and write, atomically with the file flush. Overwriting of sparse blocks in the DIO write path was modified to fall back to buffered writes. Finally, an additional semaphore (`i_alloc_sem`) was introduced to lock out deallocation of blocks by a truncate while DIO was in progress. The semaphore was implemented held in shared mode by DIO and in exclusive mode by truncate.

Note that handling the new locking rules (i.e., lock ordering of `i_sem` first and then `i_alloc_sem`) while allowing for filesystem-specific implementations of the DIO and file-write interfaces had to be handled with some care.

4.2 AIO-DIO specific races

The inclusion of AIO in Linux 2.6 added some tricky scenarios to the above-described problems because of the potential races inherent in returning without waiting for I/O completion. The interplay of AIO-DIO writes and truncate was a particular worry as it could lead to corruption of file data; for example, blocks could get deallocated and reallocated to a new file while an AIO-DIO write to the file was still in progress. To avoid this, AIO-DIO had to return with `i_alloc_sem` held, and only release it as part of I/O completion post-processing. Notice that this also had implications for AIO cancellation.

File size updates for AIO-DIO file extends could expose unwritten blocks if they happened before I/O completed asynchronously. The case involving fallback to buffered I/O was particularly non-trivial if a single request spanned allocated and sparse regions of a file. Specifically, part of the I/O could have been initiated via DIO then continued asynchronously, while the fallback to buffered I/O occurred and signaled I/O completion to the application. The application may thus have reused its I/O buffer, overwriting it with other data and potentially causing file data corruption if writeout to disk had still been pending.

It might appear that some of these problems could be avoided if I/O schedulers guaranteed the ordering of I/O requests issued to the same disk block. However, this isn't a simple proposition in the current architecture, especially in generalizing the design to all possible cases, including network block devices. The use of I/O barriers would be necessary and the costs may not be justified for these special-case situations.

Instead, a pragmatic approach was taken in order to address this based on the assumptions that true asynchronous behaviour was really meaningful in practice, mainly when performing I/O to already-allocated file blocks. For example, databases typically preallocate files at the time of creation, so that AIO writes during normal operation and in performance-critical paths do not extend the file or encounter sparse regions. Thus, for the sake of correctness, synchronous behaviour may be tolerable for AIO writes involving sparse regions or file extends. This compromise simplified the handling of the scenarios described earlier. AIO-DIO file extends now wait for I/O to complete and update the file size. AIO-DIO writes spanning allocated and sparse regions now wait for previously- issued DIO for that request to complete before falling back to buffered I/O.

5 Concurrent I/O with synchronized write guarantees

An application opts for synchronized writes (by using the `O_SYNC` option on file open) when the I/O must be committed to disk before the write request completes. In the case of DIO, writes directly go to disk anyway. For buffered I/O, data is first copied into the page cache and later written out to disk; if synchronized I/O is specified then the request returns only after the writeout is complete.

An application might also choose to synchronize previously-issued writes to disk by invoking fsync(), which writes back data from the page cache to disk and waits for writeout to complete before returning.

5.1 Concurrent DIO writes

DIO writes formerly held the inode semaphore in exclusive mode until write completion. This helped ensure atomicity of DIO writes and protected against potential file data corruption races with truncate. However, it also meant that multiple threads or processes submitting parallel DIOs to different parts of the same file effectively became serialized synchronously. If the same behaviour were extended to AIO (i.e., having the `i_sem` held through I/O completion for AIO-DIO writes), it would significantly degrade throughput of streaming AIO writes as subsequent write submissions would block until completion of the previous request.

With the fixes described in the previous section, such synchronous serialization is avoidable without loss of correctness, as the inode semaphore needs to be held only when looking up the blocks to write, and not while actual I/O is in progress on the data blocks. This could allow concurrent DIO writes on different parts of a file to proceed simultaneously, and efficient throughputs for streaming AIO-DIO writes.

5.2 Concurrent `O_SYNC` buffered writes

In the original writeback design in the Linux VFS, per-address space lists were maintained for dirty pages and pages under writeback for a given file. Synchronized write was implemented by traversing these lists to issue writeouts for the dirty pages and waiting for writeback to complete on the pages on the writeback list. The inode semaphore had to be held all through to avoid possibilities of livelocking on these lists as further writes streamed into the same file. While this helped maintain atomicity of writes, it meant that parallel `O_SYNC` writes to different parts of the file were effectively serialized synchronously. Further, dependence on `i_sem`-protected state in the address space lists across I/O waits made it difficult to retry-enable this code path for AIO support.

In order to allow concurrent `O_SYNC` writes to be active on a file, the range of pages to be written back and waited on could instead be obtained directly through a radix-tree lookup for the range of offsets in the file that was being written out by the request [4]. This would avoid traversal of the page lists and hence the need to hold `i_sem` across the I/O waits. Such an approach would also make it possible to complete `O_SYNC` writes as a sequence of non-blocking retry iterations across the range of bytes in a given request.

5.3 Data-integrity guarantees

Background writeout threads cannot block on the inode semaphore like O_SYNC/fsync writers. Hence, with the per-address space lists writeback model, some juggling involving movement across multiple lists was required to avoid livelocks. The implementation had to make sure that pages which by chance got picked up for processing by background writeouts didn't slip from consideration when waiting for writeback to complete for a synchronized write request. The latter would be particularly relevant for ensuring synchronized-write guarantees that impacted data integrity for applications. However, as Daniel McNeil's analysis would indicate [5], getting this right required the writeback code to write and wait upon I/O and dirty pages which were initiated by other processes, and that turned out to be fairly tricky.

One solution that was explored was per-address space serialization of writeback to ensure exclusivity to synchronous writers and shared mode for background writers. It involved navigating issues with busy-waits in background writers and the code was beginning to get complicated and potentially fragile.

This was one of the problems that finally prompted Andrew Morton to change the entire VFS writeback code to use radix-tree walks instead of the per-address space pagelists. The main advantage was that avoiding the need for movement across lists during state changes (e.g., when re-dirtying a page if its buffers were locked for I/O by another process) reduced the chances of pages getting missed from consideration without the added serialization of entire writebacks.

6 Tagged radix-tree based writeback

For the radix-tree walk writeback design to perform as well as the address space lists-based approach, an efficient way to get to the pages of interest in the radix trees is required. This is especially so when there are many pages in the pagecache but only a few are dirty or under writeback. Andrew Morton solved this problem by implementing tagged radix-tree lookup support to enable lookup of dirty or writeback pages in O(log64(n)) time [6].

This was achieved by adding tag bits for each slot to each radix-tree node. If a node is tagged, then the corresponding slots on all the nodes above it in the tree are tagged. Thus, to search for a particular tag, one would keep going down sub-trees under slots which have the tag bit set until the tagged leaf nodes are accessed. A tagged gang lookup function is used for in-order searches for dirty or writeback pages within a specified range. These lookups are used to replace the per-address-space page lists altogether.

To synchronize writes to disk, a tagged radix-tree gang lookup of dirty pages in the byte-range corresponding to the write request is performed and the resulting pages are written out. Next, pages under writeback in the byte-range are obtained through a tagged radix-tree gang lookup of writeback pages, and we wait for writeback to complete on these pages (without having to hold the inode semaphore across the waits). Observe how this logic lends itself to be broken up into a series of non-blocking retry iterations proceeding in-order through the range.

The same logic can also be used for a whole file sync, by specifying a byte-range that spans the entire file.

Background writers also use tagged radix-tree gang lookups of dirty pages. Instead of always scanning a file from its first dirty page, the index where the last batch of writeout terminated is tracked so the next batch of writeouts can be started after that point.

7 Streaming AIO writes

The tagged radix-tree walk writeback approach greatly simplifies the design of AIO support for synchronized writes, as mentioned in the previous section,

7.1 Basic retry pattern for synchronized AIO writes

The retry-based design for buffered AIO `O_SYNC` writes works by converting each blocking wait for writeback completion of a page into a *retry exit*. The conversion point queues an asynchronous notification callback and returns to the caller of the filesystem's AIO write operation the number of bytes for which writeback has completed so far without blocking. Then, when writeback completes for that page, the callback kicks off a retry continuation in task context which invokes the same AIO write operation again using the caller's address space, but this time with arguments modified to reflect the remaining part of the write request.

As writeouts for the range would have already been issued the first time before the loop to wait for writeback completion, the implementation takes care not to re-dirty pages or re-issue writeouts during subsequent retries of AIO write. Instead, when the code detects that it is being called in a retry context, it simply falls through directly to the step involving wait-on-writeback for the remaining range as specified by the modified arguments.

7.2 Filtered waitqueues to avoid retry storms with hashed wait queues

Code that is in a retry-exit path (i.e., the return path following a blocking point where a retry is queued) should in general take care not to call routines that could wakeup the newly-queued retry.

One thing that we had to watch for was calls to `unlock_page()` in the retry-exit path. This could cause a redundant wakeup if an async wait-on-page writeback was just queued for that page. The redundant wakeup would arise if the kernel used the same waitqueue on unlock as well as writeback completion for a page, with the expectation that the waiter would check for the condition it was waiting for and go back to sleep if it hadn't occurred. In the AIO case, however, a wakeup of the newly-queued callback in the same code path could potentially trigger a retry storm, as retries kept triggering themselves over and over again for the wrong condition.

The interplay of `unlock_page()` and `wait_on_page_writeback()` with hashed waitqueues can get quite tricky for retries. For example, consider what happens when the following sequence in retryable code is executed at the same time for 2 pages, *px* and *py*, which happen to hash to the same waitqueue (Table 1).

```
lock_page(p)
check condition and process
unlock_page(p)
if (wait_on_page_writeback_wq(p)
  == -EIOCBQUEUED)
    return bytes_done
```

The above code could keep cycling between spurious retries on *px* and *py* until I/O is done, wasting precious CPU time!

If we can ensure specificity of the wakeup with hashed waitqueues then this problem can be avoided. William Lee Irwin's implementation of filtered wakeup support in the recent Linux 2.6 kernels [7] achieves just that. The wakeup routine specifies a key to match before invoking the wakeup function for an entry in the waitqueue, thereby limiting wakeups to those entries which have a matching key. For page waitqueues, the key is computed as a function of the page and the condition (unlock or writeback completion) for the wakeup.

7.3 Streaming AIO write microbenchmark comparisons

The following graph compares aio-stress throughputs for streaming random buffered I/O `O_SYNC` writes, with and without the previously-described changes. The comparison was performed on the same setup used for the streaming AIO read results discussed earlier. The graph summarizes how the results varied across individual request sizes of 4KB to 64KB, where I/O was targeted to a single file of size 1GB and the depth of iocbs outstanding at a time was 64KB. A third run was performed to determine how the results compared with equivalent runs using AIO-DIO.

With the changes applied, the results showed an approximate 2x improvement across all

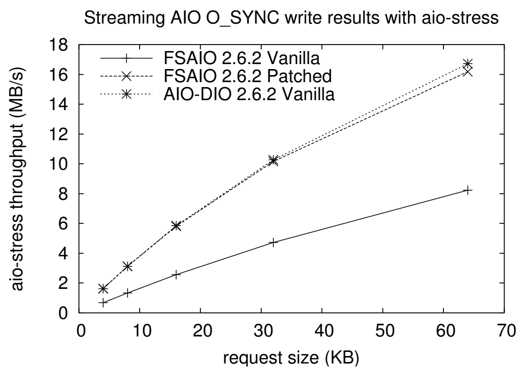


Figure 2: Comparisons of streaming random AIO write throughputs.

block sizes, bringing throughputs to levels that match the corresponding results using AIO-DIO.

8 AIO performance analysis for database workloads

Large database systems leveraging AIO can show marked performance improvements compared to those systems that use synchronous I/O alone. We use IBM® DB2® Universal Database™ V8 running an online transaction processing (OLTP) workload to illustrate the performance improvement of AIO on raw devices and on filesystems.

8.1 DB2 page cleaners

A DB2 page cleaner is a process responsible for flushing dirty buffer pool pages to disk. It simulates AIO by executing asynchronously with respect to the agent processes. The number of page cleaners and their behavior can be tuned according to the demands of the system. The agents, freed from cleaning pages themselves, can dedicate their resources (e.g., processor cycles) towards processing transactions, thereby improving throughput.

```
 CPU1                                  CPU2
 lock_page(px)
 ...
 unlock_page(px)
                                       lock_page(py)
 wait_on_page_writeback_wq(px)         ...
                                       unlock_page(py) -> wakes up p1
                                                          triggering
               <-----------------------------------       a retry
lock_page(px)                          wait_on_page_writeback_wq(py)
...
unlock_page(py)  ---- wakes up py --- causes retry ---->
```

Table 1: Retry storm livelock with redundant wakeups on hashed wait queues

8.2 AIO performance analysis for raw devices

Two experiments were conducted to measure the performance benefits of AIO on raw devices for an update-intensive OLTP database workload. The workload used was derived from a TPC[8] benchmark, but is in no way comparable to any TPC results. For the first experiment, the database was configured with one page cleaner using the native Linux AIO interface. For the second experiment, the database was configured with 55 page cleaners all using the synchronous I/O interface. These experiments showed that a database, properly configured in terms of the number of page cleaners with AIO, can out-perform a properly configured database using synchronous I/O page cleaning.

For both experiments, the system configuration consisted of DB2 V8 running on a 2-way AMD Opteron system with Linux 2.6.1 installed. The disk subsystem consisted of two FAStT 700 storage servers, each with eight disk enclosures. The disks were configured as RAID-0 arrays with a stripe size of 256KB.

Table 2 shows the relative database performance with and without AIO. Higher numbers are better. The results show that the database performed 9% better when configured with one page cleaner using AIO, than when it was configured with 55 page cleaners using synchronous I/O.

Configuration	Relative Throughput
1 page cleaner with AIO	133
55 page cleaners without AIO	122

Table 2: Database performance with and without AIO.

Analyzing the I/O write patterns (see Table 3), we see that one page cleaner using AIO was sufficient to keep the buffer pools clean under a very heavy load, but that 55 page cleaners using synchronous I/O were not, as indicated by the 30% agent writes. This data suggests that more page cleaners should have been configured to improve the performance of the case with synchronous I/O. However, additional page cleaners consumed more memory, requiring a reduction in bufferpool size and thereby decreasing throughput. For the test configuration, 55 cleaners was the optimal number before memory constraints arose.

8.3 AIO performance analysis for filesystems

This section examines the performance improvements of AIO when used in conjunction with filesystems. This experiment was per-

Configuration	Page cleaner writes (%)	Agent writes (%)
1 page cleaner with AIO	100	0
55 page cleaners without AIO	70	30

Table 3: DB2 write patterns for raw device configurations.

formed using the same OLTP benchmark as in the previous section.

The test system consisted of two 1GHz AMD Opteron processors, 4GB of RAM and two QLogic 2310 FC controllers. Attached to the server was a single FAStT900 storage server and two disk enclosures with a total of 28 15K RPM 18GB drives. The Linux kernel used for the examination was 2.6.0+mm1, which includes the AIO filesystem support patches [9] discussed in this paper.

The database tables were spread across multiple ext2 filesystem partitions. Database logs were stored on a single raw partition.

Three separate tests were performed, utilizing different I/O methods for the database page cleaners.

Test 1. Synchronous (Buffered) I/O.

Test 2. Asynchronous (Buffered) I/O.

Test 3. Direct I/O.

The results are shown in Table 4 as relative commercial processing scores using synchronous I/O as the baseline (i.e., higher is better).

Looking at the efficiency of the page cleaners (see Table 5), we see that the use of AIO is more successful in keeping the buffer pools clean. In the synchronous I/O and DIO cases, the agents needed to spend more time cleaning buffer pool pages, resulting in less time processing transactions.

Configuration	Commercial Processing Scores
Synchronous I/O	100
AIO (Buffered)	113.7
DIO	111.9

Table 4: Database performance on filesystems with and without AIO.

Configuration	Page cleaner writes (%)	Agent writes (%)
Synchronous I/O	37	63
AIO (buffered)	100	0
DIO	49	51

Table 5: DB2 write patterns for filesystem configurations.

8.4 Optimizing AIO for database workloads

Databases typically use AIO for streaming batches of random, synchronized write requests to disk (where the writes are directed to preallocated disk blocks). This has been found to improve the performance of OLTP workloads, as it helps bring down the number of dedicated threads or processes needed for flushing updated pages, and results in reduced memory footprint and better CPU utilization and scaling.

The size of individual write requests is determined by the page size used by the database. For example, a DB2 UDB installation might use a database page size of 8KB.

As observed in previous sections, the use of AIO helps reduce the number of database page cleaner processes required to keep the bufferpool clean. To keep the disk queues maximally utilized and limit contention, it may be preferable to have requests to a given disk streamed out from a single page cleaner. Typically a set of of disks could be serviced by each page cleaner if and when multiple page cleaners need to be used.

Databases might also use AIO for reads, for example, for prefetching data to service queries. This usually helps improve the performance of decision support workloads. The I/O pattern generated in these cases is that of streaming batches of large AIO reads, with sizes typically determined by the file allocation extent size used by the database (e.g., a DB2 installation might use a database extent size of 256KB). For installations using buffered AIO reads, tuning the readahead setting for the corresponding devices to be more than the extent size would help improve performance of streaming AIO reads (recall the discussion in Section 3.5).

9 Addressing AIO workloads involving both disk and communications I/O

Certain applications need to handle both disk-based AIO and communications I/O. For communications I/O, the epoll interface—which provides support for efficient scalable event polling in Linux 2.6—could be used as appropriate, possibly in conjunction with `O_NONBLOCK` socket I/O. Disk-based AIO on the other hand, uses the native AIO API `io_getevents` for completion notification. This makes it difficult to combine both types of I/O processing within a single event loop, even when such a model is a natural way to program the application, as in implementations of the application on other operating systems.

How do we address this issue? One option is to extend epoll to enable it to poll for notification of AIO completion events, so that AIO completion status can then be reaped in a non-blocking manner. This involves mixing both epoll and AIO API programming models, which is not ideal.

9.1 AIO poll interface

Another alternative is to add support for polling an event on a given file descriptor through the AIO interfaces. This function, referred to as AIO poll, can be issued through `io_submit()` just like other AIO operations, and specifies the file descriptor and the eventset to wait for. When the event occurs, notification is reported through `io_getevents()`.

The retry-based design of AIO poll works by converting the blocking wait for the event into a *retry exit*.

The generic synchronous polling code fits nicely into the AIO retry design, so most of the original polling code can be used unchanged. The private data area of the iocb can be used to hold polling-specific data structures, and a few special cases can be added to the generic polling entry points. This allows the AIO poll case to proceed without additional memory allocations.

9.2 AIO operations for communications I/O

A third option is to add support for AIO operations for communications I/O. For example, AIO support for pipes has been implemented by converting the blocking wait for I/O on pipes to a *retry exit*. The generic pipe code was also structured such that conversion to AIO retries was quite simple, the only significant change was using the current `io_wait` context instead of a locally defined waitqueue, and returning early if no data was available.

However, AIO pipe testing did show significantly more context switches then the 2.4 AIO pipe implementation, and this was coupled with much lower performance. The AIO core functions were relying on workqueues to do most of the retries, and this resulted in constant switching between the workqueue threads and user processes.

The solution was to change the AIO core to do retries in `io_submit()` and in `io_getevents()`. This allowed the process to do some of its own work while it is scheduled in. Also, retries were switched to a delayed workqueue, so that bursts of retries would trigger fewer context switches.

While delayed wakeups helped with pipe workloads, it also caused I/O stalls in filesystem AIO workloads. This was because a delayed wakeup was being used even when a user process was waiting in `io_getevents()`. When user processes are actively waiting for events, it proved best to trigger the worker thread immediately.

General AIO support for network operations has been considered but not implemented so far because of lack of supporting study that predicts a significant benefit over what epoll and non-blocking I/O can provide, except for the scope for enabling potential zero-copy implementations. This is a potential area for future research.

10 Conclusions

Our experience over the last year with AIO development, stabilization and performance improvements brought us to design and implementation issues that went far beyond the initial concern of converting key I/O blocking points to be asynchronous.

AIO uncovered scenarios and I/O patterns that were unlikely or less significant with synchronous I/O alone. For example, the issues we discussed around streaming AIO performance with readahead and concurrent synchronized writes, as well as DIO vs buffered I/O complexities in the presence of AIO. In retrospect, this was the hardest part of supporting AIO—modifiying code that was originally designed only for synchronous I/O.

Interestingly, this also meant that AIO appeared to magnify some problems early. For example, issues with hashed waitqueues that led to the filtered wakeup patches, and readahead window collapses with large random reads which precipitated improvements to the readahead code from Ramachandra Pai. Ultimately, many of the core improvements that helped AIO have had positive benefits in allowing improved concurrency for some of the synchronous I/O paths.

In terms of benchmarking and optimizing Linux AIO performance, there is room for more exhaustive work. Requirements for AIO fsync support are currently under consideration. There is also a need for more widely used AIO applications, especially those that take advantaged of AIO support for buffered I/O or bring out additional requirements like network I/O beyond epoll or AIO poll. Finally, investigations into API changes to help enable more efficient POSIX AIO implementations based on kernel AIO support may be a worthwhile endeavor.

11 Acknowledgements

We would like to thank the many people on the `linux-aio@kvack.org` and `linux-kernel@vger.kernel.org` mailing lists who provided us with valuable comments and suggestions during our development efforts.

We would especially like to acknowledge the important contributions of Andrew Morton, Daniel McNeil, Badari Pulavarty, Stephen Tweedie, and William Lee Irwin towards several pieces of work discussed in this paper.

This paper and the work it describes wouldn't have been possible without the efforts of Janet Morgan in many different ways, starting from review, test and debugging feedback to joining the midnight oil camp to help with modifications and improvements to the text during the final stages of the paper.

We also thank Brian Twitchell, Steve Pratt, Gerrit Huizenga, Wayne Young, and John Lumby from IBM for their help and discussions along the way.

This work was a part of the Linux Scalability Effort (LSE) on SourceForge, and further information about Linux 2.6 AIO is available at the LSE AIO web page [10]. All the external AIO patches including AIO support for buffered filesystem I/O, AIO poll and AIO support for pipes are available at [9].

12 Legal Statement

This work represents the view of the authors and does not necessarily represent the view of IBM.

IBM, DB2 and DB2 Universal Database are registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds.

Pentium is a trademark of Intel Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

13 Disclaimer

The benchmarks discussed in this paper were conducted for research purposes only, under laboratory conditions. Results will not be realized in all computing environments.

References

[1] Suparna Bhattacharya, Badari Pulavarthy, Steven Pratt, and Janet Morgan. Asynchronous i/o support for linux 2.5. In *Proceedings of the Linux Symposium*. Linux Symposium, Ottawa, July 2003. `http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Pulavarty-OLS2003.pdf`.

[2] Chris Mason. aio-stress microbenchmark. `ftp://ftp.suse.com/pub/people/mason/utils/aio-stress.c`.

[3] Stephen C. Tweedie. Posting on dio races in 2.4. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=105597840711609&w=2`.

[4] Andrew Morton. O_sync speedup patch. `http://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6/2.6.0/2.6.0-mm1/broken-out/O_SYNC-speedup-2.patch`.

[5] Daniel McNeil. Posting on synchronized writeback races. `http://marc.theaimsgroup.com/?l=linux-aio&m=107671729611002&w=2`.

[6] Andrew Morton. Posting on in-order tagged radix tree walk based vfs writeback. `http://marc.theaimsgroup.com/?l=bk-commits-head&m=108184544016117&w=2`.

[7] William Lee Irwin. Filtered wakeup patch. `http://marc.theaimsgroup.com/?l=bk-commits-head&m=108459430513660&w=2`.

[8] Transaction processing performance council. `http://www.tpc.org`.

[9] Suparna Bhattacharya (with contributions from Andrew Morton & Chris Mason). Additional 2.6 Linux Kernel Asynchronous I/O patches. `http://www.kernel.org/pub/linux/kernel/people/suparna/aio`.

[10] LSE team. Kernel Asynchronous I/O (AIO) Support for Linux. `http://lse.sf.net/io/aio.html`.

Methods to Improve Bootup Time in Linux

Tim R. Bird
Sony Electronics
tim.bird@am.sony.com

Abstract

This paper presents several techniques for reducing the bootup time of the Linux kernel, including Execute-In-Place (XIP), avoidance of `calibrate_delay()`, and reduced probing by certain drivers and subsystems. Using a variety of techniques, the Linux kernel can be booted on embedded hardware in under 500 milliseconds. Current efforts and future directions of work to improve bootup time are described.

1 Introduction

Users of consumer electronics products expect their devices to be available for use very soon after being turned on. Configurations of Linux for desktop and server markets exhibit boot times in the range of 20 seconds to a few minutes, which is unacceptable for many consumer products.

No single item is responsible for overall poor boot time performance. Therefore a number of techniques must be employed to reduce the boot up time of a Linux system. This paper presents several techniques which have been found to be useful for embedded configurations of Linux.

2 Overview of Boot Process

The entire boot process of Linux can be roughly divided into 3 main areas: firmware, kernel, and user space. The following is a list of events during a typical boot sequence:

1. power on
2. firmware (bootloader) starts
3. kernel decompression starts
4. kernel start
5. user space start
6. RC script start
7. application start
8. first available use

This paper focuses on techniques for reducing the bootup time up until the start of user space. That is, techniques are described which reduce the firmware time, and the kernel start time. This includes activities through the completion of event 4 in the list above.

The actual kernel execution begins with the routine `start_kernel()`, in the file `init/main.c`.

An overview of major steps in the initialization sequence of the kernel is as follows:

- `start_kernel()`
  - init architecture
  - init interrupts
  - init memory
  - start idle thread
  - call `rest_init()`
    - start ‘init’ kernel thread

The `init` kernel thread performs a few other tasks, then calls `do_basic_setup()`, which calls `do_initcalls()`, to run through the array of initialization routines for drivers statically linked in the kernel. Finally, this thread switches to user space by `execve`-ing to the first user space program, usually `/sbin/init`.

- `init` (kernel thread)
  - call `do_basic_setup()`
    - call `do_initcalls()`
      - init buses and drivers
  - prepare and mount root filesystem
  - call `run_init_process()`
    - call `execve()` to start user space process

3 Typical Desktop Boot Time

The boot times for a typical desktop system were measured and the results are presented below, to give an indication of the major areas in the kernel where time is spent. While the numbers in these tests differ somewhat from those for a typical embedded system, it is useful to see these to get an idea of where some of the trouble spots are for kernel booting.

3.1 System

An HP XW4100 Linux workstation system was used for these tests, with the following characteristics:

- Pentium 4 HT processor, running at 3GHz
- 512 MB RAM
- Western Digital 40G hard drive on hda
- Generic CDROM drive on hdc

3.2 Measurement method

The kernel used was 2.6.6, with the KFI patch applied. KFI stands for “Kernel Function Instrumentation”. This is an in-kernel system to measure the duration of each function executed during a particular profiling run. It uses the `-finstrument-functions` option of `gcc` to instrument kernel functions with callouts on each function entry and exit. This code was authored by developers at MontaVista Software, and a patch for 2.6.6 is available, although the code is not ready (as of the time of this writing) for general publication. Information about KFI and the patch are available at:

```
http://tree.celinuxforum.org/pubwiki
/moin.cgi
/KernelFunctionInstrumentation
```

3.3 Key delays

The average time for kernel startup of the test system was about 7 seconds. This was the amount of time for just the kernel and NOT the firmware or user space. It corresponds to the period of time between events 4 and 5 in the boot sequence listed in Section 2.

Some key delays were found in the kernel startup on the test system. Table 1 shows some of the key routines where time was spent during bootup. These are the low-level routines where significant time was spent inside the functions themselves, rather than in subroutines called by the functions.

Kernel Function	No. of calls	Avg. call time	Total time
delay_tsc	5153	1	5537
default_idle	312	1	325
get_cmos_time	1	500	500
psmouse_sendbyte	44	2.4	109
pci_bios_find_device	25	1.7	44
atkbd_sendbyte	7	3.7	26
calibrate_delay	1	24	24

Note: Times are in milliseconds.

Table 1: Functions consuming lots of time during a typical desktop Linux kernel startup.

Note that over 80% of the total time of the bootup (almost 6 seconds out of 7) was spent busywaiting in `delay_tsc()` or spinning in the routine `default_idle()`. It appears that great reductions in total bootup time could be achieved if these delays could be reduced, or if it were possible to run some initialization tasks concurrently.

Another interesting point is that the routine `get_cmos_time()` was extremely variable in the length of time it took. Measurements of its duration ranged from under 100 milliseconds to almost one second. This routine, and methods to avoid this delay and variability, are discussed in section 9.

3.4 High-level delay areas

Since `delay_tsc()` is used (via various `delay` mechanisms) for busywaiting by a number of different subsystems, it is helpful to identify the higher-level routines which end up invoking this function.

Table 2 shows some high-level routines called during kernel initialization, and the amount of time they took to complete on the test machine. Duration times marked with a tilde denote functions which were highly variable in duration.

Kernel Function	Duration time
ide_init	3327
time_init	~500
isapnp_init	383
i8042_init	139
prepare_namespace	~50
calibrate_delay	24

Note: Times are in milliseconds.

Table 2: High-level delays during a typical startup.

For a few of these, it is interesting to examine the call sequences underneath the high-level routines. This shows the connection between the high-level routines that are taking a long time to complete and the functions where the time is actually being spent.

Figures 1 and 2 show some call sequences for high-level calls which take a long time to complete.

In each call tree, the number in parentheses is the number of times that the routine was called by the parent in this chain. Indentation shows the call nesting level.

For example, in Figure 1, `do_probe()` is called a total of 31 times by `probe_hwif()`, and it calls `ide_delay_50ms()` 78 times, and `try_to_identify()` 8 times.

The timing data for the test system showed that IDE initialization was a significant contributor to overall bootup time. The call sequence underneath `ide_init()` shows that a large number of calls are made to the routine `ide_delay_50ms()`, which in turn calls

```
ide_init->
 probe_for_hwifs(1)->
  ide_scan_pcibus(1)->
   ide_scan_pci_dev(2)->
    piix_init_one(2)->
     init_setup_piix(2)->
      ide_setup_pci_device(2)->
       probe_hwif_init(2)->
        probe_hwif(4)->
         do_probe(31)->
          ide_delay_50ms(78)->
           __const_udelay(3900)->
            __delay(3900)->
             delay_tsc(3900)
          try_to_identify(8)->
           actual_try_to_identify(8)->
            ide_delay_50ms(24)->
             __const_udelay(1200)->
              __delay(1200)->
               delay_tsc(1200)
```

Figure 1: IDE init call tree

```
isapnp_init->
 isapnp_isolate(1)->
  isapnp_isolate_rdp_select(1)->
   __const_udelay(25)->
    __delay(25)->
     delay_tsc(25)
   isapnp_key(18)->
    __const_udelay(18)->
     __delay(18)->
      delay_tsc(18)
```

Figure 2: ISAPnP init call tree

`__const_udelay()` very many times. The busywaits in `ide_delay_50ms()` alone accounted for over 5 seconds, or about 70% of the total boot up time.

Another significant area of delay was the initialization of the ISAPnP system. This took about 380 milliseconds on the test machine.

Both the mouse and the keyboard drivers used crude busywaits to wait for acknowledgements from their respective hardware.

Finally, the routine `calibrate_delay()` took about 25 milliseconds to run, to compute the value of `loops_per_jiffy` and print (the related) `BogoMips` for the machine.

The remaining sections of this paper discuss various specific methods for reducing bootup time for embedded and desktop systems. Some of these methods are directly related to some of the delay areas identified in this test configuration.

4 Kernel Execute-In-Place

A typical sequence of events during bootup is for the bootloader to load a compressed kernel image from either disk or Flash, placing it into RAM. The kernel is decompressed, either during or just after the copy operation. Then the kernel is executed by jumping to the function `start_kernel()`.

Kernel Execute-In-Place (XIP) is a mechanism where the kernel instructions are executed directly from ROM or Flash.

In a kernel XIP configuration, the step of copying the kernel code segment into RAM is omitted, as well as any decompression step. Instead, the kernel image is stored uncompressed in ROM or Flash. The kernel data segments still need to be initialized in RAM, but by eliminating the text segment copy and decompression, the overall effect is a reduction in the time required for the firmware phase of the bootup.

Table 3 shows the differences in time duration for various parts of the boot stage for a system booted with and without use of kernel XIP. The times in the table are shown in milliseconds. The table shows that using XIP in this configuration significantly reduced the time to copy the kernel to RAM (because only the data segments were copied), and completely eliminated the time to decompress the kernel (453 milliseconds). However, the kernel initialization time increased slightly in the XIP configuration, for a net savings of 463 milliseconds.

In order to support an Execute-In-Place con-

Boot Stage	Non-XIP time	XIP time
Copy kernel to RAM	85	12
Decompress kernel	453	0
Kernel initialization	819	882
Total kernel boot time	1357	894

Note: Times are in milliseconds. Results are for PowerPC 405 LP at 266 MHz

Table 3: Comparison of Non-XIP vs. XIP bootup times

figuration, the kernel must be compiled and linked so that the code is ready to be executed from a fixed memory location. There are examples of XIP configurations for ARM, MIPS and SH platforms in the CELinux source tree, available at: `http://tree.celinuxforum.org/`

4.1 XIP Design Tradeoffs

There are tradeoffs involved in the use of XIP. First, it is common for access times to flash memory to be greater than access times to RAM. Thus, a kernel executing from Flash usually runs a bit slower than a kernel executing from RAM. Table 4 shows some of the results from running the `lmbench` benchmark on system, with the kernel executing in a standard non-XIP configuration versus an XIP configuration.

Operation	Non-XIP	XIP
stat() syscall	22.4	25.6
fork a process	4718	7106
context switching for 16 processes and 64k data size	932	1109
pipe communication	248	548

Note: Times are in microseconds. Results are for lmbench benchmark run on OMAP 1510 (ARM9 at 168 MHz) processor

Table 4: Comparison of Non-XIP and XIP performance

Some of the operations in the benchmark took significantly longer with the kernel run in the XIP configuration. Most individual operations took about 20% to 30% longer. This performance penalty is suffered permanently while the kernel is running, and thus is a serious drawback to the use of XIP for reducing bootup time.

A second tradeoff with kernel XIP is between the sizes of various types of memory in the system. In the XIP configuration the kernel must be stored uncompressed, so the amount of Flash required for the kernel increases, and is usually about doubled, versus a compressed kernel image used with a non-XIP configuration. However, the amount of RAM required for the kernel is decreased, since the kernel code segment is never copied to RAM. Therefore, kernel XIP is also of interest for reducing the runtime RAM footprint for Linux in embedded systems.

There is additional research under way to investigate ways of reducing the performance impact of using XIP. One promising technique appears to be the use of "partial-XIP," where a highly active subset of the kernel is loaded into RAM, but the majority of the kernel is executed in place from Flash.

5 Delay Calibration Avoidance

One time-consuming operation inside the kernel is the process of calibrating the value used for delay loops. One of the first routines in the kernel, `calibrate_delay()`, executes a series of delays in order to determine the correct value for a variable called `loops_per_jiffy`, which is then subsequently used to execute short delays in the kernel.

The cost of performing this calibration is, interestingly, independent of processor speed. Rather, it is dependent on the number of iterations required to perform the calibration, and the length of each iteration. Each iteration requires 1 jiffy, which is the length of time defined by the HZ variable.

In 2.4 versions of the Linux kernel, most platforms defined HZ as 100, which makes the length of a jiffy 10 milliseconds. A typical number of iterations for the calibration operation is 20 to 25, making the total time required for this operation about 250 milliseconds.

In 2.6 versions of the Linux kernel, a few platforms (notably i386) have changed HZ to 1000, making the length of a jiffy 1 millisecond. On those platforms, the typical cost of this calibration operation has decreased to about 25 milliseconds. Thus, the benefit of eliminating this operation on most standard desktop systems has been reduced. However, for many embedded systems, HZ is still defined as 100, which makes bypassing the calibration useful.

It is easy to eliminate the calibration operation. You can directly edit the code in `init/main.c:calibrate_delay()` to hardcode a value for `loops_per_jiffy`, and avoid the calibration entirely. Alternatively, there is a patch available at `http://tree.celinuxforum.org/pubwiki/moin.cgi/PresetLPJ`

This patch allows you to use a kernel configuration option to specify a value for `loops_per_jiffy` at kernel compile time. Alternatively, the patch also allows you to use a kernel command line argument to specify a preset value for `loops_per_jiffy` at kernel boot time.

6 Avoiding Probing During Bootup

Another technique for reducing bootup time is to avoid probing during bootup. As a general technique, this can consist of identifying hardware which is known not to be present on one's machine, and making sure the kernel is compiled without the drivers for that hardware.

In the specific case of IDE, the kernel supports options at the command line to allow the user to avoid performing probing for specific interfaces and devices. To do this, you can use the IDE and harddrive `noprobe` options at the kernel command line. Please see the file `Documentation/ide.txt` in the kernel source tree for details on the syntax of using these options.

On the test machine, IDE `noprobe` options were used to reduce the amount of probing during startup. The test machine had only a hard drive on hda (ide0 interface, first device) and a CD-ROM drive on hdc (ide1 interface, first device).

In one test, `noprobe` options were specified to suppress probing of non-used interfaces and devices. Specifically, the following arguments were added to the kernel command line:

```
hdb=none hdd=none ide2=noprobe
```

The kernel was booted and the result was that the function `ide_delay_50ms()` was called only 68 times, and `delay_tsc()` was called only 3453 times. During a regular kernel boot without these options specified, the function `ide_delay_50ms()` is called 102 times, and `delay_tsc()` is called 5153 times. Each call to `delay_tsc()` takes about 1 millisecond, so the total time savings from using these options was 1700 milliseconds.

These IDE `noprobe` options have been available at least since the 2.4 kernel series, and are an easy way to reduce bootup time, without even having to recompile the kernel.

7 Reducing Probing Delays

As was noted on the test machine, IDE initialization takes a significant percentage of the total bootup time. Almost all of this time is spent busywaiting in the routine `ide_delay_50ms()`.

It is trivial to modify the value of the timeout used in this routine. As an experiment, this code (located in the file `drivers/ide/ide.c`) was modified to only delay 5 milliseconds instead of 50 milliseconds.

The results of this change were interesting. When a kernel with this change was run on the test machine, the total time for the `ide_init()` routine dropped from 3327 milliseconds to 339 milliseconds. The total time spent in all invocations of `ide_delay_50ms()` was reduced from 5471 milliseconds to 552 milliseconds. The overall bootup time was reduced accordingly, by about 5 seconds.

The ide devices were successfully detected, and the devices operated without problem on the test machine. However, this configuration was not tested exhaustively.

Reducing the duration of the delay in the `ide_delay_50ms()` routine provides a substantial reduction in the overall bootup time for the kernel on a typical desktop system. It also has potential use in embedded systems where PCI-based IDE drives are used.

However, there are several issues with this modification that need to be resolved. This change may not support legacy hardware which requires long delays for proper probing and initializing. The kernel code needs to be analyzed to determine if any callers of this routine really need the 50 milliseconds of delay that they are requesting. Also, it should be determined whether this call is used only in initialization context or if it is used during regular runtime use of IDE devices also.

Also, it may be that 5 milliseconds does not represent the lowest possible value for this delay. It is possible that this value will need to be tuned to match the hardware for a particular machine. This type of tuning may be acceptable in the embedded space, where the hardware configuration of a product may be fixed. But it may be too risky to use in desktop configurations of Linux, where the hardware is not known ahead of time.

More experimentation, testing and validation are required before this technique should be used.

IMPORTANT NOTE: You should probably not experiment with this modification on production hardware unless you have evaluated the risks.

8 Using the "quiet" Option

One non-obvious method to reduce overhead during booting is to use the `quiet` option on the kernel command line. This option changes the loglevel to 4, which suppresses the output of regular (non-emergency) printk messages. Even though the messages are not printed to the system console, they are still placed in the kernel printk buffer, and can be retrieved after bootup using the `dmesg` command.

When embedded systems boot with a serial console, the speed of printing the characters to the console is constrained by the speed of the serial output. Also, depending on the driver, some VGA console operations (such as scrolling the screen) may be performed in software. For slow processors, this may take a significant amount of time. In either case, the cost of performing output of printk messages during bootup may be high. But it is easily eliminated using the `quiet` command line option.

Table 5 shows the difference in bootup time of using the `quiet` option and not, for two different systems (one with a serial console and one with a VGA console).

9 RTC Read Synchronization

One routine that potentially takes a long time during kernel startup is `get_cmos_time()`. This routine is used to read the value of the external real-time clock (RTC) when the kernel boots. Currently, this routine delays until the edge of the next second rollover, in order to ensure that the time value in the kernel is accurate with respect to the RTC.

However, this operation can take up to one full second to complete, and thus introduces up to 1 second of variability in the total bootup time. For systems where the target bootup time is under 1 second, this variability is unacceptable.

The synchronization in this routine is easy to remove. It can be eliminated by removing the first two loops in the function `get_cmos_time()`, which is located in `include/asm-i386/mach-default/mach_time.h` for the i386 architecture. Similar routines are present in the kernel source tree for other architectures.

When the synchronization is removed, the routine completes very quickly.

One tradeoff in making this modification is that the time stored by the Linux kernel is no longer completely synchronized (to the boundary of a second) with the time in the machine's realtime clock hardware. Some systems save the system time back out to the hardware clock on system shutdown. After numerous bootups and shutdowns, this lack of synchronization will cause the realtime clock value to drift from the correct time value.

Since the amount of un-synchronization is up to a second per boot cycle, this drift can be significant. However, for some embedded applications, this drift is unimportant. Also, in some situations the system time may be synchronized with an external source anyway, so the drift, if any, is corrected under normal circumstances soon after booting.

10 User space Work

There are a number of techniques currently available or under development for user space bootup time reductions. These techniques are (mostly) outside the scope of kernel development, but may provide additional benefits for reducing overall bootup time for Linux systems.

Some of these techniques are mentioned briefly in this section.

10.1 Application XIP

One technique for improving application startup speed is application XIP, which is similar to the kernel XIP discussed in this paper. To support application XIP the kernel must be compiled with a file system where files can be stored linearly (where the blocks for a file are stored contiguously) and uncompressed. One file system which supports this is CRAMFS, with the LINEAR option turned on. This is a read-only file system.

With application XIP, when a program is executed, the kernel program loader maps the text segments for applications directly from the flash memory of the file system. This saves the time required to load these segments into system RAM.

Platform	Speed	console type	w/o `quiet` option	with `quiet` option	difference
SH-4 SH7751R	240 MHz	VGA	637	461	176
OMAP 1510 (ARM 9)	168 MHz	serial	551	280	271

Note: Times are in milliseconds

Table 5: Bootup time with and without the `quiet` option

10.2 RC Script improvements

Also, there are a number of projects which strive to decrease total bootup time of a system by parallelizing the execution of the system run-command scripts ("RC scripts"). There is a list of resources for some of these projects at the following web site:

`http://tree.celinuxforum.org/pubwiki/moin.cgi/BootupTimeWorkingGroup`

Also, there has been some research conducted in reducing the overhead of running RC scripts. This consists of modifying the multi-function program `busybox` to reduce the number and cost of forks during RC script processing, and to optimize the usage of functions builtin to the `busybox` program. Initial testing has shown a reduction from about 8 seconds to 5 seconds for a particular set of Debian RC scripts on an OMAP 1510 (ARM 9) processor, running at 168 MHz.

11 Results

By use of the some of the techniques mentioned in this paper, as well as additional techniques, Sony was able to boot a 2.4.20-based Linux system, from power on to user space display of a greeting image and sound playback, in 1.2 seconds. The time from power on to the end of kernel initialization (first user space instruction) in this configuration was about 110 milliseconds. The processor was a TI OMAP 1510 processor, with an ARM9-based core, running at 168 MHz.

Some of the techniques used for reducing the bootup time of embedded systems can also be used for desktop or server systems. Often, it is possible, with rather simple and small modifications, to decrease the bootup time of the Linux kernel to only a few seconds. In the desktop configuration of Linux presented here, techniques from this paper were used to reduced the total bootup time from around 7 seconds to around 1 second. This was with no loss of functionality that the author could detect (with limited testing).

12 Further Research

As stated in the beginning of the paper, numerous techniques can be employed to reduce the overall bootup time of Linux systems. Further work continues or is needed in a number of areas.

12.1 Concurrent Driver Init

One area of additional research that seems promising is to structure driver initializations in the kernel so that they can proceed in parallel. For some items, like IDE initialization, there are large delays as buses and devices are probed and initialized. The time spent in such busywaits could potentially be used to perform other startup tasks, concurrently with the initializations waiting for hardware events to occur or time out.

The big problem to be addressed with concurrent initialization is to identify what kernel startup activities can be allowed to occur in parallel. The kernel init sequence is already a carefully ordered sequence of events to make sure that critical startup dependencies are observed. Any system of concurrent driver initialization will have to provide a mechanism to guarantee sequencing of initialization tasks which have order dependencies.

12.2 Partial XIP

Another possible area of further investigation, which has already been mentioned, is "partial XIP," whereby the kernel is executed *mostly* in-place. Prototype code already exists which demonstrates the mechanisms necessary to move a subset of an XIP-configured kernel into RAM, for faster code execution. The key to making partial kernel XIP useful will be to ensure correct identification (either statically or dynamically) of the sections of kernel code that need to be moved to RAM. Also, experimentation and testing need to be performed to determine the appropriate tradeoff between the size of the RAM-based portion of the kernel, and the effect on bootup time and system runtime performance.

12.3 Pre-linking and Lazy Linking

Finally, research is needed into reducing the time required to fixup links between programs and their shared libraries.

Two systems that have been proposed and experimented with are pre-linking and lazy linking. Pre-linking involves fixing the location in virtual memory of the shared libraries for a system, and performing fixups on the programs of the system ahead of time. Lazy linking consists of only performing fixups on demand as library routines are called by a running program.

Additional research is needed with both of these techniques to determine if they can provide benefit for current Linux systems.

13 Credits

This paper is the result of work performed by the Bootup Time Working Group of the CE Linux forum (of which the author is Chair). I would like to thank developers at some of CELF's member companies, including Hitachi, Intel, Mitsubishi, MontaVista, Panasonic, and Sony, who contributed information or code used in writing this paper.

Linux on NUMA Systems

Martin J. Bligh
mbligh@aracnet.com
Matt Dobson
colpatch@us.ibm.com
Darren Hart
dvhltc@us.ibm.com
Gerrit Huizenga
gh@us.ibm.com

Abstract

NUMA is becoming more widespread in the marketplace, used on many systems, small or large, particularly with the advent of AMD Opteron systems. This paper will cover a summary of the current state of NUMA, and future developments, encompassing the VM subsystem, scheduler, topology (CPU, memory, I/O layouts including complex non-uniform layouts), userspace interface APIs, and network and disk I/O locality. It will take a broad-based approach, focusing on the challenges of creating subsystems that work for all machines (including AMD64, PPC64, IA-32, IA-64, etc.), rather than just one architecture.

1 What is a NUMA machine?

NUMA stands for non-uniform memory architecture. Typically this means that not all memory is the same "distance" from each CPU in the system, but also applies to other features such as I/O buses. The word "distance" in this context is generally used to refer to both latency and bandwidth. Typically, NUMA machines can access any resource in the system, just at different speeds.

NUMA systems are sometimes measured with a simple "NUMA factor" ratio of N:1—meaning that the latency for a cache miss memory read from remote memory is *N* times the latency for that from local memory (for NUMA machines, $N > 1$). Whilst such a simple descriptor is attractive, it can also be highly misleading, as it describes latency only, not bandwidth, on an uncontended bus (which is not particularly relevant or interesting), and takes no account of inter-node caches.

The term *node* is normally used to describe a grouping of resources—e.g., CPUs, memory, and I/O. On some systems, a node may contain only some types of resources (e.g., only memory, or only CPUs, or only I/O); on others it may contain all of them. The interconnect between nodes may take many different forms, but can be expected to be higher latency than the connection within a node, and typically lower bandwidth.

Programming for NUMA machines generally implies focusing on *locality*—the use of resources close to the device in question, and trying to reduce traffic between nodes; this type of programming generally results in better application throughput. On some machines with high-speed cross-node interconnects, better performance may be derived under certain workloads by "striping" accesses across multiple nodes, rather than just using local resources, in order to increase bandwidth. Whilst it is easy to demonstrate a benchmark that shows improvement via this method, it is difficult to be sure that the concept is generally benefical (i.e., with the machine under full load).

2 Why use a NUMA architecture to build a machine?

The intuitive approach to building a large machine, with many processors and banks of memory, would be simply to scale up the typical 2–4 processor machine with all resources attached to a shared system bus. However, restrictions of electronics and physics dictate that accesses slow as the length of the bus grows, and the bus is shared amongst more devices.

Rather than accept this global slowdown for a larger machine, designers have chosen to instead give fast access to a limited set of local resources, and reserve the slower access times for remote resources.

Historically, NUMA architectures have only been used for larger machines (more than 4 CPUs), but the advantages of NUMA have been brought into the commodity marketplace with the advent of AMD's x86-64, which has one CPU per node, and local memory for each processor. Linux supports NUMA machines of every size from 2 CPUs upwards (e.g., SGI have machines with 512 processors).

It might help to envision the machine as a group of standard SMP machines, connected by a very fast interconnect somewhat like a network connection, except that the transfers over that bus are transparent to the operating system. Indeed, some earlier systems were built exactly like that; the older Sequent NUMA-Q hardware uses a standard 450NX 4 processor chipset, with an SCI interconnect plugged into the system bus of each node to unify them, and pass traffic between them. The complex part of the implementation is to ensure cache-coherency across the interconnect, and such machines are often referred to as *CC-NUMA* (cache coherent NUMA). As accesses over the interconnect are transparent, it is possible to program such machines as if they were standard SMP machines (though the performance will be poor). Indeed, this is exactly how the NUMA-Q machines were first bootstrapped.

Often, we are asked why people do not use clusters of smaller machines, instead of a large NUMA machine, as clusters are cheaper, simpler, and have a better price:performance ratio. Unfortunately, it makes the programming of applications much harder; all of the intercommunication and load balancing now has to be more explicit. Some large applications (e.g., database servers) do not split up across multiple cluster nodes easily—in those situations, people often use NUMA machines. In addition, the interconnect for NUMA boxes is normally very low latency, and very high bandwidth, yielding excellent performance. The management of a single NUMA machine is also simpler than that of a whole cluster with multiple copies of the OS.

We could either have the operating system make decisions about how to deal with the architecture of the machine on behalf of the user processes, or give the userspace application an API to specify how such decisions are to be made. It might seem, at first, that the userspace application is in a better position to make such decisions, but this has two major disadvantages:

1. Every application must be changed to support NUMA machines, and may need to

be revised when a new hardware platform is released.

2. Applications are not in a good position to make global holistic decisions about machine resources, coordinate themselves with other applications, and balance decisions between them.

Thus decisions on process, memory and I/O placement are normally best left to the operating system, perhaps with some hints from userspace about which applications group together, or will use particular resources heavily. Details of hardware layout are put in one place, in the operating system, and tuning and modification of the necessary algorithms are done once in that central location, instead of in every application. In some circumstances, the application or system administrator will want to override these decisions with explicit APIs, but this should be the exception, rather than the norm.

3 Linux NUMA Memory Support

In order to manage memory, Linux requires a page descriptor structure (`struct page`) for each physical page of memory present in the system. This consumes approximately 1% of the memory managed (assuming 4K page size), and the structures are grouped into an array called `mem_map`. For NUMA machines, there is a separate array for each node, called `lmem_map`. The `mem_map` and `lmem_map` arrays are simple contiguous data structures accessed in a linear fashion by their offset from the beginning of the node. This means that the memory controlled by them is assumed to be physically contiguous.

NUMA memory support is enabled by `CONFIG_DISCONTIGMEM` and `CONFIG_NUMA`. A node descriptor called a `struct pgdata_t` is created for each node. Currently we do not support discontiguous memory within a node (though large gaps in the physical address space are acceptable between nodes). Thus we must still create page descriptor structures for "holes" in memory within a node (and then mark them invalid), which will waste memory (potentially a problem for large holes).

Dave McCracken has picked up Daniel Phillips' earlier work on a better data structure for holding the page descriptors, called `CONFIG_NONLINEAR`. This will allow the mapping of discontigous memory ranges inside each node, and greatly simplify the existing code for discontiguous memory on non-NUMA machines.

`CONFIG_NONLINEAR` solves the problem by creating an artificial layer of linear addresses. It does this by dividing the physical address space into fixed size sections (akin to very large pages), then allocating an array to allow translations from linear physical address to true physical address. This added level of indirection allows memory with widely differing true physical addresses to appear adjacent to the page allocator and to be in the same zone, with a single struct page array to describe them. It also provides support for memory hotplug by allowing new physical memory to be added to an existing zone and struct page array.

Linux normally allocates memory for a process on the local node, i.e., the node that the process is currently running on. `alloc_pages` will call `alloc_pages_node` for the current processor's node, which will pass the relevant zonelist (`pgdat->node_zonelists`) to the core allocator (`__alloc_pages`). The zonelists are built by `build_zonelists`, and are set up to allocate memory in a round-robin fashion, starting from the local node (this creates a roughly even distribution of memory pressure).

In the interest of reducing cross-node traffic, and reducing memory access latency for frequently accessed data and text, it is desirable to replicate any such memory that is read-only to each node, and use the local copy on any accesses, rather than a remote copy. The obvious candidates for such replication are the kernel text itself, and the text of shared libraries such as libc. Of course, this faster access comes at the price of increased memory usage, but this is rarely a problem on large NUMA machines. Whilst it might be technically possible to replicate read/write mappings, this is complex, of dubious utility, and is unlikely to be implemented.

Kernel text is assumed by the kernel itself to appear at a fixed virtual address, and to change this would be problematic. Hence the easiest way to replicate it is to change the virtual to physical mappings for each node to point at a different address. On IA-64, this is easy, since the CPU provides hardware assistance in the form of a pinned TLB entry.

On other architectures this proves more difficult, and would depend on the structure of the pagetables. On IA-32 with PAE enabled, as long as the user-kernel split is aligned on a PMD boundary, we can have a separate kernel PMD for each node, and point the vmalloc area (which uses small page mappings) back to a globally shared set of PTE pages. The PMD entries for the `ZONE_NORMAL` areas normally never change, so this is not an issue, though there is an issue with `ioremap_nocache` that can change them (GART trips over this) and speculative execution means that we will have to deal with that (this can be a slow-path that updates all copies of the PMDs though).

Dave Hansen has created a patch to replicate read only pagecache data, by adding a per-node data structure to each node of the pagecache radix tree. As soon as any mapping is opened for write, the replication is collapsed, making it safe. The patch gives a 5%–40% increase in performance, depending on the workload.

In the 2.6 Linux kernel, we have a per-node LRU for page management and a per-node LRU lock, in place of the global structures and locks of 2.4. Not only does this reduce contention through finer grained locking, it also means we do not have to search other nodes' page lists to free up pages on one node which is under memory pressure. Moreover, we get much better locality, as only the local kswapd process is accessing that node's pages. Before splitting the LRU into per-node lists, we were spending 50% of the system time during a kernel compile just spinning waiting for `pagemap_lru_lock` (which was the biggest global VM lock at the time). Contention for the `pagemap_lru_lock` is now so small it is not measurable.

4 Sched Domains—a Topology-aware Scheduler

The previous Linux scheduler, the O(1) scheduler, provided some needed improvements to the 2.4 scheduler, but shows its age as more complex system topologies become more and more common. With technologies such as NUMA, Symmetric Multi-Threading (SMT), and variations and combinations of these, the need for a more flexible mechanism to model system topology is evident.

4.1 Overview

In answer to this concern, the mainline 2.6 tree (linux-2.6.7-rc1 at the time of this writing) contains an updated scheduler with support for generic CPU topologies with a data structure, `struct sched_domain`, that models the architecture and defines scheduling policies.

Simply speaking, sched domains group CPUs together in a hierarchy that mimics that of the physical hardware. Since CPUs at the bottom of the hierarchy are most closely related (in terms of memory access), the new scheduler performs load balancing most often at the lower domains, with decreasing frequency at each higher level.

Consider the case of a machine with two SMT CPUs. Each CPU contains a pair of virtual CPU siblings which share a cache and the core processor. The machine itself has two physical CPUs which share main memory. In such a situation, treating each of the four effective CPUs the same would not result in the best possible performance. With only two tasks, for example, the scheduler should place one on CPU0 and one on CPU2, and not on the two virtual CPUs of the same physical CPU. When running several tasks it seems natural to try to place newly ready tasks on the CPU they last ran on (hoping to take advantage of cache warmth). However, virtual CPU siblings share a cache; a task that was running on CPU0, then blocked, and became ready when CPU0 was running another task and CPU1 was idle, would ideally be placed on CPU1. Sched domains provide the structures needed to realize these sorts of policies. With sched domains, each physical CPU represents a domain containing the pair of virtual siblings, each represented in a `sched_group` structure. These two domains both point to a parent domain which contains all four effective processors in two `sched_group` structures, each containing a pair of virtual siblings. Figure 1 illustrates this hierarchy.

Next consider a two-node NUMA machine with two processors per node. In this example there are no virtual sibling CPUs, and therefore no shared caches. When a task becomes ready and the processor it last ran on is busy, the scheduler needs to consider waiting until that CPU is available to take advantage of cache warmth. If the only available CPU is on another node, the scheduler must carefully weigh the costs of migrating that task to another node, where access to its memory will be slower. The lowest level sched domains in a machine like this will contain the two processors of each node. These two CPU level domains each point to a parent domain which contains the two nodes. Figure 2 illustrates this hierarchy.

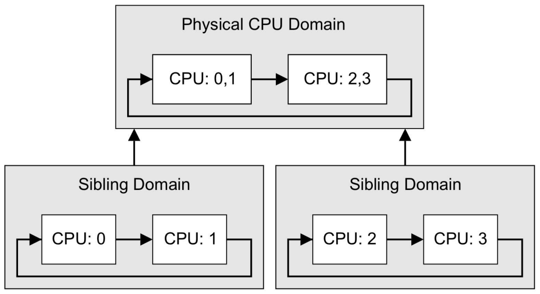


Figure 1: SMT Domains

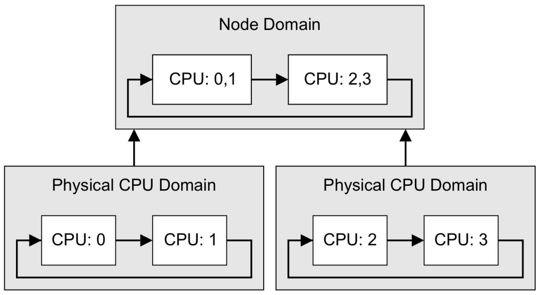


Figure 2: NUMA Domains

The next logical step is to consider an SMT NUMA machine. By combining the previous two examples, the resulting sched domain hierarchy has three levels, sibling domains, physical CPU domains, and the node domain. Figure 3 illustrates this hierarchy.

The unique AMD Opteron architecture warrants mentioning here as it creates a NUMA system on a single physical board. In this case, however, each NUMA node contains only one

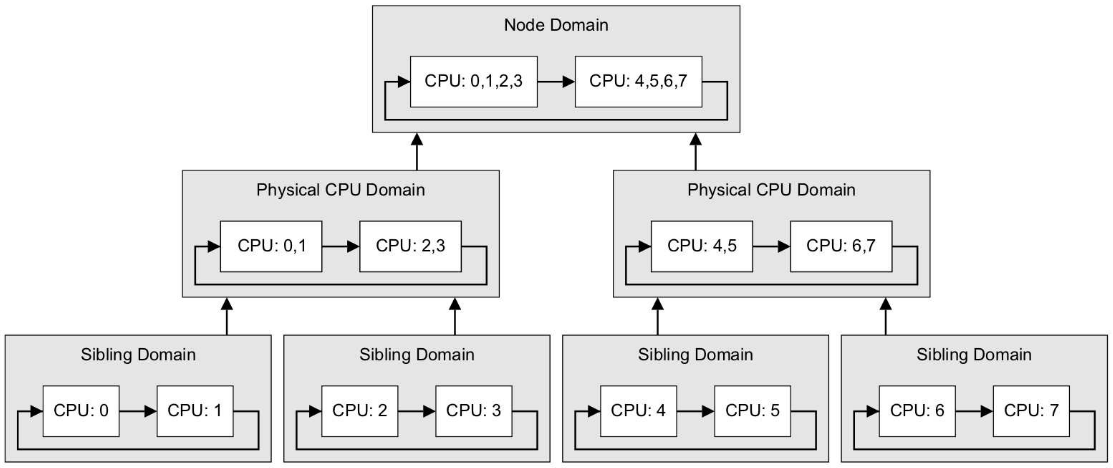


Figure 3: SMT NUMA Domains

physical CPU. Without careful consideration of this property, a typical NUMA sched domains hierarchy would perform badly, trying to load balance single CPU nodes often (an obvious waste of cycles) and between node domains only rarely (also bad since these actually represent the physical CPUs).

4.2 Sched Domains Implementation

4.2.1 Structure

The `sched_domain` structure stores policy parameters and flags and, along with the `sched_group` structure, is the primary building block in the domain hierarchy. Figure 4 describes these structures. The `sched_domain` structure is constructed into an upwardly traversable tree via the parent pointer, the top level domain setting parent to NULL. The groups list is a circular list of of `sched_group` structures which essentially define the CPUs in each child domain and the relative power of that group of CPUs (two physical CPUs are more powerful than one SMT CPU). The span member is simply a bit vector with a 1 for every CPU encompassed by that domain and is always the union of the bit vector stored in each element of the groups list. The remaining fields define the scheduling policy to be followed while dealing with that domain, see Section 4.2.2.

While the hierarchy may seem simple, the details of its construction and resulting tree structures are not. For performance reasons, the domain hierarchy is built on a per-CPU basis, meaning each CPU has a unique instance of each domain in the path from the base domain to the highest level domain. These duplicate structures do share the `sched_group` structures however. The resulting tree is difficult to diagram, but resembles Figure 5 for the machine with two SMT CPUs discussed earlier.

In accordance with common practice, each architecture may specify the construction of the sched domains hierarchy and the parameters and flags defining the various policies. At the time of this writing, only i386 and ppc64 defined custom construction routines. Both architectures provide for SMT processors and NUMA configurations. Without an architecture-specific routine, the kernel uses the default implementations in `sched.c`, which do take NUMA into account.

```
struct sched_domain {
        /* These fields must be setup */
        struct sched_domain *parent;         /* top domain must be null terminated */
        struct sched_group *groups;          /* the balancing groups of the domain */
        cpumask_t span;                      /* span of all CPUs in this domain */
        unsigned long min_interval;          /* Minimum balance interval ms */
        unsigned long max_interval;          /* Maximum balance interval ms */
        unsigned int busy_factor;            /* less balancing by factor if busy */
        unsigned int imbalance_pct;          /* No balance until over watermark */
        unsigned long long cache_hot_time;   /* Task considered cache hot (ns) */
        unsigned int cache_nice_tries;       /* Leave cache hot tasks for # tries */
        unsigned int per_cpu_gain;           /* CPU % gained by adding domain cpus */
        int flags;                           /* See SD_* */

        /* Runtime fields. */
        unsigned long last_balance;          /* init to jiffies. units in jiffies */
        unsigned int balance_interval;       /* initialise to 1. units in ms. */
        unsigned int nr_balance_failed;      /* initialise to 0 */
};

struct sched_group {
        struct sched_group *next;            /* Must be a circular list */
        cpumask_t cpumask;
        unsigned long cpu_power;
};
```

Figure 4: Sched Domains Structures

4.2.2 Policy

The new scheduler attempts to keep the system load as balanced as possible by running rebalance code when tasks change state or make specific system calls, we will call this *event balancing*, and at specified intervals measured in jiffies, called *active balancing*. Tasks must do something for event balancing to take place, while active balancing occurs independent of any task.

Event balance policy is defined in each `sched_domain` structure by setting a combination of the #defines of figure 6 in the flags member.

To define the policy outlined for the dual SMT processor machine in Section 4.1, the lowest level domains would set `SD_BALANCE_NEWIDLE` and `SD_WAKE_IDLE` (as there is no cache penalty for running on a different sibling within the same physical CPU), `SD_SHARE_CPUPOWER` to indicate to the scheduler that this is an SMT processor (the scheduler will give full physical CPU access to a high priority task by idling the virtual sibling CPU), and a few common flags `SD_BALANCE_EXEC`, `SD_BALANCE_CLONE`, and `SD_WAKE_AFFINE`. The next level domain represents the physical CPUs and will not set `SD_WAKE_IDLE` since cache warmth is a concern when balancing across physical CPUs, nor `SD_SHARE_CPUPOWER`. This domain adds the `SD_WAKE_BALANCE` flag to compensate for the removal of `SD_WAKE_IDLE`. As discussed earlier, an SMT NUMA system will have these two domains and another node-level domain. This domain removes the `SD_BALANCE_NEWIDLE` and `SD_WAKE_AFFINE` flags, resulting in far fewer balancing across nodes than within nodes. When one of these events occurs, the scheduler search up the domain hierarchy and performs the load balancing at the highest level domain with the corresponding flag set.

Active balancing is fairly straightforward and aids in preventing CPU-hungry tasks from hogging a processor, since these tasks may only

```
#define SD_BALANCE_NEWIDLE    1   /* Balance when about to become idle */
#define SD_BALANCE_EXEC       2   /* Balance on exec */
#define SD_BALANCE_CLONE      4   /* Balance on clone */
#define SD_WAKE_IDLE          8   /* Wake to idle CPU on task wakeup */
#define SD_WAKE_AFFINE       16   /* Wake task to waking CPU */
#define SD_WAKE_BALANCE      32   /* Perform balancing at task wakeup */
#define SD_SHARE_CPUPOWER    64   /* Domain members share cpu power */
```

Figure 6: Sched Domains Policies

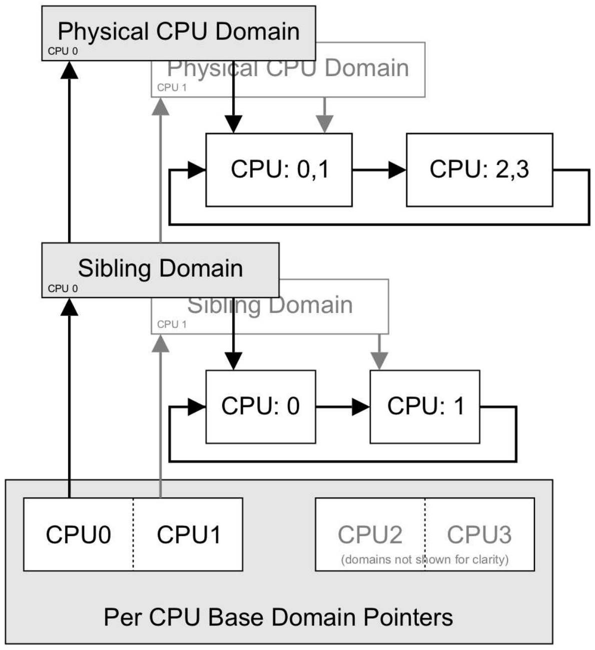


Figure 5: Per CPU Domains

rarely trigger event balancing. At each rebalance tick, the scheduler starts at the lowest level domain and works its way up, checking the `balance_interval` and `last_balance` fields to determine if that domain should be balanced. If the domain is already busy, the `balance_interval` is adjusted using the `busy_factor` field. Other fields define how out of balance a node must be before rebalancing can occur, as well as some sane limits on cache hot time and min and max balancing intervals. As with the flags for event balancing, the active balancing parameters are defined to perform less balancing at higher domains in the hierarchy.

4.3 Conclusions and Future Work

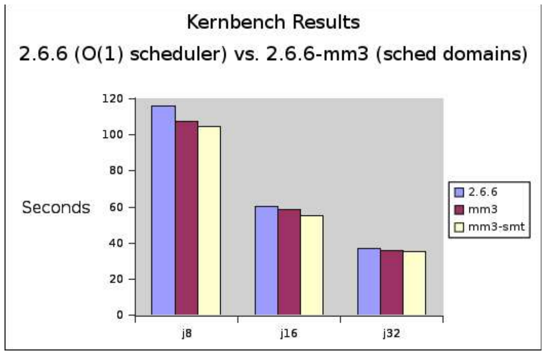


Figure 7: Kernbench Results

To compare the O(1) scheduler of mainline with the sched domains implementation in the mm tree, we ran kernbench (with the `-j` option to make set to 8, 16, and 32) on a 16 CPU SMT machine (32 virtual CPUs) on linux-2.6.6 and linux-2.6.6-mm3 (the latest tree with sched domains at the time of the benchmark) with and without `CONFIG_SCHED_SMT` enabled. The results are displayed in Figure 7. The O(1) scheduler evenly distributed compile tasks accross virtual CPUs, forcing tasks to share cache and computational units between virtual sibling CPUs. The sched domains implementation with `CONFIG_SCHED_SMT` enabled balanced the load accross physical CPUs, making far better use of CPU resources when running fewer tasks than CPUs (as in the j8 case) since each compile task would have exclusive access to the physical CPU. Surprisingly, sched domains (which would seem to have more overhead than the mainline scheduler) even showed improvement for the j32 case, where it doesn't benefit from balancing across physical CPUs before virtual CPUs as there are more tasks than virtual CPUs. Considering the sched domains implementation has not been heavily tested or tweaked for performance, some fine tuning is sure to further improve performance.

The sched domains structures replace the expanding set of `#ifdefs` of the O(1) scheduler, which should improve readability and maintainability. Unfortunately, the per CPU nature of the domain construction results in a non-intuitive structure that is difficult to work with. For example, it is natural to discuss the policy defined at "the" top level domain; unfortunately there are `NR_CPUS` top level domains and, since they are self-adjusting, each one could conceivably have a different set of flags and parameters. Depending on which CPU the scheduler was running on, it could behave radically differently. As an extension of this research, an effort to analyze the impact of a unified sched domains hierarchy is needed, one which only creates one instance of each domain.

Sched domains provides a needed structural change to the way the Linux scheduler views modern architectures, and provides the parameters needed to create complex scheduling policies that cater to the strengths and weaknesses of these systems. Currently only i386 and ppc64 machines benefit from arch specific construction routines; others must now step forward and fill in the construction and parameter setting routines for their architecture of choice. There is still plenty of fine tuning and performance tweaking to be done.

5 NUMA API

5.1 Introduction

One of the biggest impediments to the acceptance of a NUMA API for Linux was a lack of understanding of what its potential uses and users would be. There are two schools of thought when it comes to writing NUMA code. One says that the OS should take care of all the NUMA details, hide the NUMA-ness of the underlying hardware in the kernel and allow userspace applications to pretend that it's a regular SMP machine. Linux does this by having a process scheduler and a VMM that make intelligent decisions based on the hardware topology presented by arch-specific code. The other way to handle NUMA programming is to provide as much detail as possible about the system to userspace and allow applications to exploit the hardware to the fullest by giving scheduling hints, memory placement directives, etc., and the NUMA API for Linux handles this. Many applications, particularly larger applications with many concurrent threads of execution, cannot fully utilize a NUMA machine with the default scheduler and VM behavior. Take, for example, a database application that uses a large region of shared memory and many threads. This application may have a startup thread that initializes the environment, sets up the shared memory region, and forks off the worker threads. The default behavior of Linux's VM for NUMA is to bring pages into memory on the node that faulted them in. This behavior for our hypothetical app would mean that many pages would get faulted in by the startup thread on the node it is executing on, not necessarily on the node containing the processes that will actually use these pages. Also, the forked worker threads would get spread around by the scheduler to be balanced across all the nodes and their CPUs, but with no guarantees as to which threads would be associated with which nodes. The NUMA API and scheduler affinity syscalls allow this application to specify that its threads be pinned to particular CPUs and that its memory be placed on particular nodes. The application knows which threads will be working with which regions of memory, and is better equipped than the kernel to make those decisions.

The Linux NUMA API allows applications to give regions of their own virtual memory space specific allocation behaviors, called policies. Currently there are four supported policies: PREFERRED, BIND, INTERLEAVE, and DEFAULT. The DEFAULT policy is the simplest, and tells the VMM to do what it would normally do (ie: pre-NUMA API) for pages in the policied region, and fault them in from the local node. This policy applies to all regions, but is overridden if an application requests a different policy. The PREFERRED policy allows an application to specify one node that all pages in the policied region should come from. However, if the specified node has no available pages, the PREFERRED policy allows allocation to fall back to any other node in the system. The BIND policy allows applications to pass in a nodemask, a bitmap of nodes, that the VM is required to use when faulting in pages from a region. The fourth policy type, INTERLEAVE, again requires applications to pass in a nodemask, but with the INTERLEAVE policy, the nodemask is used to ensure pages are faulted in in a round-robin fashion from the nodes in the nodemask. As with the PREFERRED policy, the INTERLEAVE policy allows page allocation to fall back to other nodes if necessary. In addition to allowing a process to policy a specific region of its VM space, the NUMA API also allows a process to policy its entire VM space with a process-wide policy, which is set with a different syscall: `set_mempolicy()`. Note that process-wide policies are not persistent over swapping, however per-VMA policies are. Please also note that none of the policies will migrate existing (already allocated) pages to match the binding.

The actual implementation of the in-kernel policies uses a `struct mempolicy` that is hung off the `struct vm_area_struct`. This choice involves some tradeoffs. The first is that, previous to the NUMA API, the per-VMA structure was exactly 32 bytes on 32-bit architectures, meaning that multiple `vm_area_struct`s would fit conveniently in a single cacheline. The structure is now a little larger, but this allowed us to achieve a per-VMA granularity to policied regions. This is important in that it is flexible enough to bind a single page, a whole library, or a whole process' memory. This choice did lead to a second obstacle, however, which was for shared memory regions. For shared memory regions, we really want the policy to be shared amongst all processes sharing the memory, but VMAs are not shared across separate tasks. The solution that was implemented to work around this was to create a red-black tree of "shared policy nodes" for shared memory regions. Due to this, calls were added to the `vm_ops` structure which allow the kernel to check if a shared region has any policies and to easily retrieve these shared policies.

5.2 Syscall Entry Points

1. sys_mbind(unsigned long start, unsigned long len, unsigned long mode, unsigned long *nmask, unsigned long maxnode, unsigned flags);

   Bind the region of memory `[start, start+len)` according to `mode` and `flags` on the nodes enumerated in `nmask` and having a maximum possible node number of `maxnode`.

2. sys_set_mempolicy(int mode, unsigned

long *nmask, unsigned long maxnode);

Bind the entire address space of the current process according to `mode` on the nodes enumerated in `nmask` and having a maximum possible node number of `maxnode`.

3. sys_get_mempolicy(int *policy, unsigned long *nmask, unsigned long maxnode, unsigned long addr, unsigned long flags);

   Return the current binding's mode in `policy` and node enumeration in `nmask` based on the `maxnode`, `addr`, and `flags` passed in.

In addition to the raw syscalls discussed above, there is a user-level library called "libnuma" that attempts to present a more cohesive interface to the NUMA API, topology, and scheduler affinity functionality. This, however, is documented elsewhere.

5.3 At `mbind()` Time

After argument validation, the passed-in list of nodes is checked to make sure they are all online. If the node list is ok, a new memory policy structure is allocated and populated with the binding details. Next, the given address range is checked to make sure the vma's for the region are present and correct. If the region is ok, we proceed to actually install the new policy into all the vma's in that range. For most types of virtual memory regions, this involves simply pointing the `vma->vm_policy` to the newly allocated memory policy structure. For shared memory, hugetlbfs, and tmpfs, however, it's not quite this simple. In the case of a memory policy for a shared segment, a red-black tree root node is created, if it doesn't already exist, to represent the shared memory segment and is populated with "shared policy nodes." This allows a user to bind a single shared memory segment with multiple different bindings.

5.4 At Page Fault Time

There are now several new and different flavors of `alloc_pages()` style functions. Previous to the NUMA API, there existed `alloc_page()`, `alloc_pages()` and `alloc_pages_node()`. Without going into too much detail, `alloc_page()` and `alloc_pages()` both called `alloc_pages_node()` with the current node id as an argument. `alloc_pages_node()` allocated 2^{order} pages from a specific node, and was the only caller to the *real* page allocator, `__alloc_pages()`.

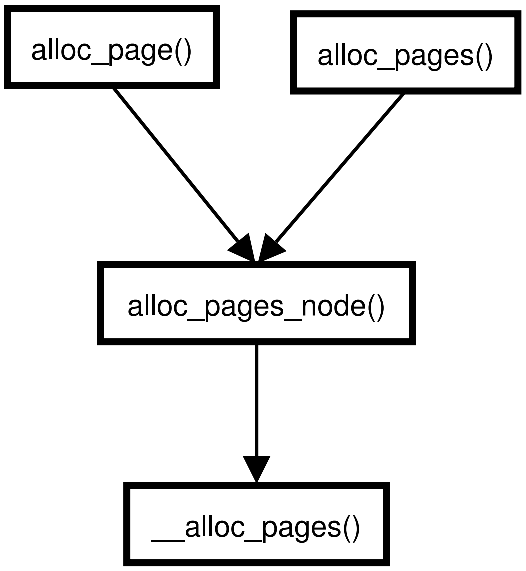


Figure 8: old `alloc_pages`

With the introduction of the NUMA API, non-NUMA kernels still retain the old `alloc_page*()` routines, but the NUMA allocators have changed. `alloc_pages_node()` and `__alloc_pages()`, the core routines remain untouched, but all calls to `alloc_page()`/`alloc_pages()` now end up calling `alloc_pages_current()`, a new function.

There has also been the addition of two new page allocation functions: `alloc_page_vma()` and `alloc_page_interleave()`. `alloc_pages_current()` checks that the system is not currently `in_interrupt()`, and if it isn't, uses the current process's process policy for allocation. If the system is currently in interrupt context, `alloc_pages_current()` falls back to the old default allocation scheme. `alloc_page_interleave()` allocates pages from regions that are bound with an interleave policy, and is broken out separately because there are some statistics kept for interleaved regions. `alloc_page_vma()` is a new allocator that allocates only single pages based on a per-vma policy. The `alloc_page_vma()` function is the only one of the new allocator functions that must be called explicity, so you will notice that some calls to `alloc_pages()` have been replaced by calls to `alloc_page_vma()` throughout the kernel, as necessary.

5.5 Problems/Future Work

There is no checking that the nodes requested are online at page fault time, so interactions with hotpluggable CPUs/memory will be tricky. There is an asymmetry between how you bind a memory region and a whole process's memory: One call takes a flags argument, and one doesn't. Also the `maxnode` argument is a bit strange, the get/set_affinity calls take a number of bytes to be read/written instead of a maximum CPU number. The `alloc_page_interleave()` function could be dropped if we were willing to forgo the statistics that are kept for interleaved regions. Again, a lack of symmetry exists because other types of policies aren't tracked in any way.

6 Legal statement

This work represents the view of the authors, and does not necessarily represent the view of IBM.

IBM, NUMA-Q and Sequent are registerd trademarks of International Business Machines Corporation in the United States, other contries, or both. Other company, product, or service names may be trademarks of service names of others.

References

[LWN] LWN Editor, "Scheduling Domains," `http://lwn.net/Articles/80911/`

[MM2] Linux 2.6.6-rc2/mm2 source, `http://www.kernel.org`

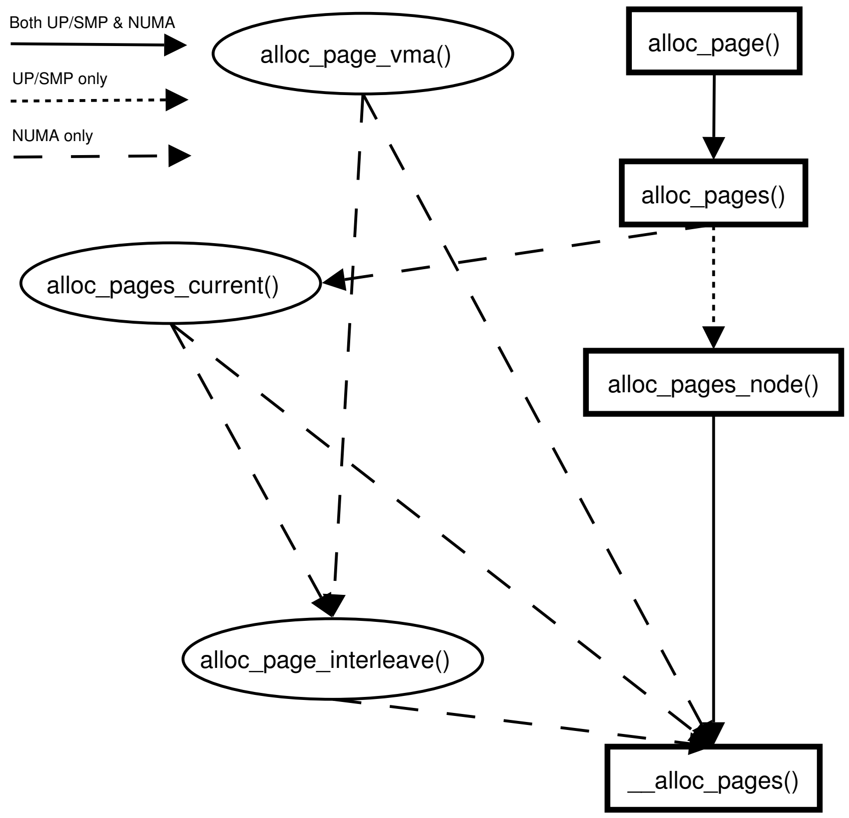


Figure 9: new `alloc_pages`

Improving Kernel Performance by Unmapping the Page Cache

James Bottomley
SteelEye Technology, Inc.
`James.Bottomley@SteelEye.com`

Abstract

The current DMA API is written on the founding assumption that the coherency is being done between the device and kernel virtual addresses. We have a different API for coherency between the kernel and userspace. The upshot is that every Process I/O must be flushed twice: Once to make the user coherent with the kernel and once to make the kernel coherent with the device. Additionally, having to map all pages for I/O places considerable resource pressure on x86 (where any highmem page must be separately mapped).

We present a different paradigm: Assume that by and large, read/write data is only required by a single entity (the major consumers of large multiply shared mappings are libraries, which are read only) and optimise the I/O path for this case. This means that any other shared consumers of the data (including the kernel) must separately map it themselves. The DMA API would be changed to perform coherence to the preferred address space (which could be the kernel). This is a slight paradigm shift, because now devices that need to peek at the data may have to map it first. Further, to free up more space for this mapping, we would break the assumption that any page in ZONE_NORMAL is automatically mapped into kernel space.

The benefits are that I/O goes straight from the device into the user space (for processors that have virtually indexed caches) and the kernel has quite a large unmapped area for use in kmapping highmem pages (for x86).

1 Introduction

In the Linux kernel[1] there are two addressing spaces: memory physical which is the location in the actual memory subsystem and CPU virtual, which is an address the CPU's Memory Management Unit (MMU) translates to a memory physical address internally. The Linux kernel operates completely in CPU virtual space, keeping separate virtual spaces for the kernel and each of the current user processes. However, the kernel also has to manage the mappings between physical and virtual spaces, and to do that it keeps track of where the physical pages of memory currently are.

In the Linux kernel, memory is split into zones in memory physical space:

- `ZONE_DMA`: A historical region where ISA DMAable memory is allocated from. On x86 this is all memory under 16MB.
- `ZONE_NORMAL`: This is where normally allocated kernel memory goes. Where

[1]This is not quite true, there are kernels for processors without memory management units, but these are very specialised and won't be considered further

this zone ends depends on the architecture. However, all memory in this zone is mapped in kernel space (visible to the kernel).

- `ZONE_HIGHMEM`: This is where the rest of the memory goes. Its characteristic is that it is not mapped in kernel space (thus the kernel cannot access it without first mapping it).

1.1 The x86 and Highmem

The main reason for the existence of `ZONE_HIGHMEM` is a peculiar quirk on the x86 processor which makes it rather expensive to have different page table mappings between the kernel and user space. The root of the problem is that the x86 can only keep one set of physical to virtual mappings on-hand at once. Since the kernel and the processes occupy different virtual mappings, the TLB context would have to be switched not only when the processor changes current user tasks, but also when the current user task calls on the kernel to perform an operation on its behalf. The time taken to change mappings, called the TLB flushing penalty, contributes to a degradation in process performance and has been measured at around 30%[1]. To avoid this penalty, the Kernel and user spaces share a partitioned virtual address space so that the kernel is actually mapped into user space (although protected from user access) and vice versa.

The upshot of this is that the x86 userspace is divided 3GB/1GB with the virtual address range `0x00000000-0xbfffffff` being available for the user process and `0xc0000000-0xffffffff` being reserved for the kernel.

The problem, for the kernel, is that it now only has 1GB of virtual address to play with *including* all memory mapped I/O regions. The result being that `ZONE_NORMAL` actually ends at around 850kb on most x86 boxes. Since the kernel must also manage the mappings for every user process (and these mappings must be memory resident), the larger the physical memory of the kernel becomes, the less of `ZONE_NORMAL` becomes available to the kernel. On a 64GB x86 box, the usable memory becomes minuscule and has lead to the proposal[2] to use a 4G/4G split and just accept the TLB flushing penalty.

1.2 Non-x86 and Virtual Indexing

Most other architectures are rather better implemented and are able to cope easily with separate virtual spaces for the user and the kernel without imposing a performance penalty transitioning from one virtual address space to another. However, there are other problems the kernel's penchant for keeping all memory mapped causes, notably with Virtual Indexing.

Virtual Indexing[3] (VI) means that the CPU cache keeps its data indexed by virtual address (rather than by physical address like the x86 does). The problem this causes is that if multiple virtual address spaces have the same physical address mapped, but at different virtual addresses then the cache may contain duplicate entries, called aliases. Managing these aliases becomes impossible if there are multiple ones that become dirty.

Most VI architectures find a solution to the multiple cache line problem by having a "congruence modulus" meaning that if two virtual addresses are equal modulo this congruence (usually a value around 4MB) then the cache will detect the aliasing and keep only a single copy of the data that will be seen by all the virtual addresses.

The problems arise because, although architectures go to great lengths to make sure all user mappings are congruent, because the kernel memory is always mapped, it is highly unlikely that any given kernel page would be congruent to a user page.

1.3 The solution: Unmapping `ZONE_NORMAL`

It has already been pointed out[4] that x86 could recover some of its precious `ZONE_NORMAL` space simply by moving page table entries into unmapped highmem space. However, the penalty of having to map and unmap the page table entries to modify them turned out to be unacceptable.

The solution, though, remains valid. There are many pages of data currently in `ZONE_NORMAL` that the kernel doesn't ordinarily use. If these could be unmapped and their virtual address space given up then the x86 kernel wouldn't be facing quite such a memory crunch.

For VI architectures, the problems stem from having unallocated kernel memory already mapped. If we could keep the majority of kernel memory unmapped, and map it only when we really need to use it, then we would stand a very good chance of being able to map the memory congruently even in kernel space.

The solution this paper will explore is that of keeping the majority of kernel memory unmapped, mapping it only when it is used.

2 A closer look at Virtual Indexing

As well as the aliasing problem, VI architectures also have issues with I/O coherency on DMA. The essence of the problem stems from the fact that in order to make a device access to physical memory coherent, any cache lines that the processor is holding need to be flushed/invalidates as part of the DMA transaction. In order to do DMA, a device simply presents a physical address to the system with a request to read or write. However, if the processor indexes the caches virtually, it will have no idea whether it is caching this physical address or not. Therefore, in order to give the processor an idea of where in the cache the data might be, the DMA engines on VI architectures also present a virtual index (called the "coherence index") along with the physical address.

2.1 Coherence Indices and DMA

The Coherence Index is computed by the processor on a per page basis, and is used to identify the line in the cache belonging to the physical address the DMA is using.

One will notice that this means the coherence index must be computed on *every* DMA transaction for a *particular* address space (although, if all the addresses are congruent, one may simply pick any one). Since, at the time the dma mapping is done, the only virtual address the kernel knows about is the kernel virtual address, it means that DMA is always done coherently with the kernel.

In turn, since the kernel address is pretty much not congruent with any user address, before the DMA is signalled as being completed to the user process, the kernel mapping and the user mappings must likewise be made coherent (using the `flush_dcache_page()` function). However, since the majority of DMA transactions occur on *user* data in which the kernel has no interest, the extra flush is simply an unnecessary performance penalty.

This performance penalty would be eliminated if either we knew that the designated kernel address was congruent to all the user addresses or we didn't bother to map the DMA region into kernel space and simply computed the coherence index from a given user process. The latter would be preferable from a performance point of view since it eliminates an unnecessary map and unmap.

2.2 Other Issues with Non-Congruence

On the parisc architecture, there is an architectural requirement that we don't simultaneously enable multiple read and write translations of a non-congruent address. We can either enable a single write translation or multiple read (but no write) translations. With the current manner of kernel operation, this is almost impossible to satisfy without going to enormous lengths in our page translation and fault routines to work around the issues.

Previously, we were able to get away with ignoring this restriction because the machine would only detect it if we allowed multiple aliases to become dirty (something Linux never does). However, in the next generation systems, this condition will be detected when it occurs. Thus, addressing it has become critical to providing a bootable kernel on these new machines.

Thus, as well as being a simple performance enhancement, removing non-congruence becomes vital to keeping the kernel booting on next generation machines.

2.3 VIPT vs VIVT

This topic is covered comprehensively in [3]. However, there is a problem in VIPT caches, namely that if we are reusing the virtual address in kernel space, we must flush the processor's cache for that page on this re-use otherwise it may fall victim to stale cache references that were left over from a prior use.

Flushing a VIPT cache is easier said than done, since in order to flush, a valid translation must exist for the virtual address in order for the flush to be effective. This causes particular problems for pages that were mapped to a user space process, since the address translations are destroyed *before* the page is finally freed.

3 Kernel Virtual Space

Although the kernel is nominally mapped in the same way the user process is (and can theoretically be fragmented in physical space), in fact it is usually offset mapped. This means there is a simple mathematical relation between the physical and virtual addresses:

$$virtual = physical + \texttt{\_\_PAGE\_OFFSET}$$

where `__PAGE_OFFSET` is an architecture defined quantity. This type of mapping makes it very easy to calculate virtual addresses from physical ones and vice versa without having to go to all the bother (and CPU time) of having to look them up in the kernel page tables.

3.1 Moving away from Offset Mapping

There's another wrinkle on some architectures in that if an interruption occurs, the CPU turns off virtual addressing to begin processing it. This means that the kernel needs to save the various registers and turn virtual addressing back on, all in physical space. If it's no longer a simple matter of subtracting `__PAGE_OFFSET` to get the kernel stack for the process, then extra time will be consumed in the critical path doing potentially cache cold page table lookups.

3.2 Keeping track of Mapped pages

In general, when mapping a page we will either require that it goes in the first available slot (for x86), or that it goes at the first available slot congruent with a given address (for VI architectures). All we really require is a simple mechanism for finding the first free page virtual address given some specific constraints. However, since the constraints are architecture specific, the specifics of this tracking are also implemented in architectures (see section 5.2 for details on parisc).

3.3 Determining Physical address from Virtual and Vice-Versa

In the Linux kernel, the simple macros `__pa()` and `__va()` are used to do physical to virtual translation. Since we are now filling the mappings in randomly, this is no longer a simple offset calculation.

The kernel does have help for finding the virtual address of a given page. There is an optional `virtual` entry which is turned on and populated with the page's current virtual address when the architecture defines `WANT_PAGE_VIRTUAL`. The `__va()` macro can be programmed simply to do this lookup.

To find the physical address, the best method is probably to look the page up in the kernel page table mappings. This is obviously less efficient than a simple subtraction.

4 Implementing the unmapping of `ZONE_NORMAL`

It is not surprising, given that the entire kernel is designed to operate with `ZONE_NORMAL` mapped it is surprising that unmapping it turns out to be fairly easy. The primary reason for this is the existence of highmem. Since pages in `ZONE_HIGHMEM` are always unmapped and since they are usually assigned to user processes, the kernel must proceed on the assumption that it potentially has to map into its address space any page from a user process that it wishes to touch.

4.1 Booting

The kernel has an entire bootmem API whose sole job is to cope with memory allocations while the system is booting and before paging has been initialised to the point where normal memory allocations may proceed. On parisc, we simply get the available page ranges from the firmware, map them all and turn them over lock stock and barrel to bootmem.

Then, when we're ready to begin paging, we simply release all the unallocated bootmem pages for the kernel to use from its `mem_map`[2] array of pages.

We can implement the unmapping idea simply by covering all our page ranges with an offset map for bootmem, but then unmapping all the unreserved pages that bootmem releases to the `mem_map` array.

This leaves us with the kernel text and data sections contiguously offset mapped, and all other boot time

4.2 Pages Coming From User Space

The standard mechanisms for mapping potential highmem pages from user space for the kernel to see are `kmap`, `kunmap`, `kmap_atomic`, and `kmap_atomic_to_page`. Simply hijacking them and divorcing their implementation from `CONFIG_HIGHMEM` is sufficient to solve all user to kernel problems that arise because of the unmapping of `ZONE_NORMAL`.

4.3 In Kernel Problems: Memory Allocation

Since now every free page in the system will be unmapped, they will have to be mapped

[2]This global array would be a set of per-zone arrays on NUMA

before the *kernel* can use them (pages allocated for use in user space have no need to be mapped additionally in kernel space at allocation time). The engine for doing this is a single point in `__alloc_pages()` which is the central routine for allocating every page in the system. In the single successful page return, the page is mapped for the kernel to use it if `__GFP_HIGH` is not set—this simple test is sufficient to ensure that kernel pages only are mapped here.

The unmapping is done in two separate routines: `__free_pages_ok()` for freeing bulk pages (accumulations of contiguous pages) and `free_hot_cold_page()` for freeing single pages. Here, since we don't know the gfp mask the page was allocated with, we simply check to see if the page is currently mapped, and unmap it if it is before freeing it. There is another side benefit to this: the routine that transfers all the unreserved bootmem to the `mem_map` array does this via `__free_pages()`. Thus, we additionally achieve the unmapping of all the free pages in the system after booting with virtually no additional effort.

4.4 Other Benefits: Variable size pages

Although it wasn't the design of this structure to provide variable size pages, one of the benefits of this approach is now that the pages that are mapped as they are allocated. Since pages in the kernel are allocated with a specified order (the power of two of the number of contiguous pages), it becomes possible to cover them with a TLB entry that is larger than the usual page size (as long as the architecture supports this). Thus, we can take the `order` argument to `__alloc_pages()` and work out the smallest number of TLB entries that we need to allocate to cover it.

Implementation of variable size pages is actually transparent to the system; as far as Linux is concerned, the page table entries it deal with describe 4k pages. However, we add additional flags to the pte to tell the software TLB routine that actually we'd like to use a larger size TLB to access this region.

As a further optimisation, in the architecture specific routines that free the boot mem, we can remap the kernel text and data sections with the smallest number of TLB entries that will entirely cover each of them.

5 Achieving The VI architecture Goal: Fully Congruent Aliasing

The system possesses every attribute it now needs to implement this. We no-longer map any user pages into kernel space unless the kernel actually needs to touch them. Thus, the pages will have congruent user addresses allocated to them in user space *before* we try to map them in kernel space. Thus, all we have to do is track up the free address list in increments of the congruence modulus until we find an empty place to map the page congruently.

5.1 Wrinkles in the I/O Subsystem

The I/O subsystem is designed to operate without mapping pages into the kernel *at all*. This becomes problematic for VI architectures because we have to know the user virtual address to compute the coherence index for the I/O. If the page is unmapped in kernel space, we can no longer make it coherent with the kernel mapping and, unfortunately, the information in the BIO is insufficient to tell us the user virtual address.

The proposal for solving this is to add an architecture defined set of elements to `struct bio_vec` and an architecture specific function for populating this (possibly empty) set of elements as the biovec is created. In parisc, we need to add an extra unsigned long for the coherence index, which we compute from a pointer to the mm and the user virtual address. The architecture defined components are pulled into `struct scatterlist` by yet another callout when the request is mapped for DMA.

5.2 Tracking the Mappings in `ZONE_DMA`

Since the tracking requirements vary depending on architectures: x86 will merely wish to find the first free pte to place a page into; however VI architectures will need to find the first free pte satisfying the congruence requirements (which vary by architecture), the actual mechanism for finding a free pte for the mapping needs to be architecture specific.

On parisc, all of this can be done in `kmap_kernel()` which merely uses rmap[5] to determine if the page is mapped in user space and find the congruent address if it is. We use a simple hash table based bitmap with one bucket representing the set of available congruent pages. Thus, finding a page congruent to any given virtual address is the simple computation of finding the first set bit in the congruence bucket. To find an arbitrary page, we keep a global bucket counter, allocating a page from that bucket and then incrementing the counter[3].

6 Implementation Details on PA-RISC

Since the whole thrust of this project was to improve the kernel on PA-RISC (and bring it back into architectural compliance), it is appropriate to investigate some of the other problems that turned up during the implementation.

[3]This can all be done locklessly with atomic increments, since it doesn't really matter if we get two allocations from the same bucket because of race conditions

6.1 Equivalent Mapping

The PA architecture has a software TLB meaning that in Virtual mode, if the CPU accesses an address that isn't in the CPU's TLB cache, it will take a TLB fault so the software routine can locate the TLB entry (by walking the page tables) and insert it into the CPU's TLB. Obviously, this type of interruption must be handled purely by referencing physical addresses. In fact, the PA CPU is designed to have fast and slow paths for faults and interruptions. The fast paths (since they cannot take another interruption, i.e. not a TLB miss fault) must all operate on physical addresses. To assist with this, the PA CPU even turns off virtual addressing when it takes an interruption.

When the CPU turns off virtual address translation, it is said to be operating in absolute mode. All address accesses in this mode are physical. However, all accesses in this mode also go through the CPU cache (which means that for this particular mode the cache is actually Physically Indexed). Unfortunately, this can also set up unwanted aliasing between the physical address and its virtual translation. The fix for this is to obey the architectural definition for "equivalent mapping." Equivalent mapping is defined as virtual and physical addresses being equal; however, we benefit from the obvious loophole in that the physical and virtual addresses don't have to be exactly equal, merely equal modulo the congruent modulus.

All of this means that when a page is allocated for use by the kernel, we must determine if it will ever be used in absolute mode, and make it equivalently mapped if it will be. At the time of writing, this was simply implemented by making all kernel allocated pages equivalent. However, really all that needs to be equivalently mapped is

1. the page tables (pgd, pmd and pte),

2. the task structure and
3. the kernel stacks.

6.2 Physical to Virtual address Translation

In the interruption slow path, where we save all the registers and transition to virtual mode, there is a point where execution must be switched (and hence pointers moved from physical to virtual). Currently, with offset mapping, this is simply done by and addition of `__PAGE_OFFSET`. However, in the new scheme we cannot do this, nor can we call the address translation functions when in absolute mode. Therefore, we had to reorganise the interruption paths in the PA code so that both the physical and virtual address was available. Currently parisc uses a control register (`%cr30`) to store the virtual address of the `struct thread_info`. We altered all paths to change `%cr30` to contain the physical address of `struct thread_info` and also added a physical address pointer to the `struct task_struct` to the thread info. This is sufficient to perform all the necessary register saves in absolute addressing mode.

6.3 Flushing on Page Freeing

as was documented in section 2.3, we need to find a way of flushing a user virtual address *after* its translation is gone. Actually, this turns out to be quite easy on PARISC. We already have an area of memory (called the tmpalias space) that we use to copy to priming the user cache (it is simply a 4MB memory area we dynamically program to map to the page). Therefore, as long as we know the user virtual address, we can simply flush the page through the tmpalias space. In order to confound any attempted kernel use of this page, we reserve a separate 4MB virtual area that produces a page fault if referenced, and point the page's `virtual` address into this when it is *removed* from process mappings (so that any kernel attempt to use the page produces an immediate fault). Then, when the page is freed, if its `virtual` pointer is within this range, we convert it to a tmpalias address and flush it using the tmpalias mechanism.

7 Results and Conclusion

The best result is that on a parisc machine, the total amount of memory the operational kernel keeps mapped is around 10MB (although this alters depending on conditions).

The current implementation makes all pages congruent or equivalent, but the allocation routine contains `BUG_ON()` asserts to detect if we run out of equivalent addresses. So far, under fairly heavy stress, none of these has tripped.

Although the primary reason for the unmapping was to move parisc back within its architectural requirements, it also produces a knock on effect of speeding up I/O by eliminating the cache flushing from kernel to user space. At the time of writing, the effects of this were still unmeasured, but expected to be around 6% or so.

As a final side effect, the flush on free necessity releases the parisc from a very stringent "flush the entire cache on process death or exec" requirement that was producing horrible latencies in the parisc fork/exec. With this code in place, we see a vast (50%) improvement in the fork/exec figures.

References

[1] Andrea Arcangeli *3:1 4:4 100HZ 1000HZ comparison with the HINT benchmark* 7 April 2004 `http://www.kernel.org/pub/`

`linux/kernel/people/andrea/misc/31-44-100-1000/31-44-100-1000.html`

[2] Ingo Molnar *[announce, patch] 4G/4G split on x86, 64 GB RAM (and more) support* 8 July 2003 `http://marc.theaimsgroup.com/?t=105770467300001`

[3] James E.J. Bottomley *Understanding Caching* Linux Journal January 2004, Issue 117 p58

[4] Ingo Molnar *[patch] simpler 'highpte' design* 18 February 2002 `http://marc.theaimsgroup.com/?l=linux-kernel&m=101406121032371`

[5] Rik van Riel *Re: Rmap code?* 22 August 2001 `http://marc.theaimsgroup.com/?l=linux-mm&m=99849912207578`

Linux Virtualization on IBM POWER5 Systems

Dave Boutcher
IBM
boutcher@us.ibm.com

Dave Engebretsen
IBM
engebret@us.ibm.com

Abstract

In 2004 IBM® is releasing new systems based on the POWER5™ processor. There is new support in both the hardware and firmware for virtualization of multiple operating systems on a single platform. This includes the ability to have multiple operating systems share a processor. Additionally, a hypervisor firmware layer supports virtualization of I/O devices such as SCSI, LAN, and console, allowing limited physical resources in a system to be shared.

At its extreme, these new systems allow 10 Linux images per physical processor to run concurrently, contending for and sharing the system's physical resources. All changes to support these new functions are in the 2.4 and 2.6 Linux kernels.

This paper discusses the virtualization capabilities of the processor and firmware, as well as the changes made to the PPC64 kernel to take advantage of them.

1 Introduction

IBM's new POWER5** processor is being used in both IBM iSeries® and pSeries® systems capable of running any combination of Linux, AIX®, and OS/400® in logical partitions. The hardware and firmware, including a *hypervisor* [AAN00], in these systems provide the ability to create "virtual" system images with virtual hardware. The virtualization technique used on POWER™ hardware is known as paravirtualization, where the operating system is modified in select areas to make calls into the hypervisor. PPC64 Linux has been enhanced to make use of these virtualization interfaces. Note that the same PPC64 Linux kernel binary works on both virtualized systems and previous "bare metal" pSeries systems that did not offer a hypervisor.

All changes related to virtualization have been made in the kernel, and almost exclusively in the PPC64 portion of the code. One challenge has been keeping as much code common as possible between POWER5 portions of the code and other portions, such as those supporting the Apple G5.

Like previous generations of POWER processors such as the RS64 and POWER4™ families, POWER5 includes hardware enablement for logical partitioning. This includes features such as a hypervisor state which is more privileged than supervisor state. This higher privilege state is used to restrict access to system resources, such as the hardware page table, to hypervisor only access. All current systems based on POWER5 run in a hypervised environment, even if only one partition is active on the system.

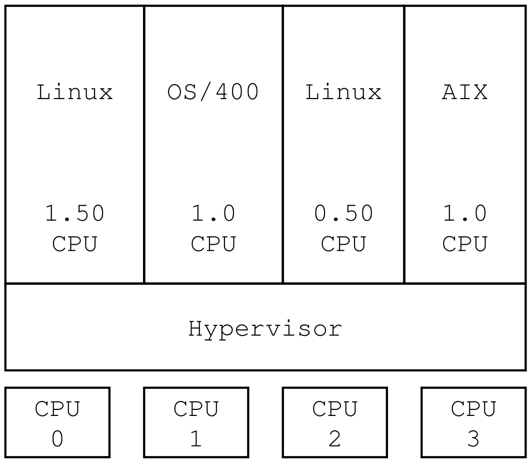


Figure 1: POWER5 Partitioned System

2 Processor Virtualization

2.1 Virtual Processors

When running in a partition, the operating system is allocated virtual processors (VP's), where each VP can be configured in either shared or dedicated mode of operation. In shared mode, as little as 10%, or 10 *processing units*, of a physical processor can be allocated to a partition and the hypervisor layer timeslices between the partitions. In dedicated mode, 100% of the processor is given to the partition such that its capacity is never multiplexed with another partition.

It is possible to create more virtual processors in the partition than there are physical processors on the system. For example, a partition allocated 100 processing units (the equivalent of 1 processor) of capacity could be configured to have 10 virtual processors, where each VP has 10% of a physical processor's time. While not generally valuable, this extreme configuration can be used to help test SMP configurations on small systems.

On POWER5 systems with multiple logical partitions, an important requirement is to be able to move processors (either shared or dedicated) from one logical partition to another. In the case of dedicated processors, this truly means moving a CPU from one logical partition to another. In the case of shared processors, it means adjusting the number of processors used by Linux on the fly.

This "hotplug CPU" capability is far more interesting in this environment than in the case that the covers are going to be removed from a real system and a CPU physically added. The goal of virtualization on these systems is to dynamically create and adjust operating system images as required. Much work has been done, particularly by Rusty Russell, to get the architecture independent changes into the mainline kernel to support hotplug CPU.

Hypervisor interfaces exist that help the operating system optimize its use of the physical processor resources. The following sections describe some of these mechanisms.

2.2 Virtual Processor Area

Each virtual processor in the partition can create a *virtual processor area* (VPA), which is a small (one page) data structure shared between the hypervisor and the operating system. Its primary use is to communicate information between the two software layers. Examples of the information that can be communicated in the VPA include whether the OS is in the idle loop, if floating point and performance counter register state must be saved by the hypervisor between operating system dispatches, and whether the VP is running in the partition's operating system.

2.3 Spinlocks

The hypervisor provides an interface that helps minimize wasted cycles in the operating system when a lock is held. Rather than simply spin on the held lock in the OS, a new hypervisor call, `h_confer`, has been provided. This interface is used to confer any remaining virtual processor cycles from the lock requester to the lock holder.

The PPC64 spinlocks were changed to identify the logical processor number of the lock holder, examine that processor's VPA *yield count* field to determine if it is not running in the OS (even values indicate the VP is running in the OS), and to make the `h_confer` call to the hypervisor to give any cycles remaining in the virtual processor's timeslice to the lock holder. Obviously, this more expensive leg of spinlock processing is only taken if the spinlock cannot be immediately acquired. In cases where the lock is available, no additional pathlength is incurred.

2.4 Idle

When the operating system no longer has active tasks to run and enters its idle loop, the `h_cede` interface is used to indicate to the hypervisor that the processor is available for other work. The operating system simply sets the VPA *idle* bit and calls `h_cede`. Under this call, the hypervisor is free to allocate the processor resources to another partition, or even to another virtual processor within the same partition. The processor is returned to the operating system if an external, decrementer (timer), or interprocessor interrupt occurs. As an alternative to sending an IPI, the ceded processor can be awoken by another processor calling the *h_prod* interface, which has slightly less overhead in this environment.

Making use of the cede interface is especially important on systems where partitions configured to run *uncapped* exist. In uncapped mode, any physical processor cycles not used by other partitions can be allocated by the hypervisor to a non-idle partition, even if that partition has already consumed its defined quantity of processor units. For example, a partition that is defined as uncapped, 2 virtual processors, and 20 processing units could consume 2 full processors (200 processing units), if all other partitions are idle.

2.5 SMT

The POWER5 processor provides symmetric multithreading (SMT) capabilities that allow two threads of execution to simultaneously execute on one physical processor. This results in twice as many processor contexts being presented to the operating system as there are physical processors. Like other processor threading mechanisms found in POWER RS64 and Intel® processors, the goal of SMT is to enable higher processor utilization.

At Linux boot, each processor thread is discovered in the open firmware device tree and a logical processor is created for Linux. A command line option, `smt-enabled = [on, off, dynamic]`, has been added to allow the Linux partition to config SMT in one of three states. The *on* and *off* modes indicate that the processor always runs with SMT either on or off. The dynamic mode allows the operating system and firmware to dynamically configure the processor to switch between threaded (SMT) and a single threaded (ST) mode where one of the processor threads is dormant. The hardware implementation is such that running in ST mode can provide additional performance when only a single task is executing.

Linux can cause the processor to switch between SMT and ST modes via the `h_cede` hypervisor call interface. When entering its idle loop, Linux sets the VPA *idle* state bit, and after a selectable delay, calls `h_cede`. Under this interface, the hypervisor layer determines if only one thread is idle, and if so, switches the processor into ST mode. If both threads are idle (as indicated by the VPA *idle* bit), then the hypervisor keeps the processor in SMT mode and returns to the operating system.

The processor switches back to SMT mode if an external or decrementer interrupt is presented, or if another processor calls the `h_prod` interface against the dormant thread.

3 Memory Virtualization

Memory is virtualized only to the extent that all partitions on the system are presented a contiguous range of logical addresses that start at zero. Linux sees these logical addresses as its real storage. The actual real memory is allocated by the hypervisor from any available space throughout the system, managing the storage in *logical memory blocks* (LMB's). Each LMB is presented to the partition via a memory node in the open firmware device tree. When Linux creates a mapping in the hardware page table for effective addresses, it makes a call to the hypervisor (`h_enter`) indicating the effective and partition logical address. The hypervisor translates the logical address to the corresponding real address and inserts the mapping into the hardware page table.

One additional layer of memory virtualization managed by the hypervisor is a *real mode offset* (RMO) region. This is a 128 or 256 MB region of memory covering the first portion of the logical address space within a partition. It can be accessed by Linux when address relocation is off, for example after an exception occurs. When a partition is running relocation off and accesses addresses within the RMO region, a simple offset is added by the hardware to generate the actual storage access. In this manner, each partition has what it considers logical address zero.

4 I/O Virtualization

Once CPU and memory have been virtualized, a key requirement is to provide virtualized I/O. The goal of the POWER5 systems is to have, for example, 10 Linux images running on a small system with a single CPU, 1GB of memory, and a single SCSI adapter and Ethernet adapter.

The approach taken to virtualize I/O is a cooperative implementation between the hypervisor and the operating system images. One operating system image always "owns" physical adapters and manages all I/O to those adapters (DMA, interrupts, etc.)

The hypervisor and Open Firmware then provide "virtual" adapters to any operating systems that require them. Creation of virtual adapters is done by the system administrator as part of logically partitioning the system. A key concept is that these virtual adapters do not interact in any way with the physical adapters. The virtual adapters interact with other operating systems in other logical partitions, which may choose to make use of physical adapters.

Virtual adapters are presented to the operating system in the Open Firmware device tree just as physical adapters are. They have very similar attributes as physical adapters, including DMA windows and interrupts.

The adapters currently supported by the hypervisor are virtual SCSI adapters, virtual Ethernet adapters, and virtual TTY adapters.

4.1 Virtual Bus

Virtual adapters, of course, exist on a virtual bus. The bus has slots into which virtual adapters are configured. The number of slots available on the virtual bus is configured by the system administrator. The goal is to make the behavior of virtual adapters consistent with physical adapters. The virtual bus is *not* presented as a PCI bus, but rather as its own bus type.

4.2 Virtual LAN

Virtual LAN adapters are conceptually the simplest kind of virtual adapter. The hypervisor implements a switch, which supports 802.1Q semantics for having multiple VLANs share a physical switch. Adapters can be marked as 802.1Q aware, in which case the hypervisor expects the operating system to handle the 802.1Q VLAN headers, or 802.1Q unaware, in which case the hypervisor connects the adapter to a single VLAN. Multiple virtual Ethernet adapters can be created for a given partition.

Virtual Ethernet adapters have an additional attribute called “Trunk Adapter.” An adapter marked as a Trunk Adapter will be delivered all frames that don’t match any MAC address on the virtual Ethernet. This is similar, but not identical, to promiscuous mode on a real adapter.

For a logical partition to have network connectivity to the outside world, the partition owning a “real” network adapter generally has both the real Ethernet adapter and a virtual Ethernet adapter marked as a Trunk adapter. That partition then performs either routing or bridging between the real adapter and the virtual adapter. The Linux bridge-utils package works well to bridge the two kinds of networks.

Note that there is no architected link between the real and virtual adapters, it is the responsibility of some operating system to route traffic between them.

The implementation of the virtual Ethernet adapters involves a number of hypervisor interfaces. Some of the more significant interfaces are `h_register_logical_lan` to establish the initial link between a device driver and a virtual Ethernet device, `h_send_logical_lan` to send a frame, and `h_add_logical_lan_buffer` to tell the hypervisor about a data buffer into which a received frame is to be placed. The hypervisor interfaces then support either polled or interrupt driven notification of new frames arriving.

For additional information on the virtual Ethernet implementation, the code is the documentation (`drivers/net/ibmveth.c`).

4.3 Virtual SCSI

Unlike virtual Ethernet adapters, virtual SCSI adapters come in two flavors. A “client” virtual SCSI adapter behaves just as a regular SCSI host bus adapter and is implemented within the SCSI framework of the Linux kernel. The SCSI mid-layer issues standard SCSI commands such as Inquiry to determine devices connected to the adapter, and issues regular SCSI operations to those devices.

A “server” virtual SCSI adapter, generally in a different partition than the client, receives all the SCSI commands from the client and is responsible for handling them. The hypervisor is not involved in what the server does with the commands. There is no requirement for the server to link a virtual SCSI adapter to any kind of real adapter. The server can process and return SCSI responses in any fashion it likes. If it happens to issue I/O operations to a real adapter as part of satisfying those requests, that is an implementation detail of the operating system containing the server adapter.

The hypervisor provides two very primitive interpartition communication mechanisms on which the virtual SCSI implementation is built. There is a queue of 16 byte messages referred to as a “Command/Response Queue” (CRQ). Each partition provides the hypervisor with a page of memory where its receive queue resides, and a partition wishing to send a message to its partner's queue issues an `h_send_crq` hypervisor call. When a message is received on the queue, an interrupt is (optionally) generated in the receiving partition.

The second hypervisor mechanism is a facility for issuing DMA operations between partitions. The `h_copy_rdma` call is used to DMA a block of memory from the memory space of one logical partition to the memory space of another.

The virtual SCSI interpartition protocol is implemented using the ANSI "SCSI RDMA Protocol" (SRP) (available at `http://www.t10.org`). When the client wishes to issue a SCSI operation, it builds an SRP frame, and sends the address of the frame in a 16 byte CRQ message. The server DMA's the SRP frame from the client, and processes it. The SRP frame may itself contain DMA addresses required for data transfer (read or write buffers, for example) which may require additional interpartition DMA operations. When the operation is complete, the server DMA's the SRP response back to the same location as the SRP command came from and sends a 16 byte CRQ message back indicating that the SCSI command has completed.

The current Linux virtual SCSI server decodes incoming SCSI commands and issues block layer commands (`generic_make_request`). This allows the SCSI server to share any block device (e.g., `/dev/sdb6` or `/dev/loop0`) with client partitions as a virtual SCSI device.

Note that consideration was given to using protocols such as iSCSI for device sharing between partitions. The virtual SCSI SRP design above, however, is a much simpler design that does not rely on riding above an existing IP stack. Additionally, the ability to use DMA operations between partitions fits much better into the SRP model than an iSCSI model.

The Linux virtual SCSI client (`drivers/scsi/ibmvscsi/ibmvscsi.c`) is close, at the time of writing, to being accepted into the Linux mainline. The Linux virtual SCSI server is sufficiently unlike existing SCSI drivers that it will require much more mailing list "discussion."

4.4 Virtual TTY

In addition to virtual Ethernet and SCSI adapters, the hypervisor supports virtual serial (TTY) adapters. As with SCSI adapter, these can be configured as "client" adapters, and "server" adapters and connected between partitions. The first virtual TTY adapter is used as the system console, and is treated specially by the hypervisor. It is automatically connected to the partition console on the Hardware Management Console.

To date, multiple concurrent "consoles" have not been implemented, but they could be. Similarly, this interface could be used for kernel debugging as with any serial port, but such an implementation has not been done.

5 Dynamic Resource Movement

As mentioned for processors, the logical partition environment lends itself to moving resources (processors, memory, I/O) between partitions. In a perfect world, such movement should be done dynamically while the operating system is running. Dynamic movement of processors is currently being implemented, and dynamic movement of I/O devices (including dynamically adding and removing virtual I/O devices) is included in the kernel mainline.

The one area for future work in Linux is the dynamic movement of memory into and out of an active partition. This function is already supported on other POWER5 operating systems, so there is an opportunity for Linux to catch up.

6 Multiple Operating Systems

A key feature of the POWER5 systems is the ability to run different operating systems in different logical partitions on the same physical system. The operating systems currently supported on the POWER5 hardware are AIX, OS/400, and Linux.

While running multiple operating systems, all of the functions for interpartion interaction described above must work between operating systems. For example, idle cycles from an AIX partition can be given to Linux. A processor can be moved from OS/400 to Linux while both operating systems are active.

For I/O, multiple operating systems must be able to communicate over the virtual Ethernet, and SCSI devices must be sharable from (say) an AIX virtual SCSI server to a Linux virtual SCSI client.

These requirements, along with the architected hypervisor interfaces, limit the ability to change implementations just to fit a Linux kernel internal behavior.

7 Conclusions

While many of the basic virtualization technologies described in this paper existed in the Linux implementation provided on POWER RS64 and POWER4 iSeries systems [Bou01], they have been significantly enhanced for POWER5 to better use the firmware provided interfaces.

The introduction of POWER5-based systems converged all of the virtualization interfaces provided by firmware on legacy iSeries and pSeries systems to a model in line with the legacy pSeries partitioned system architecture. As a result much of the PPC64 Linux virtualization code was updated to use these new virtualization interface definitions.

8 Acknowledgments

The authors would like to thank the entire Linux/PPC64 team for the work that went into the POWER5 virtualization effort. In particular Anton Blanchard, Paul Mackerras, Rusty Rusell, Hollis Blanchard, Santiago Leon, Ryan Arnold, Will Schmidt, Colin Devilbiss, Kyle Lucke, Mike Corrigan, Jeff Scheel, and David Larson.

9 Legal Statement

References

[AAN00] Bill Armstrong, Troy Armstrong, Naresh Nayar, Ron Peterson, Tom Sand, and Jeff Scheel. *Logical Partitioning*, `http://www-1.ibm.com/servers/eserver/iseries/beyondtech/lpar.htm`.

[Bou01] David Boutcher, *The iSeries Linux Kernel* 2001 Linux Symposium, (July 2001).

The State of ACPI in the Linux Kernel

A. Leonard Brown
Intel
`len.brown@intel.com`

Abstract

ACPI puts Linux in control of configuration and power management. It abstracts the platform BIOS and hardware so Linux and the platform can interoperate while evolving independently.

This paper starts with some background on the ACPI specification, followed by the state of ACPI deployment on Linux.

It describes the implementation architecture of ACPI on Linux, followed by details on the configuration and power management features.

It closes with a summary of ACPI bugzilla activity, and a list of what is next for ACPI in Linux.

1 ACPI Specification Background

> "ACPI (Advanced Configuration and Power Interface) is an open industry specification co-developed by Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba.
>
> ACPI establishes industry-standard interfaces for OS-directed configuration and power management on laptops, desktops, and servers.
>
> ACPI evolves the existing collection of power management BIOS code, Advanced Power Management (APM) application programming interfaces (APIs, PNPBIOS APIs, Multiprocessor Specification (MPS) tables and so on into a well-defined power management and configuration interface specification."[1]

ACPI 1.0 was published in 1996. 2.0 added 64-bit support in 2000. ACPI 3.0 is expected in summer 2004.

2 Linux ACPI Deployment

Linux supports ACPI on three architectures: `ia64`, `i386`, and `x86_64`.

2.1 `ia64` Linux/ACPI support

Most `ia64` platforms require ACPI support, as they do not have the legacy configuration methods seen on `i386`. All the Linux distributions that support `ia64` include ACPI support, whether they're based on Linux-2.4 or Linux-2.6.

2.2 `i386` Linux/ACPI support

Not all Linux-2.4 distributions enabled ACPI by default on `i386`. Often they used just enough table parsing to enable Hyper-Threading (HT), ala `acpi=ht` below, and relied on MPS and PIRQ routers to configure the

[1] `http://www.acpi.info`

```
setup_arch()
  dmi_scan_machine()
    Scan DMI blacklist
    BIOS Date vs Jan 1, 2001
  acpi_boot_init()
    acpi_table_init()
      locate and checksum all ACPI tables
      print table headers to console
    acpi_blacklisted()
      ACPI table headers vs. blacklist
    parse(BOOT) /* Simple Boot Flags */
    parse(FADT) /* PM timer address */
    parse(MADT) /* LAPIC, IOAPIC */
    parse(HPET) /* HiPrecision Timer */
    parse(MCFG) /* PCI Express base */
```

Figure 1: Early ACPI init on `i386`

machine. Some included ACPI support by default, but required the user to add `acpi=on` to the cmdline to enable it.

So far, the major Linux 2.6 distributions all support ACPI enabled by default on `i386`.

Several methods are used to make it more practical to deploy ACPI onto `i386` installed base. Figure 1 shows the early ACPI startup on the `i386` and where these methods hook in.

1. Most modern system BIOS support DMI, which exports the date of the BIOS. Linux DMI scan in `i386` disables ACPI on platforms with a BIOS older than January 1, 2001. There is nothing magic about this date, except it allowed developers to focus on recent platforms without getting distracted debugging issues on very old platforms that:

   (a) had been running Linux w/o ACPI support for years.

   (b) had virtually no chance of a BIOS update from the OEM.

   Boot parameter `acpi=force` is available to enable ACPI on platforms older than the cutoff date.

2. DMI also exports the hardware manufacturer, baseboard name, BIOS version, etc. that you can observe with `dmidecode`.[2] `dmi_scan.c` has a general purpose blacklist that keys off this information, and invokes various platform-specific workarounds. `acpi=off` is the most severe—disabling all ACPI support, even the simple table parsing needed to enable Hyper-Threading (HT). `acpi=ht` does the same, excepts parses enough tables to enable HT. `pci=noacpi` disables ACPI for PCI enumeration and interrupt configuration. And `acpi=noirq` disables ACPI just for interrupt configuration.

3. The ACPI tables also contain header information, which you see near the top of the kernel messages. ACPI maintains a blacklist based on the table headers. But this blacklist is somewhat primitive. When an entry matches the system, it either prints warnings or invokes `acpi=off`.

All three of these methods share the problem that if they are successful, they tend to hide root-cause issues in Linux that should be fixed. For this reason, adding to the blacklists is discouraged in the upstream kernel. Their main value is to allow Linux distributors to quickly react to deployment issues when they need to support deviant platforms.

2.3 `x86_64` Linux/ACPI support

All `x86_64` platforms I've seen include ACPI support. The major `x86_64` Linux distributions, whether Linux-2.4 or Linux-2.6 based, all support ACPI.

[2] `http://www.nongnu.org/dmidecode`

3 Implementation Overview

The ACPI specification describes platform registers, ACPI tables, and operation of the ACPI BIOS. Figure 2 shows these ACPI components logically as a layer above the platform specific hardware and firmware.

The ACPI kernel support centers around the ACPICA (ACPI Component Architecture[3]) core. ACPICA includes the AML[4] interpreter that implements ACPI's hardware abstraction. ACPICA also implements other OS-agnostic parts of the ACPI specification. The ACPICA code does not implement any policy, that is the realm of the Linux-specific code. A single file, `osl.c`, glues ACPICA to the Linux-specific functions it requires.

The box in Figure 2 labeled "Linux/ACPI" represents the Linux-specific ACPI code, including boot-time configuration.

Optional "ACPI drivers," such as Button, Battery, Processor, etc. are (optionally loadable) modules that implement policy related to those specific features and devices.

3.1 Events

ACPI registers for a "System Control Interrupt" (SCI) and all ACPI events come through that interrupt.

The kernel interrupt handler de-multiplexes the possible events using ACPI constructs. In some cases, it then delivers events to a user-space application such as `acpid` via `/proc/acpi/events`.

[3] `http://www.intel.com/technology/iapc/acpi`

[4] AML, ACPI Machine Language.

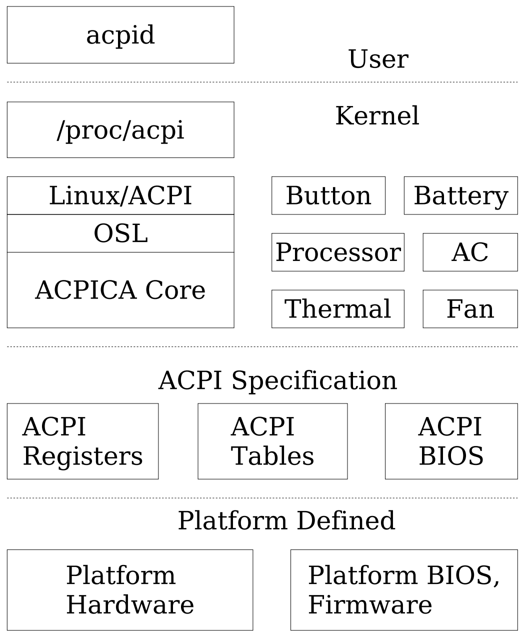


Figure 2: Implementation Architecture

4 ACPI Configuration

Interrupt configuration on `i386` dominated the ACPI bug fixing activity over the last year.

The algorithm to configure interrupts on an `i386` system with an IOAPIC is shown in Figure 3. ACPI mandates that all PIC mode IRQs be identity mapped to IOAPIC pins. Exceptions are specified in MADT [5] interrupt source override entries.

Over-rides are often used, for example, to specify that the 8254 timer on IRQ0 in PIC mode does not use pin0 on the IOAPIC, but uses pin2. Over-rides also often move the ACPI SCI to a different pin in IOAPIC mode than it had in PIC mode, or change its polarity or trigger from the default.

[5] MADT, Multiple APIC Description Table.

```
setup_arch()
 acpi_boot_init()
  parse(MADT);
   parse(LAPIC); /* processors */
   parse(IOAPIC)
    parse(INT_SRC_OVERRIDE);
    add_identity_legacy_mappings();
    /* mp_irqs[] initialized */

init()
 smp_boot_cpus()
  setup_IO_APIC()
   enable_IO_APIC();
   setup_IO_APIC_irqs(); /* mp_irqs[] */
 do_initcalls()
  acpi_init()
"ACPI: Subsystem revision 20040326"
   acpi_initialize_subsystem();
   /* AML interpreter */
   acpi_load_tables(); /* DSDT */
   acpi_enable_subsystem();
   /* HW into ACPI mode */
"ACPI: Interpreter enabled"
   acpi_bus_init_irq();
    AML(_PIC, PIC | IOAPIC | IOSAPIC);

  acpi_pci_link_init()
   for(every PCI Link in DSDT)
    acpi_pci_link_add(Link)
     AML(_PRS, Link);
     AML(_CRS, Link);
"... Link [LNKA] (IRQs 9 10 *11)"

  pci_acpi_init()
"PCI: Using ACPI for IRQ routing"
  acpi_irq_penalty_init();
  for (PCI devices)
   acpi_pci_irq_enable(device)
    acpi_pci_irq_lookup()
     find _PRT entry
      if (Link) {
       acpi_pci_link_get_irq()
        acpi_pci_link_allocate()
         examine possible & current IRQs
          AML(_SRS, Link)
      } else {
        use hard-coded IRQ in _PRT entry
      }
    acpi_register_gsi()
     mp_register_gsi()
      io_apic_set_pci_routing()
"PCI: PCI interrupt 00:06.0[A] ->
  GSI 26 (level, low) -> IRQ 26"
```

Figure 3: Interrupt Initialization

So after identifying that the system will be in IOAPIC mode, the 1st step is to record all the Interrupt Source Overrides in `mp_irqs[]`. The second step is to add the legacy identity mappings where pins and IRQs have not been consumed by the over-rides.

Step three is to digest `mp_irqs[]` in `setup_IO_APIC_irqs()`, just like it would be if the system were running in legacy MPS mode.

But that is just the start of interrupt configuration in ACPI mode. The system still needs to enable the mappings for PCI devices, which are stored in the DSDT[6] _PRT[7] entries. Further, the _PRT can contain both static entries, analogous to MPS table entries, or it can contain dynamic _PRT entries that use PCI Interrupt Link Devices.

So Linux enables the AML interpreter and informs the ACPI BIOS that it plans to run the system in IOAPIC mode.

Next the PCI Interrupt Link Devices are parsed. These "links" are abstract versions of what used to be called PIRQ-routers, though they are more general. `acpi_pci_link_init()` searches the DSDT for Link Devices and queries each about the IRQs it can be set to (_PRS)[8] and the IRQ that it is already set to (_CRS)[9]

A penalty table is used to help decide how to program the PCI Interrupt Link Devices. Weights are statically compiled into the table to avoid programming the links to well known legacy IRQs. `acpi_irq_penalty_init()` updates the table to add penalties to the IRQs where the Links have possible set-

[6]DSDT, Differentiated Services Description Table, written in AML

[7]_PRT, PCI Routing Table

[8]PRS, Possible Resource Settings.

[9]CRS, Current Resource Settings.

tings. The idea is to minimize IRQ sharing, while not conflicting with legacy IRQ use. While it works reasonably well in practice, this heuristic is inherently flawed because it assumes the legacy IRQs rather than asking the DSDT what legacy IRQs are actually in use.[10]

The PCI sub-system calls `acpi_pci_irq_enable()` for every device. ACPI looks up the device in the _PRT by device-id and if it a simple static entry, programs the IOAPIC. If it is a dynamic entry, `acpi_pci_link_allocate()` chooses an IRQ for the link and programs the link via AML (_SRS).[11] Then the associated IOAPIC entry is programmed.

Later, the drivers initialize and call `request_irq(IRQ)` with the IRQ the PCI sub-system told it to request.

One issue we have with this scheme is that it can't automatically recover when the heuristic balancing act fails. For example when the parallel port grabs IRQ7 and a PCI Interrupt Links gets programmed to the same IRQ, then `request_irq(IRQ)` correctly fails to put ISA and PCI interrupts on the same pin. But the system doesn't realize that one of the contenders could actually be re-programmed to a different IRQ.

The fix for this issue will be to delete the heuristic weights from the IRQ penalty table. Instead the kernel should scan the DSDT to enumerate exactly what legacy devices reserve exactly what IRQs.[12]

[10]In PIC mode, the default is to keep the BIOS provided current IRQ setting, unless cmdline `acpi_irq_balance` is used. Balancing is always enabled in IOAPIC mode.

[11]SRS, Set Resource Setting

[12]bugzilla 2733

4.1 Issues With PCI Interrupt Link Devices

Most of the issues have been with PCI Interrupt Link Devices, an ACPI mechanism primarily used to replace the chip-set-specific Legacy PIRQ code.

- The status (_STA) returned by a PCI Interrupt Link Device does not matter. Some systems mark the ones we should use as enabled, some do not.

- The status set by Linux on a link is important on some chip sets. If we do not explicitly disable some unused links, they result in tying together IRQs and can cause spurious interrupts.

- The current setting returned by a link (_CRS) can not always be trusted. Some systems return invalid settings always. Linux must assume that when it sets a link, the setting was successful.

- Some systems return a current setting that is outside the list of possible settings. Per above, this must be ignored and a new setting selected from the possible-list.

4.2 Issues With ACPI SCI Configuration

Another area that was ironed out this year was the ACPI SCI (System Control Interrupt). Originally, the SCI was always configured as level/low, but SCI failures didn't stop until we implemented the algorithm in Figure 4. During debugging, the kernel gained the cmdline option that applies to either PIC or IOAPIC mode: `acpi_sci={level,edge,high,low}` but production systems seem to be working properly and this has seen use recently only to work around prototype BIOS bugs.

```
if (PIC mode) {
  set ELCR to level trigger();
} else { /* IOAPIC mode */
  if (Interrupt Source Override) {
    Use IRQ specified in override
    if(trigger edge or level)
      use edge or level
    else (compatible trigger)
      use level

    if (polarity high or low)
      use high or low
    else
      use low
  } else { /* no Override */
    use level-trigger
    use low-polarity
  }
}
```

Figure 4: SCI configuration algorithm

4.3 Unresolved: Local APIC Timer Issue

The most troublesome configuration issue today is that many systems with no IO-APIC will hang during boot unless their LOCAL-APIC has been disabled, eg. by booting `nolapic`. While this issue has gone away on several systems with BIOS upgrades, entire product lines from high-volume OEMS appear to be subject to this failure. The current workaround to disable the LAPIC timer for the duration of the SMI-CMD update that enables ACPI mode. [13]

4.4 Wanted: Generic Linux Driver Manager

The ACPI DSDT enumerates motherboard devices via PNP identifiers. This method is used to load the ACPI specific devices today, eg. battery, button, fan, thermal etc. as well as `8550_acpi`. PCI devices are enumerated via PCI-ids from PCI config space. Legacy devices probe out using hard-coded address values.

But a device driver should not have to know or care how it is enumerated by its parent bus. An 8250 driver should worry about the 8250 and not if it is being discovered by legacy means, ACPI enumeration, or PCI.

[13]`http://bugzilla.kernel.org` 1269

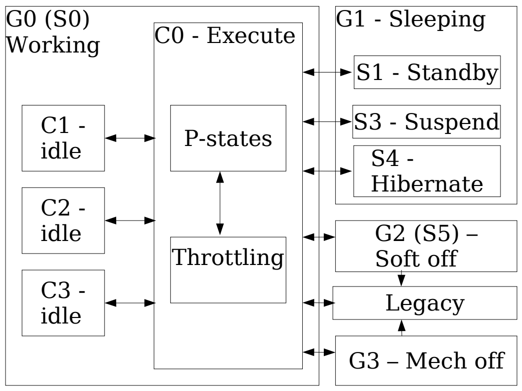


Figure 5: ACPI Global, CPU, and Sleep states.

One fix would be to be to abstract the PCI-ids, PNP-ids, and perhaps even some hard-coded values into a generic device manager directory that maps them to device drivers.

This would simply add a veneer to the PCI device configuration, simplifying a very small number of drivers that can be configured by PCI or ACPI. However, it would also fix the real issue that the configuration information in the ACPI DSDT for most motherboard devices is currently not parsed and not communicated to any Linux drivers.

The Device driver manager would also be able to tell the power management sub-system which methods are used to power-manage the device. Eg. PCI or ACPI.

5 ACPI Power Management

The Global System States defined by ACPI are illustrated in Figure 5. G0 is the working state, G1 is sleeping, G2 is soft-off and G3 is mechanical off. The “Legacy” state illustrates where the system is not in ACPI mode.

5.1 P-states

In the context of G0 – Global Working State, and C0 – CPU Executing State, P-states (Performance states) are available to reduce power of the running processor. P-states simultaneously modulate both the MHz and the voltage. As power varies by voltage squared, P-states are extremely effective at saving power.

While P-states are extremely important, the `cpufreq` sub-system handles P-states on a number of different platforms, and the topic is best addressed in that larger context.

5.2 Throttling

In the context of the G0-Working, C0-Executing state, Throttling states are defined to modulate the frequency of the running processor.

Power varies (almost) directly with MHz, so when the MHz is cut if half, so is the power. Unfortunately, so is the performance.

Linux currently uses Throttling only in response to thermal events where the processor is too hot. However, in the future, Linux could add throttling when the processor is already in the lowest P-state to save additional power.

Note that most processors also include a backup Thermal Monitor throttling mechanism in hardware, set with higher temperature thresholds than ACPI throttling. Most processors also have in hardware an thermal emergency shutdown mechanism.

5.3 C-states

In the context of G0 Working system state, C-state (CPU-state) C0 is used to refer to the executing state. Higher number C-states are entered to save successively more power when the processor is idle. No instructions are executed when in C1, C2, or C3.

ACPI replaces the default idle loop so it can enter C1, C2 or C3. The deeper the C-state, the more power savings, but the higher the latency to enter/exit the C-state. You can observe the C-states supported by the system and the success at using them in `/proc/acpi/processor/CPU0/power`

C1 is included in every processor and has negligible latency. C1 is implemented with the HALT or MONITOR/MWAIT instructions. Any interrupt will automatically wake the processor from C1.

C2 has higher latency (though always under 100 usec) and higher power savings than C1. It is entered through writes to ACPI registers and exits automatically with any interrupt.

C3 has higher latency (though always under 1000 usec) and higher power savings than C2. It is entered through writes to ACPI registers and exits automatically with any interrupt or bus master activity. The processor does not snoop its cache when in C3, which is why bus-master (DMA) activity will wake it up. Linux sees several implementation issues with C3 today:

1. C3 is enabled even if the latency is up to 1000 usec. This compares with the Linux 2.6 clock tick rate of 1000Hz = 1ms = 1000usec. So when a clock tick causes C3 to exit, it may take all the way to the next clock tick to execute the next kernel instruction. So the benefit of C3 is lost because the system effectively pays C3 latency and gets negligible C3 residency to save power.

2. Some devices do not tolerate the DMA latency introduced by C3. Their device buffers underrun or overflow. This is cur-

rently an issue with the ipw2100 WLAN NIC.

3. Some platforms can lie about C3 latency and transparently put the system into a higher latency C4 when we ask for C3—particularly when running on batteries.

4. Many processors halt their local APIC timer (a.k.a. TSC – Timer Stamp Counter) when in C3. You can observe this by watching LOC fall behind IRQ0 in /proc/interrupts.

5. USB makes it virtually impossible to enter C3 because of constant bus master activity. The workaround at the moment is to unplug your USB devices when idle. Longer term, it will take enhancements to the USB sub-system to address this issue. Ie. USB software needs to recognize when devices are present but idle, and reduce the frequency of bus master activity.

Linux decides which C-state to enter on idle based on a promotion/demotion algorithm. The current algorithm measures the residency in the current C-state. If it meets a threshold the processor is promoted to the deeper C-state on re-entrance into idle. If it was too short, then the processor is demoted to a lower-numbered C-state.

Unfortunately, the demotion rules are overly simplistic, as Linux tracks only its previous success at being idle, and doesn't yet account for the load on the system.

Support for deeper C-states via the _CST method is currently in prototype. Hopefully this method will also give the OS more accurate data than the FADT about the latency associated with C3. If it does not, then we may need to consider discarding the table-provided latencies and measuring the actual latency at boot time.

5.4 Sleep States

ACPI names sleeps states S0 – S5. S0 is the non-sleep state, synonymous with G0. S1 is standby, it halts the processor and turns off the display. Of course turning off the display on an idle system saves the same amount of power without taking the system off line, so S1 isn't worth much. S2 is deprecated. S3 is suspend to RAM. S4 is hibernate to disk. S5 is soft-power off, AKA G2.

Sleep states are unreliable enough on Linux today that they're best considered "experimental." Suspend/Resume suffers from (at least) two systematic problems:

- `_init()` and `_initdata()` on items that may be referenced after boot, say, during resume, is a bad idea.

- PCI configuration space is not uniformly saved and restored either for devices or for PCI bridges. This can be observed by using `lspci` before and after a suspend/resume cycle. Sometimes `setpci` can be used to repair this damage from user-space.

5.5 Device States

Not shown on the diagram, ACPI defines power saving states for devices: D0 – D3. D0 is on, D3 is off, D1 and D2 are intermediate. Higher device states have

1. more power savings,

2. less device context saved by hardware,

3. more device driver state restoring,

4. higher restore latency.

ACPI defines semantics for each device state in each device class. In practice, D1 and D2 are often optional - as many devices support only on and off either because they are low-latency, or because they are simple.

Linux-2.6 includes an updated device driver model to accommodate power management.[14] This model is highly compatible with PCI and ACPI. However, this vision is not yet fully realized. To do so, Linux needs a global power policy manager.

5.6 Wanted: Generic Linux Run-time Power Policy Manager

PCI device drivers today call `pci_set_power_state()` to enter D-states. This uses the power management capabilities in the PCI power management specification.

The ACPI DSDT supplies methods for ACPI enumerated devices to access ACPI D-states. However, no driver calls into ACPI to enter D-states today.[15]

Drivers shouldn't have to care if they are power managed by PCI or by ACPI. Drivers should be able to up-call to a generic run-time power policy manager. That manager should know about calling the PCI layer or the ACPI layer as appropriate.

The power manager should also put those requests in the context of user-specified power policy. Eg. Does the user want maximum performance, or maximum battery life? Currently there is no method to specify the detailed policy, and the kernel wouldn't know how to handle it anyway.

In a related point, it appears that devices currently only suspend upon system suspend. This is probably not the path to industry leading battery life.

Device drivers should recognize when their device has gone idle. They should invoke a suspend up-call to a power manager layer which will decide if it really is a good idea to grant that request now, and if so, how. In this case by calling the PCI or ACPI layer as appropriate.

6 ACPI as seen by bugzilla

Over the last year the ACPI developers have made heavy use of bugzilla[16] to help prioritize and track 460 bugs. 300 bugs are closed or resolved, 160 are open. [17]

We cc: `acpi-bugzilla@lists.sourceforge.net` on these bugs, and we encourage the community to add that alias to ACPI-specific bugs in other bugzillas so that the team can help out wherever the problems are found.

We haven't really used the bugzilla priority field. Instead we've split the bugs into categories and have addressed the configuration issues first. This explains why most of the interrupt bugs are resolved, and most of the suspend/resume bugs are unresolved.

We've seen an incoming bug rate of 10-bugs/week for many months, but the new reports favor the power management features over configuration, so we're hopeful that the torrent of configuration issues is behind us.

[14]Patrick Mochel, Linux Kernel Power Management, OLS 2003.

[15]Actually, the ACPI hot-plug driver invokes D-states, but that is the only exception.

[16]`http://bugzilla.kernel.org/`

[17]The resolved state indicates that a patch is available for testing, but that it is not yet checked into the kernel.org kernel.

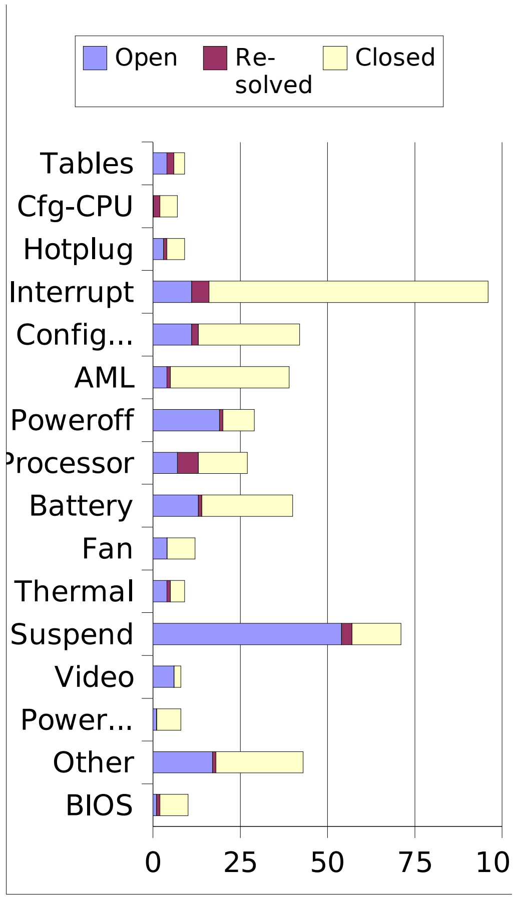


Figure 6: ACPI bug profile

7 Future Work

7.1 Linux 2.4

Going forward, I expect to back-port only critical configuration related fixes to Linux-2.4. For the latest power management code, users need to migrate to Linux-2.6.

7.2 Linux 2.6

Linux-2.6 is a "stable" release, so it is not appropriate to integrate significant new features. However, the power management side of ACPI is widely used in 2.6 and there will be plenty of bug-fixes necessary. The most visible will probably be anything that makes Suspend/Resume work on more platforms.

7.3 Linux 2.7

These feature gaps will not be addressed in Linux 2.6, and so are candidates for Linux 2.7:

- Device enumeration is not abstracted in a generic device driver manager that can shield drivers from knowing if they're enumerated by ACPI, PCI, or other.
- Motherboard devices enumerated by ACPI in the DSDT are ignored, and probed instead via legacy methods. This can lead to resource conflicts.
- Device power states are not abstracted in a generic device power manager that can shield drivers from knowing whether to call ACPI or PCI to handle D-states.
- There is no power policy manager to translate the user-requested power policy into kernel policy.
- No devices invoke ACPI methods to enter D-states.
- Devices do not detect that they are idle and request of a power manager whether they should enter power saving device states.
- There is no MP/SMT coordination of P-states. Today, P-states are disabled on SMP systems. Coordination needs to account for multiple threads and multiple cores per package.
- Coordinate P-states and T-states. Throttling should be used only after the system is put in the lowest P-state.
- Idle states above C1 are disabled on SMP.
- Enable Suspend in PAE mode. [18]

[18]PAE, Physical Address Extended—MMU mode to handle > 4GB RAM—optional on `i386`, always used on `x86_64`.

- Enable Suspend on SMP.
- Tick timer modulation for idle power savings.
- Video control extensions. Video is a large power consumer. The ACPI spec Video extensions are currently in prototype.
- Docking Station support is completely absent from Linux.
- ACPI 3.0 features. TBD after the specification is published.

7.4 ACPI 3.0

Although ACPI 3.0 has not yet been published, two ACPI 3.0 tidbits are already in Linux.

- PCI Express table scanning. This is the basic PCI Express support, there will be more coming. Those in the PCI SIG can read all about it in the PCI Express Firmware Specification.
- Several clarifications to the ACPI 2.0b spec resulted directly from open source development, [19] and the text of ACPI 3.0 has been updated accordingly. For example, some subtleties of SCI interrupt configuration and device enumeration.

When the ACPI 3.0 specification is published there will instantly be a multiple additions to the ACPI/Linux feature to-do list.

7.5 Tougher Issues

- Battery Life on Linux is not yet competitive. This single metric is the sum of all the power savings features in the platform, and if any of them are not working properly, it comes out on this bottom line.
- Laptop Hot Keys are used to control things such as video brightness, etc. ACPI does not specify Hot Keys. But when they work in APM mode and don't work in ACPI mode, ACPI gets blamed. There are 4 ways to implement hot keys:

    1. SMI[20] handler, the BIOS handles interrupts from the keys, and controls the device directly. This acts like "hardware" control as the OS doesn't know it is happening. But on many systems this SMI method is disabled as soon as the system transitions into ACPI mode. Thus the complaint "the button works in APM mode, but doesn't work in ACPI mode."

       But ACPI doesn't specify how hot keys work, so in ACPI mode one of the other methods listed here needs to handle the keys.
    2. Keyboard Extension driver, such as `i8k`. Here the keys return scan codes like any other keys on the keyboard, and the keyboard driver needs to understand those scan code. This is independent of ACPI, and generally OEM specific.
    3. OEM-specific ACPI hot key driver. Some OEMs enumerate the hot keys as OEM-specific devices in the ACPI tables. While the device is described in AML, such devices are not described in the ACPI spec so we can't build generic ACPI support for them. The OEM must supply the appropriate hot-key driver since only they know how it is supposed to work.
    4. Platform-specific "ACPI" driver. Today Linux includes Toshiba and

[19] FreeBSD deserves kudos in addition to Linux

[20] SMI, System Management Interrupt; invisible to the OS, handled by the BIOS, generally considered evil.

Asus platform specific extension drivers to ACPI. They do not use portable ACPI compliant methods to recognize and talk to the hot keys, but generally use the methods above.

The correct solution to the the Hot Key issue on Linux will require direct support from the OEMs, either by supplying documentation, or code to the community.

8 Summary

This past year has seen great strides in the configuration aspects of ACPI. Multiple Linux distributors now enable ACPI on multiple architectures.

This sets the foundation for the next era of ACPI on Linux where we can evolve the more advanced ACPI features to meet the expectations of the community.

9 Resources

The ACPI specification is published at `http://www.acpi.info`.

The home page for the Linux ACPI development community is here: `http://acpi.sourceforge.net/` It contains numerous useful pointers, including one to the `acpi-devel` mailing list.

The latest ACPI code can be found against various recent releases in the BitKeeper repositories: `http://linux-acpi.bkbits.net/`

Plain patches are available on `kernel.org`.[21] Note that Andrew Morton currently includes the latest ACPI test tree in the `-mm` patch, so you can test the latest ACPI code combined with other recent updates there.[22]

10 Acknowledgments

Many thanks to the following people whose direct contributions have significantly improved the quality of the ACPI code in the last year: Jesse Barnes, John Belmonte, Dominik Brodowski, Bruno Ducrot, Bjorn Helgaas, Nitin, Kamble, Andi Kleen, Karol Kozimor, Pavel Machek, Andrew Morton, Jun Nakajima, Venkatesh Pallipadi, Nate Lawson, David Shaohua Li, Suresh Siddha, Jes Sorensen, Andrew de Quincey, Arjan van de Ven, Matt Wilcox, and Luming Yu. Thanks also to all the bug submitters, and the enthusiasts on `acpi-devel`.

Special thanks to Intel's Mobile Platforms Group, which created ACPICA, particularly Bob Moore and Andy Grover.

[21]`http://ftp.kernel.org/pub/linux/kernel/people/lenb/acpi/patches/`

[22]`http://ftp.kernel.org/pub/linux/kernel/people/akpm/patches/`

Scaling Linux® to the Extreme

From 64 to 512 Processors

Ray Bryant
raybry@sgi.com
John Hawkes
hawkes@sgi.com
Jack Steiner
steiner@sgi.com
Jesse Barnes
jbarnes@sgi.com
Jeremy Higdon
jeremy@sgi.com

Silicon Graphics, Inc.

Abstract

In January 2003, SGI announced the SGI® Altix® 3000 family of servers. As announced, the SGI Altix 3000 system supported up to 64 Intel® Itanium® 2 processors and 512 GB of main memory in a single Linux® image. Altix now supports up to 256 processors in a single Linux system, and we have a few early-adopter customers who are running 512 processors in a single Linux system; others are running with as much as 4 terabytes of memory. This paper continues the work reported on in our 2003 OLS paper by describing the changes necessary to get Linux to efficiently run high-performance computing workloads on such large systems.

Introduction

At OLS 2003 [1], we discussed changes to Linux that allowed us to make Linux scale to 64 processors for our high-performance computing (HPC) workloads. Since then, we have continued our scalability work, and we now support up to 256 processors in a single Linux image, and we have a few early-adopter customers who are running 512 processors in a single-system image; other customers are running with as much as 4 terabytes of memory.

As can be imagined, the type of changes necessary to get a single Linux system to scale on a 512 processor system or to support 4 terabytes of memory are of a different nature than those necessary to get Linux to scale up to a 64 processor system, and the majority of this paper will describe such changes.

While much of this work has been done in the context of a Linux 2.4 kernel, Altix is now a supported platform in the Linux 2.6 series (`www.kernel.org` versions of Linux 2.6 boot and run well on many small to moderate sized Altix systems), and our plan is to port many of these changes to Linux 2.6 and propose them as enhancements to the community kernel. While some of these changes will be unique to the Linux kernel for Altix, many of the changes we propose will also improve performance on smaller SMP and NUMA systems, so should be of general interest to the Linux scalability community.

In the rest of this paper, we will first provide a brief review of the SGI Altix 3000 hardware. Next we will describe why we believe that very large single-system image, shared-memory machine can be more effective tools for HPC than similar sized non-shared memory clusters. We will then discuss changes that we made to Linux for Altix in order to make that system a more effective system for HPC on systems with as many as 512 processors. A second large topic of discussion will be the changes to support high-performance I/O on Altix and some of the hardware underpinnings for that support. We believe that the latter set of problems are general in the sense that they apply to any large scale NUMA system and the solutions we have adopted should be of general interest for this reason.

Even though this paper is focused on the changes that we have made to Linux to effectively support very large Altix platforms, it should be remembered that the total number of such changes is small in relation to the overall size of the Linux kernel and its supporting software. SGI is committed to supporting the Linux community and continues to support Linux for Altix as a member of the Linux family of kernels, and in general to support binary compatibility between Linux for Altix and Linux on other Itanium Processor Family platforms.

In many cases, the scaling changes described in this paper have already been submitted to the community for consideration for inclusion in Linux 2.6. In other cases, the changes are under evaluation to determine if they need to be added to Linux 2.6, or whether they are fixes for problems in Linux 2.4.21 (the current product base for Linux for Altix) that are no longer present in Linux 2.6.

Finally, this paper contains forward-looking statements regarding SGI® technologies and third-party technologies that are subject to risks and uncertainties. The reader is cautioned not to rely unduly on these forward-looking statements, which are not a guarantee of future or current performance, nor are they a guarantee that features described herein will or will not be available in future SGI products.

The SGI Altix Hardware

This section is condensed from [1]; the reader should refer to that paper for additional details.

An Altix system consists of a configurable number of rack-mounted units, each of which SGI refers to as a *brick*. The most common type of brick is the C-brick (or compute brick). A fully configured C-brick consists of two separate dual-processor Intel Itanium 2 systems, each of which is a bus-connected multiprocessor or *node*.

In addition to the two processors on the bus, there is also a SHUB chip on each bus. The SHUB is a proprietary ASIC that (1) acts as a memory controller for the local memory, (2) provides the interface to the interconnection network, (3) manages the global cache coherency protocol, and (4) some other functions as discussed in [1].

Memory accesses in an Altix system are either local (i.e., the reference is to memory in the same node as the processor) or remote. The SHUB detects whether a reference is local, in which case it directs the request to the memory on the node, or remote, in which case it forwards the request across the interconnection network to the SHUB chip where the memory reference will be serviced.

Local memory references have lower latency; the Altix system is thus a NUMA (non-uniform memory access) system. The ratio of remote to local memory access times on an Altix system varies from 1.9 to 3.5, depending on the size of the system and the relative locations of the processor and memory module involved in the transfer.

The cache-coherency policy in the Altix system can be divided into two levels: *local* and *global*. The local cache-coherency protocol is defined by the processors on the local bus and is used to maintain cache-coherency between the Itanium processors on the bus. The global cache-coherency protocol is implemented by the SHUB chip. The global protocol is directory-based and is a refinement of the protocol originally developed for DASH [2].

The Altix system interconnection network uses routing bricks to provide connectivity in system sizes larger than 16 processors. In systems with 128 or more processors a second layer of routing bricks is used to forward requests among subgroups of 32 processors each. The routing topology is a fat-tree topology with additional "express" links being inserted to improve performance.

Why Big SSI?

In this section we discuss the rationale for building such a large single-system image (SSI) box as an Altix system with 512 CPU's and (potentially) several TB of main memory:

(1) Shared memory systems are more flexible and easier to manage than a cluster. One can simulate message passing on shared memory, but not the other way around. Software for cluster management and system maintenance exists, but can be expensive or complex to use.

(2) Shared memory style programming is generally simpler and more easily understood than message passing. Debugging of code is often simpler on a SSI system than on a cluster.

(3) It is generally easier to port or write codes from scratch using the shared memory paradigm. Additionally it is often possible to simply ignore large sections of the code (e.g. those devoted to data input and output) and only parallelize the part that matters.

(4) A shared memory system supports easier load balancing within a computation. The mapping of grid points to a node determines the computational load on the node. Some grid points may be located near more rapidly changing parts of computation, resulting in higher computational load. Balancing this over time requires moving grid points from node to node in a cluster, where in a shared memory system such re-balancing is typically simpler.

(5) Access to large global data sets is simplified. Often, the parallel computation depends on a large data set describing, for example, the precise dimensions and characteristics of the physical object that is being modeled. This data set can be too large to fit into the node memories available on a clustered machine, but it can readily be loaded into memory on a large shared memory machine.

(6) Not everything fits into the cluster model. While many production codes have been converted to message passing, the overall computation may still contain one or more phases that are better performed using a large shared memory system. Or, there may be a subset of users of the system who would prefer a shared memory paradigm to a message passing one. This can be a particularly important consideration in large data-center environments.

Kernel Changes

In this section we describe the most significant kernel problems we have encountered in running Linux on a 512 processor Altix system.

Cache line and TLB Conflicts

Cache line conflicts occur in every cache-coherent multiprocessor system, to one extent or another, and whether or not the conflict exhibits itself as a performance problem is dependent on the rate at which the conflict occurs and the time required by the hardware to resolve the conflict. The latter time is typically proportional to the number of processors involved in the conflict. On Altix systems with 256 processors or more, we have encountered some cache line conflicts that can effectively halt forward progress of the machine. Typically, these conflicts involve global variables that are updated at each timer tick (or faster) by every processor in the system.

One example of this kind of problem is the default kernel profiler. When we first enabled the default kernel profiler on a 512 CPU system, the system would not boot. The reason was that once per timer tick, each processor in the system was trying to update the profiler bin corresponding to the CPU idle routine. A work around to this problem was to initialize `prof_cpu_mask` to `CPU_MASK_NONE` instead of the default. This disables profiling on all processors until the user sets the `prof_cpu_mask`.

Another example of this kind of problem was when we imported some timer code from Red Hat® AS 3.0. The timer code included a global variable that was used to account for differences between HZ (typically a power of 2) and the number of microseconds in a second (nominally 1,000,000). This global variable was updated by each processor on each timer tick. The result was that on Altix systems larger than about 384 processors, forward progress could not be made with this version of the code. To fix this problem, we made this global variable a per processor variable. The result was that the adjustment for the difference between HZ and microseconds is done on a per processor rather than on a global basis, and now the system will boot.

Still other cache line conflicts were remedied by identifying cases of false cache line sharing i.e., those cache lines that inadvertently contain a field that is frequently written by one CPU and another field (or fields) that are frequently read by other CPUs.

Another significant bottleneck is the ia64 `do_gettimeofday()` with its use of `cmpxchg`. That operation is expensive on most architectures, and concurrent `cmpxchg` operations on a common memory location scale worse than concurrent simple writes from multiple CPUs. On Altix, four concurrent user `gettimeofday()` system calls complete in almost an order of magnitude more time than a single `gettimeofday()`; eight are 20 times slower than one; and the scaling deteriorates nonlinearly to the point where 32 concurrent system calls is 100 times slower than one. At the present time, we are still exploring a way to improve this scaling problem in Linux 2.6 for Altix.

While moving data to per-processor storage is often a solution to the kind of scaling problems we have discussed here, it is not a panacea, particularly as the number of processors becomes large. Often, the system will want to inspect some data item in the per-processor storage of each processor in the system. For small numbers of processors this is not a problem. But when there are hundreds of processors involved, such loops can cause a TLB miss each time through the loop as well as a couple of cache-line misses, with the result that the loop may run quite slowly. (A TLB miss is caused because the per-processor storage areas are typically isolated from one another in the kernel's virtual address space.)

If such loops turn out to be bottlenecks, then what one must often do is to move the fields that such loops inspect out of the per-processor storage areas, and move them into a global static array with one entry per CPU.

An example of this kind of problem in Linux 2.6 for Altix is the current allocation scheme of the per-CPU run queue structures. Each per-CPU structure on an Altix system requires a unique TLB to address it, and each structure begins at the same virtual offset in a page, which for a virtually indexed cache means that the same fields will collide at the same index. Thus, a CPU scheduler that wishes to do a quick peek at every other CPU's `nr_running` or `cpu_load` will not only suffer a TLB miss on every access, but will also likely suffer a cache miss because these same virtual offsets will collide in the cache. Cache coloring of these addresses would be one way to solve this problem; we are still exploring ways to fix this problem in Linux 2.6 for Altix.

Lock Conflicts

A cousin of cache line conflicts are the lock conflicts. Indeed, the root mechanism of the lock bottleneck is a cache line conflict. For a `spinlock_t` the conflict is the `cmpxchg` operation on the word that signifies whether or not the lock is owned. For a `rwlock_t` the conflict is the cmpxchg or fetch-and-add operation on the count of the number of readers or the bit signifying whether or not the lock is owned exclusively by a writer. For a `seqlock_t` the conflict is the increment of the sequence number.

For some lock conflicts, such as the `rcu_ctrlblk.mutex`, the remedy is to make the spinlock more fine-grained, e.g., by making it hierarchical or per-CPU. For other lock conflicts, the most effective remedy is to reduce the use of the lock.

The O(1) CPU scheduler replaced the global `runqueue_lock` with per-CPU run queue locks, and replaced the global run queue with per-CPU run queues. While this did substantially decrease the CPU scheduling bottleneck for CPU counts in the 8 to 32 range, additional effort has been necessary to remedy additional bottlenecks that appear with even large configurations.

For example, we discovered that at 256 processors and above, we encountered a live lock early in system boot because hundreds of idle CPUs are load-balancing and are racing in contention on one or a few busy CPUs. The contention is so severe that the busy CPU's scheduler cannot itself acquire its own run queue lock, and thus the system live locks.

A remedy we applied in our Altix 2.4-based kernel was to introduce a progressively longer back off between successive load-balancing attempts, if the load-balancing CPU continues to be unsuccessful in finding a task to pull-migrate. Perhaps all the busiest CPU's tasks are pinned to that CPU, or perhaps all the tasks are still cache-hot. Regardless of the reason, a load-balancing failure results in that CPU delaying the next load-balance attempt by another incremental increase in time. This algorithm effectively solved the live lock, as well as improved other high-contention conflicts on a busiest CPU's run queue lock (e.g., always finding pinned tasks that can never be migrated).

This load-balance back off algorithm did not get accepted into the early 2.6 kernels. The latest 2.6.7 CPU scheduler, as developed by Nick Piggin, incorporates a similar back off algorithm. However, this algorithm (at least as it appears in 2.6.7-rc2) continues to cause a boot-time live lock at 512 processors on Altix so we are continuing to investigate this matter.

Page Cache

Managing the page cache in Altix has been a challenging problem. The reason is that while a large Altix system may have a lot of memory, each node in the system only has a relatively small fraction of that memory available as local memory. For example, on a 512 CPU system, if the entire system has 512 GB of memory, each node on the system has only 2 GB of local memory; less than 0.4% of the available memory on the system is local. When you consider that it is quite common on such systems to deal with files that are tens of GB in size, it is easy to understand how the page cache could consume all of the memory on several nodes in the system just doing normal, buffered-file I/O.

Stated another way, this is the challenge of a large NUMA system: all memory is addressable, but only a tiny fraction of that memory is local. Users of NUMA systems need to place their most frequently accessed data in local memory; this is crucial to obtain the maximum performance possible from the system. Typically this is done by allocating pages on a first-touch basis; that is, we attempt to allocate a page on the node where it is first referenced. If all of the local memory on a node is consumed by the page cache, then these local storage allocations will spill over to other (remote) nodes, the result being a potentially significant impact on program performance.

Similarly, it is important that the amount of free memory be balanced across idle nodes in the system. An imbalance could lead to some components of a parallel computation running slower than others because not all components of the computation were able to allocate their memory entirely out of local storage. Since the overall speed of parallel computation is determined by the execution of its slowest component, the performance of the entire application can be impacted by a non-local storage allocation on only a few nodes.

One might think that `bdflush` or `kupdated` (in a Linux 2.4 system) would be responsible for cleaning up unused page-cache pages. As the OLS reader knows, these daemons are responsible not for deallocating page-cache pages, but cleaning them. It is the swap daemon `kswapd` that is responsible for causing page-cache pages to be deallocated. However, in many situations we have encountered, even though multiple nodes of the system would be completely out of local memory, there would still be lots of free memory elsewhere in the system. As a result, `kswapd` will never start. Once the system gets into such a state, the local memory on those nodes can remain allocated entirely to page-cache pages for very long stretches of time since as far as the kernel is concerned there is no memory "pressure". To get around this problem, particularly for benchmarking studies, users have often resorted to programs that allocate and touch all of the memory on the system, thus causing `kswapd` to wake up and free unneeded buffer cache pages.

We have dealt with this problem in a number of ways, but the first approach was to change `page_cache_alloc()` so that instead of allocating the page on the local node, we spread allocations across all nodes in the system. To do this, we added a new GFP flag: `GFP_ROUND_ROBIN` and a new procedure: `alloc_pages_round_robin()`. `alloc_pages_round_robin()` maintains a counter in per-CPU storage; the counter is incremented on each call to `page_cache_alloc()`. The value of the counter, modulus the number of nodes in the system, is used to select the `zonelist` passed to `__alloc_pages()`. Like other NUMA implementations, in Linux for Altix there is a `zonelist` for each node, and the `zonelist`s are sorted in nearest neighbor order with the `zone` for the local node as the first entry of the `zonelist`. The result is that each time `page_cache_alloc()` is called, the returned page is allocated on the next node in sequence, or as close as possible to that node.

The rationale for allocating page-cache pages in this way is that while pages are local resources, the page cache is a global resource, usable by all processes on the system. Thus, even if a process is bound to a particular node, in general it does not make sense to allocate page-cache pages just on that node, since some other process in the system may be reading that same file and hence sharing the pages. So instead of flooding the current node with the page-cache pages for files that processes on that node have opened, we "tax" every node in the system with a fraction of the page-cache pages. In this way, we try to conserve a scarce resource (local memory) by spreading page-cache allocations over all nodes in the system.

However, even this step was not enough to keep local storage usage balanced among nodes in the system. After reading a 10 GB file, for example, we found that the node where the reading process was running would have up to 40,000 pages more storage allocated than other nodes in the system. It turned out the reason for this was that buffer heads for the read operation were being allocated locally. To solve this problem in our Linux 2.4.21 kernel for Altix, we modified `kmem_cache_grow()` so that it would pass the `GFP_ROUND_ROBIN` flag to `kmem_getpages()` with the result that the slab caches on our systems are now also allocated out of round-robin storage. Of course, this is not a perfect solution, since there are situations where it makes perfect sense to allocate a slab cache entry locally; but this was an expedient solution appropriate for our product. For Linux 2.6 for Altix we would like to see the slab allocator be made NUMA aware. (Manfred Spraul has created some patches to do this and we are currently evaluating these changes.)

The previous two changes solved many of the cases where a local storage could be exhausted by allocation of page-cache pages. However, they still did not solve the problem of local allocations spilling off node, particularly in those cases where storage allocation was tight across the entire system. In such situations, the system would often start running the synchronous swapping code even though most (if not all) of the page-cache pages on the system were clean and unreferenced outside of the page-cache. With the very-large memory sizes typical of our larger Altix customers, entering the synchronous swapping code needs to be avoided if at all possible since this tends to freeze the system for 10s of seconds. Additionally, the round robin allocation fixes did not solve the problem of poor and unrepeatable performance on benchmarks due to the existence of significant amounts of page-cache storage left over from previous executions.

To solve these problems, we introduced a routine called `toss_buffer_cache_pages_node()` (referred to here as `toss()`, for brevity). In a related change, we made the active and inactive lists per node rather than global. `toss()` first scans the inactive list (on a particular node) looking for idle page-cache pages to release back to the free page pool. If not enough such pages are found on the inactive list, then the active list is also scanned. Finally, if `toss()` has not called `shrink_slab_caches()` recently, that routine is also invoked in order to more aggressively free unused slab-cache entries. `toss()` was patterned after the main loop of `shrink_caches()` except that it would never call `swap_out()` and if it encountered a page that didn't look to be easily free able, it would just skip that page and go on to the next page.

A call to `toss()` was added in `__alloc_pages()` in such a way that if allocation on the current node fails, then before trying to allocate from some other node (i. e. spilling to another node), the system will first see if it can free enough page-cache pages from the current node so that the current node allocation can succeed. In subsequent allocation passes, `toss()` is also called to free page-cache pages on nodes other than the current one. The result of this change is that clean page-cache pages are effectively treated as free memory by the page allocator.

At the same time that the `toss()` code was added, we added a new user command `bcfree` that could be used to free all idle page-cache pages. (On the `__alloc_pages()` path, `toss()` would only try to free 32 pages per node.) The `bcfree` command was intended to be used only for resetting the state of the page cache before running a benchmark, and in lieu of rebooting the system in order to get a clean system state. However, our customers found that this command could be used to reduce the size of the page cache and to avoid situations where large amounts of buffered-file I/O could force the system to begin swapping. Since `bcfree` kills the entire page-cache, however, this was regarded as a substandard solution that could also hurt read performance of cached data and we began looking for another way to solve this "BIGIO" problem.

Just to be specific, the BIGIO problem we were trying to solve was based on the behavior of our Linux 2.4.21 kernel for Altix. A customer reported that on a 256 GB Altix system, if 200 GB were allocated and 50 GB free, that if the user program then tried to write 100 GB of data out to disk, the system would start to swap, and then in many cases fill up the swap space. At that point our Out-of-memory (OOM) killer would wake up and kill the user program! (See the next section for discussion of our OOM killer changes.)

Initially we were able to work around this problem by increasing the amount of swap space on the system. Our experiments showed that with an amount of swap space equal to one-quarter the main memory size, the 256 GB example discussed above would continue to completion without the OOM killer being invoked. I/O performance during this phase was typically one-half of what the hardware could deliver, since two I/O operations often had to be completed: one to read the data in from the swap device, and one to write the data to the output file. Additionally, while the swap scan was active, the system was very sluggish. These problems led us to search for another solution.

Eventually what we developed is an aggressive method of trimming the page cache when it started to grow too big. This solution involved several steps:

(1) We first added a new page list, the `reclaim_list`. This increased the size of `struct page` by another 16 bytes. On our system, `struct page` is allocated on cache-aligned boundaries anyway, so this really did not cause an increase in storage, since the current `struct page` size was less than 112 bytes. Pages were added to the reclaim list when they were inserted into the page cache. The reclaim list is per node, with per node locking. Pages were removed from the reclaim list when they were no longer reclaimable; that is, they were removed from the reclaim list when they were marked as dirty due to buffer file-I/O or when they were mapped into an address space.

(2) We rewrote `toss()` to scan the reclaim list instead of the inactive and active lists. Herein we will refer to the new version of `toss()` as `toss_fast()`.

(3) We introduced a variant of `page_cache_alloc()` called `page_cache_alloc_limited()`. Associated with this new routine were two control variables settable via `sysctl()`: `page_cache_limit` and `page_cache_limit_threshold`.

(4) We modified the `generic_file_write()` path to call `page_cache_alloc_limited()` instead of `page_cache_alloc()`. `page_cache_alloc_limited()` examines the size of the page cache. If the total amount of free memory in the system is less than `page_cache_limit_threshold` and the size of the page cache is larger than `page_cache_limit`, then `page_cache_alloc_limited()` calls `page_cache_reduce()` to free enough page-cache pages on the system to bring the page cache size down below `page_cache_limit`. If this succeeds, then `page_cache_alloc_limited()` calls `page_cache_alloc` to allocate the page. If not, then we wakeup `bdflush` and the current thread is put to sleep for 30ms (a tunable parameter)

The rationale for the `reclaim_list` and `toss_fast()` was that when we needed to trim the page cache, practically all pages in the system would typically be on the inactive list. The existing `toss()` routine scanned the inactive list and thus was too slow to call from `generic_file_write`. Moreover, most of the pages on the inactive list were not reclaimable anyway. Most of the pages on the `reclaim_list` are reclaimable. As a result `toss_fast()` runs much faster and is more efficient at releasing idle page-cache pages than the old routine.

The rationale for the `page_cache_limit_threshold` in addition to the `page_cache_limit` is that if there is lots of free memory then there is no reason to trim the page cache. One might think that because we only trim the page cache on the file write path that this approach would still let the page cache to grow arbitrarily due to file reads. Unfortunately, this is not the case, since the Linux kernel in normal multiuser operation is constantly writing something to the disk. So, a page cache limit enforced at file write time is also an effective limit on the size of the page cache due to file reads.

Finally, the rationale for delaying the calling task when `page_cache_reduce()` fails is that we do not want the system to start swapping to make space for new buffered I/O pages, since that will reduce I/O bandwidth by as much as one-half anyway, as well as take a lot of CPU time to figure out which pages to swap out. So it is better to reduce the I/O bandwidth directly, by limiting the rate of requested I/O, instead of allowing that I/O to proceed at rate that causes the system to be overrun by page-cache pages.

Thus far, we have had good experience with this algorithm. File I/O rates are not substantially reduced from what the hardware can provide, the system does not start swapping, and the system remains responsive and usable during the period of time when the BIGIO is running.

Of course, this entire discussion is specific to Linux 2.4.21. For Linux 2.6, we have plans to evaluate whether this is a problem in the system at all. In particular, we want to see if an appropriate setting for `vm_swappiness` to zero can eliminate the "BIGIO causes swapping" problem. We also are interested in evaluating the recent set of VM patches that Nick Piggin [6] has assembled to see if they eliminate this problem for systems of the size of a large Altix.

VM and Memory Allocation Fixes

In addition to the page-cache changes described in the last section, we have made a number of smaller changes related to virtual memory and paging performance.

One set of such changes increased the parallelism of page-fault handling for anonymous pages in multi-threaded applications. These applications allocate space using routines that eventually call `mmap()`; the result is that when the application touches the data area for the first time, it causes a minor page fault. These faults are serviced while holding the address space's `page_table_lock`. If the address space is large and there are a large number of threads executing in the address space, this spinlock can be an initialization-time bottleneck for the application. Examination of the `handle_mm_fault()` path for this case shows that the `page_table_lock` is acquired unconditionally but then released as soon as we have determined that this is a not-present fault for an anonymous page. So, we reordered the code checks in `handle_mm_fault()` to determine in advance whether or not this was the case we were in, and if so, to skip acquiring the lock altogether.

The second place the `page_table_lock` was used on this path was in `do_anonymous_page()`. Here, the lock was re-acquired to make sure that the process of allocating a page frame and filling in the pte is atomic. On Itanium, stores to page-table entries are normal stores (that is, the `set_pte` macro evaluates to a simple store). Thus, we can use `cmpxchg` to update the pte and make sure that only one thread allocates the page and fills in the pte. The compare and exchange effectively lets us lock on each individual pte. So, for Altix, we have been able to completely eliminate the `page_table_lock` from this particular page-fault path.

The performance improvement from this change is shown in Figure 1. Here we show the time required to initially touch 96 GB of data. As additional processors are added to the problem, the time required for both the baseline-Linux and Linux for Altix versions decrease until around 16 processors. At that point the `page_table_lock` starts to become a significant bottleneck. For the largest number of processors, even the time for the Linux for Altix case is starting to increase again. We believe that this is due to contention for the address space's `mmap` semaphore.

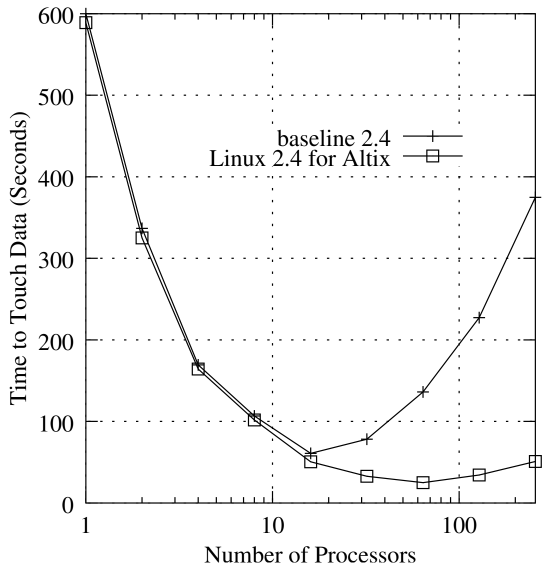


Figure 1: *Time to initially touch 96 GB of data.*

This is particularly important for HPC applications since OpenMP™[5], a common parallel programming model for FORTRAN, is implemented using a single address space, multiple-thread programming model. The optimization described here is one of the reasons that Altix has recently set new performance records for the SPEC® SPEComp® L2001 benchmark [7].

While the above measurements were taken using Linux 2.4.21 for Altix, a similar problem exists in Linux 2.6. For many other architectures, this same kind of change can be made; i386 is one of the exceptions to this statement. We are planning on porting our Linux 2.4.21 based changes to Linux 2.6 and submitting the changes to the Linux community for inclusion in Linux 2.6. This may require moving part of `do_anonymous_page()` to architecture dependent code to allow for the fact that not all architectures can use the compare and exchange approach to eliminate the use of the `page_table_lock` in `do_anonymous_page()`. However, the performance improvement shown in Figure 1 is significant for Altix so we would we would like to explore some way of incorporating this code into the mainline kernel.

We have encountered similar scalability limitations for other kinds of page-fault behavior. Figure 2 shows the number of page faults per second of wall clock time measured for multiple processes running simultaneously and faulting in a 1 GB `/dev/zero` mapping. Unlike the previous case described here, in this case each process has its own private mapping. (Here the number of processes is equal to the number of CPUs.) The dramatic difference between the baseline 2.4 and 2.6 cases and Linux for Altix is due to elimination of a lock in the super block for `/dev/zero`.

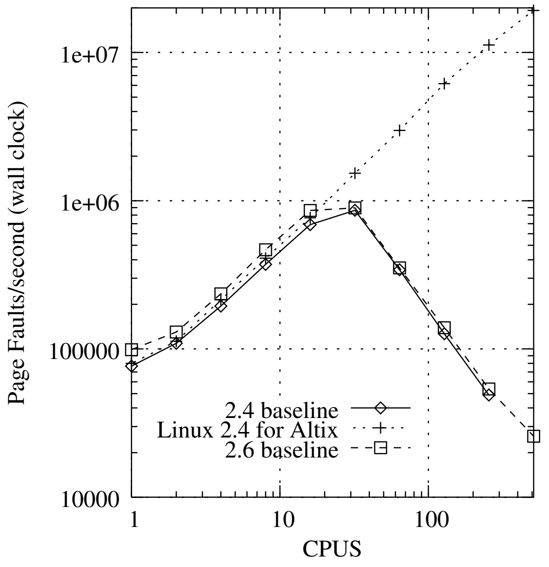


Figure 2: *Page Faults per Second of Wall Clock Time.*

The lock in the super block protects two counts: One count limits the maximum number of `/dev/zero` mappings to 2^{63}; the second count limits the number of pages assigned to a `/dev/zero` mapping to 2^{63}. Neither one of these counts is particularly useful for a `/dev/zero` mapping. We eliminated this lock and obtained a dramatic performance improvement for this micro-benchmark (at 512 CPUs the improvement was in excess of 800x). This optimization is important in decreasing startup time for large message-passing applications on the Altix system.

A related change is to distribute the count of pages in the page cache from a single global variable to a per node variable. Because every processor in the system needs to update the page-cache count when adding or removing pages from the page cache, contention for the cache line containing this global variable becomes significant. We changed this global count to a per-node count. When a page is inserted into (or removed from) the page cache, we update the page cache-count on the same node as the page itself. When we need the total number of pages in the page cache (for example if someone reads `/proc/meminfo`) we run a loop that sums the per node counts. However, since the latter operation is much less frequent than insertions and deletions from the page cache, this optimization is an overall performance improvement.

Another change we have made in the VM subsystem is in the out-of-memory (OOM) killer for Altix. In Linux 2.4.21, the OOM killer is called from the top of memory-free and swap-out call chain. `oom_kill()` is called from `try_to_free_pages_zone()` when calls to `shrink_caches()` at memory priority levels 6 through 0 have all failed. Inside `oom_kill()` a number of checks are performed, and if any of these checks succeed, the system is declared to not be out-of-memory. One of those checks is "if it has been more than 5 seconds since `oom_kill()` was last called, then we are not

OOM.” On a large-memory Altix system, it can easily take much longer than that to complete the necessary calls to `shrink_caches()`. The result is that an Altix system never goes OOM in spite of the fact that swap space is full and there is no memory to be allocated.

It seemed to us that part of the problem here is the amount of time it can take for a swap full condition (readily detectable in `try_to_swap_out()` to bubble all the way up to the top level in `try_to_free_pages_zone()`, especially on a large memory machine. To solve this problem on Altix, we decided to drive the OOM killer directly off of detection of swap-space-full condition provided that the system also continues to try to swap out additional pages. A count of the number of successful swaps and unsuccessful swap attempts is maintained in `try_to_swap_out()`. If, in a 10 second interval, the number of successful swap outs is less than one percent of the number of attempted swap outs, and the total number of swap out attempts exceeds a specified threshold, then `try_to_swap_out()`) will directly wake the OOM killer thread (also new in our implementation). This thread will wait another 10 seconds, and if the out-of-swap condition persists, it will invoke `oom_kill()` to select a victim and kill it. The OOM killer thread will repeat this sleep and kill cycle until it appears that swap space is no longer full or the number of attempts to swap out new pages (since the thread went to sleep) falls below the threshold.

In our experience, this has made invocation of the OOM killer much more reliable than it was before, at least on Altix. Once again, this implementation was for Linux 2.4.21; we are in the process of evaluating this problem and the associated fix on Linux 2.6 at the present time.

Another fix we have made to the VM system in Linux 2.4.21 for Altix is in handling of HUGETLB pages. The existing implementation in Linux 2.4.21 allocates HUGETLB pages to an address space at `mmap()` time (see `hugetlb_prefault()`); it also zeroes the pages at this time. This processing is done by the thread that makes the `mmap()` call. In particular, this means that zeroing of the allocated HUGETLB pages is done by a single processor. On a machine with 4 TB of memory and with as much memory allocated to HUGETLB pages as possible, our measurements have shown that it can take as long as 5,000 seconds to allocate and zero all available HUGETLB pages. Worse yet, the thread that does this operation holds the address space’s `mmap_sem` and the `page_table_lock` for the entire 5,000 seconds. Unfortunately, many commands that query system state (such as `ps` and `w`) also wish to acquire one of these locks. The result is that the system appears to be hung for the entire 5,000 seconds.

We solved this problem on Altix by changing the implementation of HUGETLB page allocation from *prefault* to *allocate on fault*. Many others have created similar patches; our patch was unique in that it also allowed zeroing of pages to occur in parallel if the HUGETLB page faults occurred on different processors. This was crucial to allow a large HUGETLB page region to be faulted into an address space in parallel, using as many processors as possible. For example, we have observed speedups of 25x using 16 processors to touch O(100 GB) of HUGETLB pages. (The speedup is super linear because if you use just one processor it has to zero many remote pages, whereas if you use more processors, at least some of the pages you are zeroing are local or on nearby nodes.) Assuming we can achieve the same kind of speedup on a 4 TB system, we would reduce the 5,000 second time stated above to 200 seconds.

Recently, we have worked with Kenneth Chen to get a similar set of changes proposed for Linux 2.6 [3]. Once this set of changes is accepted into the mainline this particular problem will be solved for Linux 2.6. These changes are also necessary for Andi Kleen's NUMA placement algorithms [4] to apply to HUGETLB pages, since otherwise pages are placed at `hugetlb_prefault()` time.

A final set of changes is related to large kernel tables. As previously mentioned, on an Altix system with 512 processors, less than 0.4% of the available memory is local. Certain tables in the Linux kernel are sized to be on the order of one percent of available memory. (An example of this is the TCP/IP hash table.) Allocating a table of this size can use all of the local memory on a node, resulting in exactly the kind of storage-allocation imbalance we developed the page-cache changes to solve. To avoid this problem, we also implement round-robin allocation of these large tables. Our current technique uses `vm_alloc()` to do this. Unfortunately, this is not portable across all architectures, since certain architectures have limited amounts of space that can be allocated by `vm_alloc()`. Nonetheless, this is a change that we need to make; we are still exploring ways of making this change acceptable to the Linux community.

Once we have solved the initial allocation problem for these tables, there is still the problem of getting them appropriately sized for an Altix system. Clearly if there are 4 TB of main memory, it does not make much sense to allocate a TCP/IP hash table of 40 GB, particularly since the TCP/IP traffic into an Altix system does not increase with memory size the way one might expect it to scale with a traditional Linux server. We have seen cases where system performance is significantly hampered due to lookups in these overly large tables. At the moment, we are still exploring a solution acceptable to the community to solve this particular problem.

I/O Changes for Altix

One of the design goals for the Altix system is that it support standard PCI devices and their associated Linux drivers as much as possible. In this section we discuss the performance improvements built into the Altix hardware and supported through new driver interfaces in Linux that help us to meet this goal with excellent performance even on very large Altix systems.

According to the PCI specification, DMA writes and PIO read responses are strongly ordered. On large NUMA systems, however, DMA writes can take a long time to complete. Since most PIO reads do not imply completion of a previous DMA write, relaxing the ordering rules of DMA writes and PIO read responses can greatly improve system performance.

Another large system issue relates to initiating PIO writes from multiple CPUs. PIO writes from two different CPUs may arrive out of order at a device. The usual way to ensure ordering is through a combination of locking and a PIO read (see Documentation/io_ordering.txt). On large systems, however, doing this read can be very expensive, particularly if it must be ordered with respect to unrelated DMA writes.

Finally, the NUMA nature of large machines make some optimizations obvious and desirable. Many devices use so-called consistent system memory for retrieving commands and storing status information; allocating that memory close to its associated device makes sense.

Making non–dependent PIO reads fast

In its I/O chipsets, SGI chose to relax the ordering between DMAs and PIOs, instead adding a barrier attribute to certain DMA writes (to consistent PCI allocations on Altix) and to interrupts. This works well with controllers that use DMA writes to indicate command completions (for example a SCSI controller with a response queue, where the response queue is allocated using `pci_alloc_consistent`, so that writes to the response queue have the barrier attribute). When we ported Linux to Altix, this behavior became a problem, because many Linux PCI drivers use PIO read responses to imply a status of a DMA write. For example, on an IDE controller, a bit status register read is performed to find out if a command is complete (command complete status implies that DMA writes of that command's data are completed). As a result, SGI had to implement a rather heavyweight mechanism to guarantee ordering of DMA writes and PIO reads. This mechanism involves doing an explicit flush of DMA write data after each PIO read.

For the cases in which strong ordering of PIO read responses and DMA writes are not necessary, a new API was needed so that drivers could communicate that a given PIO read response could used relaxed ordering with respect to prior DMA writes. The `read_relaxed` API [8] was added early in the 2.6 series for this purpose, and mirrors the normal read routines, which have variants for various sized reads.

The results below show how expensive a normal PIO read transaction can be, especially on a system doing a lot of I/O (and thus DMA).

Type of PIO	Time (ns)
normal PIO read	3875
relaxed PIO read	1299

Table 1: Normal vs. relaxed PIO reads on an idle system

It remains to be seen whether this API will also apply to the newly added RO bit in the PCI-X specification—the author is hopeful! Either way, it does give hardware vendors who want to support Linux some additional flexibility in their design.

Type of PIO	Time (ns)
normal PIO read	4889
relaxed PIO read	1646

Table 2: Normal vs. relaxed PIO reads on a busy system

Ordering posted writes efficiently

On many platforms, PIO writes from different CPUs will not necessarily arrive in order (i.e., they may be intermixed) even when locking is used. Since the platform has no way of knowing whether a given PIO read depends on preceding writes, it has to guarantee that all writes have completed before allowing a read transaction to complete. So performing a read prior to releasing a lock protecting a region doing writes is sufficient to guarantee that the writes arrive in the correct order.

However, performing PIO reads can be an expensive operation, especially if the device is on a distant node. SGI chipset designers foresaw this problem, however, and provided a way to ensure ordering by simply reading a register from the chipset on the local node. When the register indicates that all PIO writes are complete, it means they have arrived at the chipset attached to the device, and so are guaranteed to arrive at the device in the intended order. The SGI sn2 specific portion of the Linux ia64 port (sn2 is the architecture name for Altix in the Linux kernel source tree) provides a small function, `sn_mmiob()` (for memory–mapped I/O barrier, analogous to the `mb()` macro), to do just that. It can be used in place of reads that are intended to deal with posted writes and provides some benefit:

Type of flush	Time (ns)
regular PIO read	5940
relaxed PIO read	2619
`sn_mmiob()`	1610
(local chipset read alone)	399

Table 3: Normal vs. fast flushing of 5 PIO writes

Adding this API to Linux (i.e., making it non-sn2-specific) was discussed some time ago [9], and may need to be raised again, since it does appear to be useful on Altix, and is probably similarly useful on other platforms.

Local allocation of consistent DMA mappings

Consistent DMA mappings are used frequently by drivers to store command and status buffers. They are frequently read and written by the device that owns them, so making sure they can be accessed quickly is important. The table below shows the difference in the number of operations per second that can be achieved using local versus remote allocation of consistent DMA buffers. Local allocations were guaranteed by changing the `pci_alloc_consistent` function so that it calls `alloc_pages_node` using the node closest to the PCI device in question.

Type	I/Os per second
Local consistent buffer	46231
Remote consistent buffer	41295

Table 4: Local vs. remote DMA buffer allocation

Although this change is platform specific, it can be made generic if a `pci_to_node` or `pci_to_nodemask` routine is added to the Linux topology API.

Concluding Remarks

Today, our Linux 2.4.21 kernel for Altix provides a productive platform for our high-performance-computing users who desire to exploit the features of the SGI Altix 3000 hardware. To achieve this goal, we have made a number of changes to our Linux for Altix kernel. We are now in the process of either moving those changes forward to Linux 2.6 for Altix, or of evaluating the Linux 2.6 kernel on Altix in order to determine if these changes are indeed needed at all. Our goal is to develop a version of the Linux 2.6 kernel for Altix that not only supports our HPC customers equally well as our existing Linux 2.4.21 kernel, but also consists as much as possible of community supported code.

References

[1] Ray Bryant and John Hawkes, Linux Scalability for Large NUMA Systems, *Proceedings of the 2003 Ottawa Linux Symposium*, Ottawa, Ontario, Canada, (July 2003).

[2] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennesy, The DASH prototype: Logic overhead and performance, *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[3] Kenneth Chen, "hugetlb demand paging patch part [0/3]," linux-kernel@vger.kernel.org, 2004-04-13 23:17:04, `http://marc.theaimsgroup.com/?l=linux-kernel&m=108189860419356&w=2`

[4] Andi Kleen, "Patch: NUMA API for Linux," linux-kernel@vger.kernel.org,

Tue, 6 Apr 2004 15:33:22 +0200, `http://lwn.net/Articles/79100/`

[5] `http://www.openmp.org`

[6] Nick Piggin, "MM patches," `http://www.kerneltrap.org/~npiggin/nickvm-267r1m1.gz`

[7] `http://www.spec.org/omp/results/ompl2001.html`

[8] `http://linux.bkbits.net:8080/linux-2.5/cset%4040213ca0d3eIznHTPAR_kLCsMZI9VQ?nav=index.html|ChangeSet@-1d`

[9] `http://www.cs.helsinki.fi/linux/linux-kernel/2002-01/1540.html`

Get More Device Drivers out of the Kernel!

Peter Chubb[*]
National ICT Australia
and
The University of New South Wales
peterc@gelato.unsw.edu.au

Abstract

Now that Linux has fast system calls, good (and getting better) threading, and cheap context switches, it's possible to write device drivers that live in user space for whole new classes of devices. Of course, some device drivers (Xfree, in particular) have always run in user space, with a little bit of kernel support. With a little bit more kernel support (a way to set up and tear down DMA safely, and a generalised way to be informed of and control interrupts) almost any PCI bus-mastering device could have a user-mode device driver.

I shall talk about the benefits and drawbacks of device drivers being in user space or kernel space, and show that performance concerns are not really an issue—in fact, on some platforms, our user-mode IDE driver out-performs the in-kernel one. I shall also present profiling and benchmark results that show where time is spent in in-kernel and user-space drivers, and describe the infrastructure I've added to the Linux kernel to allow portable, efficient user-space drivers to be written.

[*]This work was funded by HP, National ICT Australia, the ARC, and the University of NSW through the Gelato programme (http://www.gelato.unsw.edu.au)

1 Introduction

Normal device drivers in Linux run in the kernel's address space with kernel privilege. This is not the only place they can run—see Figure 1.

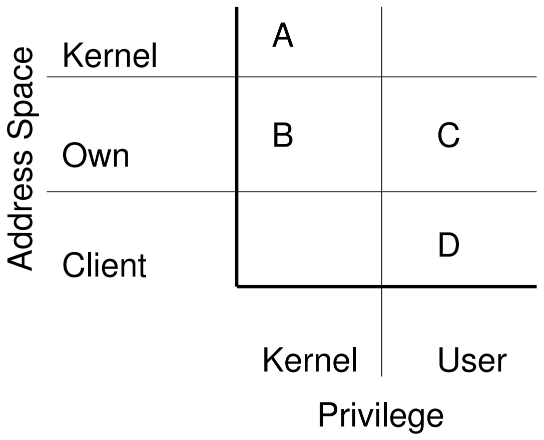


Figure 1: Where a Device Driver can Live

Point A is the normal Linux device driver, linked with the kernel, running in the kernel address space with kernel privilege.

Device drivers can also be linked directly with the applications that use them (Point B)—the so-called 'in-process' device drivers proposed by [Keedy, 1979]—or run in a separate process, and be talked to by an IPC mechanism (for example, an X server, point D). They can also run with kernel privilege, but with a separate kernel address space (Point

C) (as in the Nooks system described by [Swift et al., 2002]).

2 Motivation

Traditionally, device drivers have been developed as part of the kernel source. As such, they *have* to be written in the C language, and they have to conform to the (rapidly changing) interfaces and conventions used by kernel code. Even though drivers can be written as modules (obviating the need to reboot to try out a new version of the driver[1]), in-kernel driver code has access to all of kernel memory, and runs with privileges that give it access to all instructions (not just unprivileged ones) and to all I/O space. As such, bugs in drivers can easily cause kernel lockups or panics. And various studies (e.g., [Chou et al., 2001]) estimate that more than 85% of the bugs in an operating system are driver bugs.

Device drivers that run as user code, however, can use any language, can be developed using any IDE, and can use whatever internal threading, memory management, etc., techniques are most appropriate. When the infrastructure for supporting user-mode drivers is adequate, the processes implementing the driver can be killed and restarted almost with impunity as far as the rest of the operating system goes.

Drivers that run in the kernel have to be updated regularly to match in-kernel interface changes. Third party drivers are therefore usually shipped as source code (or with a compilable stub encapsulating the interface) that has to be compiled against the kernel the driver is to be installed into.

This means that everyone who wants to run a third-party driver also has to have a toolchain and kernel source on his or her system, or obtain a binary for their own kernel from a trusted third party.

Drivers for uncommon devices (or devices that the mainline kernel developers do not use regularly) tend to lag behind. For example, in the 2.6.6 kernel, there are 81 drivers known to be broken because they have not been updated to match the current APIs, and a number more that are still using APIs that have been deprecated.

User/kernel interfaces tend to change much more slowly than in-kernel ones; thus a user-mode driver has much more chance of not needing to be changed when the kernel changes. Moreover, user mode drivers can be distributed under licences other than the GPL, which may make them more attractive to some people[2].

User-mode drivers can be either closely or loosely coupled with the applications that use them. Two obvious examples are the X server (XFree86) which uses a socket to communicate with its clients and so has isolation from kernel and client address spaces and can be very complex; and the Myrinet drivers, which are usually linked into their clients to gain performance by eliminating context switch overhead on packet reception.

The Nooks work [Swift et al., 2002] showed that by isolating drivers from the kernel address space, the most common programming errors could be made recoverable. In Nooks, drivers are insulated from the rest of the kernel by running each in a separate address space, and replacing the driver $\leftrightarrow$ kernel interface with a new one that uses cross-domain procedure calls to replace any procedure calls in the ABI, and that creates shadow copies of any

[1]except that many drivers currently cannot be unloaded

[2]for example, the ongoing problems with the Nvidia graphics card driver could possibly be avoided.

shared variables in the protected address space of the driver.

This approach provides isolation, but also has problems: as the driver model changes, there is quite a lot of wrapper code that has to be changed to accommodate the changed APIs. Also, the value of any shared variable is frozen for the duration of a driver ABI call. The Nooks work is uniprocessor only; locking issues therefore have not yet been addressed.

Windriver [Jungo, 2003] allows development of user mode device drivers. It loads a proprietary device module `/dev/windrv6`; user code can interact with this device to setup and teardown DMA, catch interrupts, etc.

Even from user space, of course, it is possible to make your machine unusable. Device drivers have to be trusted to a certain extent to do what they are advertised to do; this means that they can program their devices, and possibly corrupt or spy on the data that they transfer between their devices and their clients. Moving a driver to user space does not change this. It does however make it less likely that a fault in a driver will affect anything other than its clients

3 Existing Support

Linux has good support for user-mode drivers that do not need DMA or interrupt handling—see, e.g., [Nakatani, 2002].

The `ioperm()` and `iopl()` system calls allow access to the first 65536 I/O ports; and, with a patch from Albert Calahan[3] one can map the appropriate parts of */proc/bus/pci/...* to gain access to memory-mapped registers. Or on some architectures it is safe to `mmap()` */dev/mem*.

[3]`http://lkml.org/lkml/2003/7/13/258`

It is usually best to use MMIO if it is available, because on many 64-bit platforms there are more than 65536 ports—the PCI specification says that there are 2^{32} ports available—(and on many architectures the ports are emulated by mapping memory anyway).

For particular devices—USB input devices, SCSI devices, devices that hang off the parallel port, and video drivers such as XFree86—there is explicit kernel support. By opening a file in */dev*, a user-mode driver can talk through the USB hub, SCSI controller, AGP controller, etc., to the device. In addition, the *input* handler allows input events to be queued back into the kernel, to allow normal event handling to proceed.

libpci allows access to the PCI configuration space, so that a driver can determine what interrupt, IO ports and memory locations are being used (and to determine whether the device is present or not).

Other recent changes—an improved scheduler, better and faster thread creation and synchronisation, a fully preemptive kernel, and faster system calls—mean that it is possible to write a driver that operates in user space that is almost as fast as an in-kernel driver.

4 Implementing the Missing Bits

The parts that are missing are:

1. the ability to claim a device from user space so that other drivers do not try to handle it;

2. The ability to deliver an interrupt from a device to user space,

3. The ability to set up and tear-down DMA between a device and some process's memory, and

4. the ability to loop a device driver's control and data interfaces into the appropriate part of the kernel (so that, for example, an IDE driver can appear as a standard block device), preferably without having to copy any payload data.

The work at UNSW covers only PCI devices, as that is the only bus available on all of the architectures we have access to (IA64, X86, MIPS, PPC, alpha and arm).

4.1 PCI interface

Each device should have only a single driver. Therefore one needs a way to associate a driver with a device, and to remove that association automatically when the driver exits. This has to be implemented in the kernel, as it is only the kernel that can be relied upon to clean up after a failed process. The simplest way to keep the association and to clean it up in Linux is to implement a new filesystem, using the PCI namespace. Open files are automatically closed when a process exits, so cleanup also happens automatically.

A new system call, `usr_pci_open(int bus, int slot, int fn)` returns a file descriptor. Internally, it calls `pci_enable_device()` and `pci_set_master()` to set up the PCI device after doing the standard filesystem boilerplate to set up a vnode and a `struct file`.

Attempts to open an already-opened PCI device will fail with `-EBUSY`.

When the file descriptor is finally closed, the PCI device is released, and any DMA mappings removed. All files are closed when a process dies, so if there is a bug in the driver that causes it to crash, the system recovers ready for the driver to be restarted.

4.2 DMA handling

On low-end systems, it's common for the PCI bus to be connected directly to the memory bus, so setting up a DMA transfer means merely pinning the appropriate bit of memory (so that the VM system can neither swap it out nor relocate it) and then converting virtual addresses to physical addresses.

There are, in general, two kinds of DMA, and this has to be reflected in the kernel interface:

1. Bi-directional DMA, for holding scatter-gather lists, etc., for communication with the device. Both the CPU and the device read and write to a shared memory area. Typically such memory is uncached, and on some architectures it has to be allocated from particular physical areas. This kind of mapping is called *PCI-consistent*; there is an internal kernel ABI function to allocate and deallocate appropriate memory.

2. Streaming DMA, where, once the device has either read or written the area, it has no further immediate use for it.

I implemented a new system call[4], `usr_pci_map()`, that does one of three things:

1. Allocates an area of memory suitable for a PCI-consistent mapping, and maps it into the current process's address space; or

2. Converts a region of the current process's virtual address space into a scatterlist in terms of virtual addresses (one entry per page), pins the memory, and converts the

[4]Although multiplexing system calls are in general deprecated in Linux, they are extremely useful while developing, because it is not necessary to change every architecture-dependent *entry.S* when adding new functionality

scatterlist into a list of addresses suitable for DMA (by calling `pci_map_sg()`, which sets up the IOMMU if appropriate), or

3. Undoes the mapping in point 2.

The file descriptor returned from `usr_pci_open()` is an argument to `usr_pci_map()`. Mappings are tracked as part of the private data for that open file descriptor, so that they can be undone if the device is closed (or the driver dies).

Underlying `usr_pci_map()` are the kernel routines `pci_map_sg()` and `pci_unmap_sg()`, and the kernel routine `pci_alloc_consistent()`.

Different PCI cards can address different amounts of DMA address space. In the kernel there is an interface to request that the dma addresses supplied are within the range addressable by the card. The current implementation assumes 32-bit addressing, but it would be possible to provide an interface to allow the real capabilities of the device to be communicated to the kernel.

4.2.1 The IOMMU

Many modern architectures have an IO memory management unit (see Figure 2), to convert from physical to I/O bus addresses—in much the same way that the processor's MMU converts virtual to physical addresses—allowing even thirty-two bit cards to do single-cycle DMA to anywhere in the sixty-four bit memory address space.

On such systems, after the memory has been pinned, the IOMMU has to be set up to translate from bus to physical addresses; and then after the DMA is complete, the translation can be removed from the IOMMU.

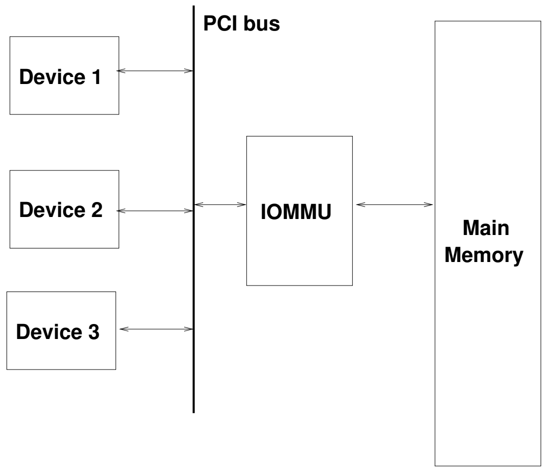


Figure 2: The IO MMU

The processor's MMU also protects one virtual address space from another. Currently shipping IOMMU hardware does not do this: all mappings are visible to all PCI devices, and moreover for some physical addresses on some architectures the IOMMU is bypassed.

For fully secure user-space drivers, one would want this capability to be turned off, and also to be able to associate a range of PCI bus addresses with a particular card, and disallow access by that card to other addresses. Only thus could one ensure that a card could perform DMA only into memory areas explicitly allocated to it.

4.3 Interrupt Handling

There are essentially two ways that interrupts can be passed to user level.

They can be mapped onto signals, and sent asynchronously, or a synchronous 'wait-for-signal' mechanism can be used.

A signal is a good intuitive match for what an interrupt *is*, but has other problems:

1. One is fairly restricted in what one can do in a signal handler, so a driver will usually

have to take extra context switches to respond to an interrupt (into and out of the signal handler, and then perhaps the interrupt handler thread wakes up)

2. Signals can be slow to deliver on busy systems, as they require the process table to be locked. It would be possible to short circuit this to some extent.

3. One needs an extra mechanism for registering interest in an interrupt, and for tearing down the registration when the driver dies.

For these reasons I decided to map interrupts onto file descriptors. */proc* already has a directory for each interrupt (containing a file that can be written to to adjust interrupt routing to processors); I added a new file to each such directory. Suitably privileged processes can open and read these files. The files have open-once semantics; attempts to open them while they are open return -1 with `EBUSY`.

When an interrupt occurs, the in-kernel interrupt handler masks just that interrupt in the interrupt controller, and then does an `up()` operation on a semaphore (well, actually, the implementation now uses a wait queue, but the effect is the same).

When a process reads from the file, then kernel enables the interrupt, then calls `down()` on a semaphore, which will block until an interrupt arrives.

The actual data transferred is immaterial, and in fact none ever is transferred; the `read()` operation is used merely as a synchronisation mechanism.

`poll()` is also implemented, so a user process is not forced into the 'wait for interrupt' model that we use.

Obviously, one cannot share interrupts between devices if there is a user process involved. The in-kernel driver merely passes the interrupt onto the user-mode process; as it knows nothing about the underlying hardware, it cannot tell if the interrupt is *really* for this driver or not. As such it always reports the interrupt as 'handled.'

This scheme works only for level-triggered interrupts. Fortunately, all PCI interrupts are level triggered.

If one really wants a signal when an interrupt happens, one can arrange for a `SIGIO` using `fcntl()`.

It may be possible, by more extensive rearrangement of the interrupt handling code, to delay the end-of-interrupt to the interrupt controller until the user process is ready to get an interrupt. As masking and unmasking interrupts is slow if it has to go off-chip, delaying the EOI should be significantly faster than the current code. However, interrupt delivery to userspace turns out not to be a bottleneck, so there's not a lot of point in this optimisation (profiles show less than 0.5% of the time is spent in the kernel interrupt handler and delivery even for heavy interrupt load—around 1000 cycles per interrupt).

5 Driver Structure

The user-mode drivers developed at UNSW are structured as a preamble, an interrupt thread, and a control thread (see Figure 3).

The preamble:

1. Uses *libpci.a* to find the device or devices it is meant to drive,

2. Calls `usr_pci_open()` to claim the device, and

3. Spawns the interrupt thread, then

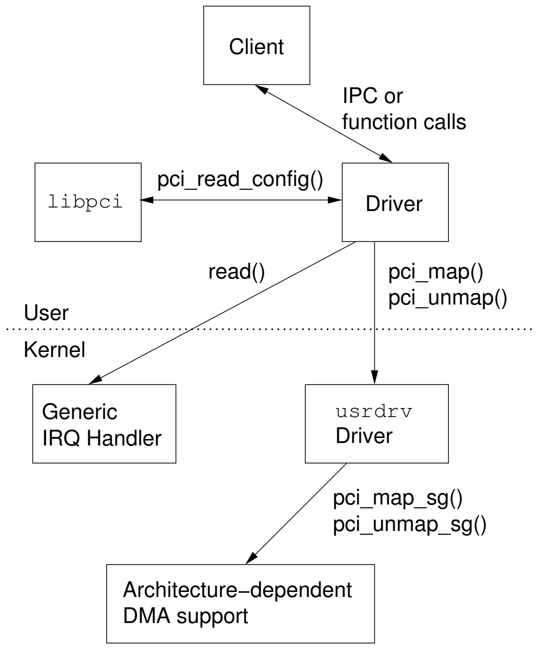


Figure 3: Architecture of a User-Mode Device Driver

4. Goes into a loop collecting client requests.

The interrupt thread:

1. Opens */proc/irq/***irq***/irq*
2. Loops calling `read()` on the resulting file descriptor and then calling the driver proper to handle the interrupt.
3. The driver handles the interrupt, calls out to the control thread(s) to say that work is completed or that there has been an error, queues any more work to the device, and then repeats from step 2.

For the lowest latency, the interrupt thread can be run as a real time thread. For our benchmarks, however, this was not done.

The control thread queues work to the driver then sleeps on a semaphore. When the driver, running in the interrupt thread, determines that a request is complete, it signals the semaphore so that the control thread can continue. (The semaphore is implemented as a pthreads mutex).

The driver relies on system calls and threading, so the fast system call support now available in Linux, and the NPTL are very important to get good performance. Each physical I/O involves at least three system calls, plus whatever is necessary for client communication: a `read()` on the interrupt FD, calls to set up and tear down DMA, and maybe a `futex()` operation to wake the client.

The system call overhead could be reduced by combining DMA setup and teardown into a single system call.

6 Looping the Drivers

An operating system has two functions with regard to devices: firstly to drive them, and secondly to abstract them, so that all devices of the same class have the same interface. While a standalone user-level driver is interesting in its own right (and could be used, for example, to test hardware, or could be linked into an application that doesn't like sharing the device with anyone), it is much more useful if the driver can be used like any other device.

For the network interface, that's easy: use the tun/tap interface and copy frames between the driver and */dev/net/tun*. Having to copy slows things down; others on the team here are planning to develop a zero-copy equivalent of tun/tap.

For the IDE device, there's no standard Linux way to have a user-level block device, so I implemented one. It is a filesystem that has pairs of directories: a master and a slave. When the filesystem is mounted, creating a file in the master directory creates a set of block device special files, one for each potential partition, in the slave directory. The file in the master directory can then be used to communicate via a very simple protocol between a user level block device and the kernel's block layer. The block device special files in the slave directory can then be opened, closed, read, written or mounted, just as any other block device.

The main reason for using a mounted filesystem was to allow easy use of dynamic major numbers.

I didn't bother implementing ioctl; it was not necessary for our performance tests, and when the driver runs at user level, there are cleaner ways to communicate out-of-band data with the driver, anyway.

7 Results

Device drivers were coded up by [Leslie and Heiser, 2003] for a CMD680 IDE disc controller, and by another PhD student (Daniel Potts) for a DP83820 Gigabit ethernet controller. Daniel also designed and implemented the tuntap interface.

7.1 IDE driver

The disc driver was linked into a program that read 64 Megabytes of data from a Maxtor 80G disc into a buffer, using varying read sizes. Measurements were also made using Linux's in-kernel driver, and a program that read 64M of data from the same on-disc location using `O_DIRECT` and the same read sizes.

We also measured write performance, but the results are sufficiently similar that they are not reproduced here.

At the same time as the tests, a low-priority process attempted to increment a 64-bit counter as fast as possible. The number of increments was calibrated to processor time on an otherwise idle system; reading the counter before and after a test thus gives an indication of how much processor time is available to processes other than the test process.

The initial results were disappointing; the user-mode drivers spent far too much time in the kernel. This was tracked down to `kmalloc()`; so the `usr_pci_map()` function was changed to maintain a small cache of free mapping structures instead of calling `kmalloc()` and `kfree()` each time (we could have used the slab allocator, but it's easier to ensure that the same cache-hot descriptor is reused by coding a small cache ourselves). This resulted in the performance graphs in Figure 4.

The two drivers compared are the new CMD680 driver running in user space, and Linux's in-kernel SIS680 driver. As can be seen, there is very little to choose between them.

The graphs show average of ten runs; the standard deviations were calculated, but are negligible.

Each transfer request takes five system calls to do, in the current design. The client queues work to the driver, which then sets up DMA for the transfer (system call one), starts the transfer, then returns to the client, which then sleeps on a semaphore (system call two). The interrupt thread has been sleeping in `read()`, when the controller finishes its DMA, it cause an interrupt, which wakes the interrupt thread (half of system call three). The interrupt thread then tears down the DMA (system call four), and starts any queued and waiting activity, then signals the semaphore (system call five) and goes back to read the interrupt FD again (the other half of system call three).

When the transfer is above 128k, the IDE controller can no longer do a single DMA opera-

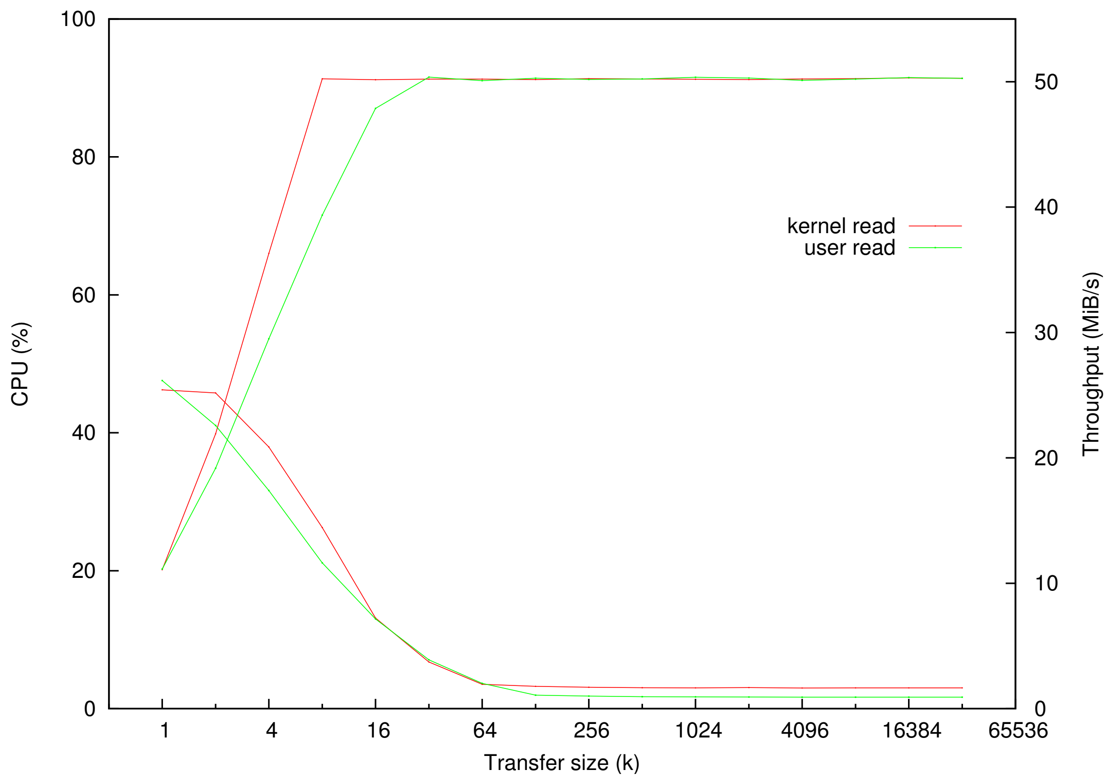


Figure 4: Throughput and CPU usage for the user-mode IDE driver on Itanium-2, reading from a disk

tion, so has to generate multiple transfers The Linux kernel splits DMA requests above 64k, thus increasing the overhead.

The time spent in this driver is divided as shown in Figure 5.

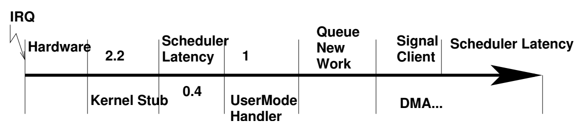


Figure 5: Timeline (in μseconds)

7.2 Gigabit Ethernet

The Gigabit driver results are more interesting. We tested these using [ipbench, 2004] with four clients, all with pause control turned off. We ran three tests:

1. Packet receive performance, where packets were dropped and counted at the layer immediately above the driver
2. Packet transmit performance, where packets were generated and fed to the driver, and
3. Ethernet-layer packet echoing, where the protocol layer swapped source and destination MAC-addresses, and fed received packets back into the driver.

We did not want to start comparing IP stacks, so none of these tests actually use higher level protocols.

We measured three different configurations: a standalone application linked with the driver, the driver looped back into */dev/net/tap* and the standard in-kernel driver, all with interrupt holdoff set to 0, 1, or 2. (By default, the normal kernel driver sets the interrupt holdoff to 300 μseconds, which led to too many packets being dropped because of FIFO overflow) Not all tests were run in all configurations—for example the linux in-kernel packet generator is sufficiently different from ours that no fair comparison could be made.

For the tests that had the driver residing in or feeding into the kernel, we implemented a new protocol module to count and either echo or drop packets, depending on the benchmark.

In all cases, we used the amount of work achieved by a low priority process to measure time available for other work while the test was going on.

The throughput graphs in all cases are the same. The maximum possible speed on the wire is given for raw ethernet by $10^9 \times p/(p + 38)$ bits per second (the parameter 38 is the ethernet header size (14 octets), plus a 4 octet frame check sequence, plus a 7 octet preamble, plus a 1 octet start frame delimiter plus the minimum 12 octet interframe gap; p is the packet size in octets). For large packets the performance in all cases was the same as the theoretical maximum. For small packet sizes, the throughput is limited by the PCI bus; you'll notice that the slope of the throughput curve when echoing packets is around half the slope when discarding packets, because the driver has to do twice as many DMA operations per packet.

The user-mode driver ('Linux user' on the graph) outperforms the in-kernel driver ('Linux orig')—not in terms of throughput, where all the drivers perform identically, but in using *much* less processing time.

This result was so surprising that we repeated the tests using an EEpro1000, purportedly a card with a much better driver, but saw the same effect—in fact the achieved echo performance is worse than for the in-kernel ns83820 driver for some packet sizes.

The reason appears to be that our driver has a fixed number of receive buffers, which are reused when the client is finished with them—they are allocated only once. This is to provide congestion control at the lowest possible level—the card drops packets when the upper layers cannot keep up.

The Linux kernel drivers have an essentially unlimited supply of receive buffers. Overhead involved in allocating and setting up DMA for these buffers is excessive, and if the upper layers cannot keep up, congestion is detected and the packets dropped in the protocol layer—after significant work has been done in the driver.

One sees the same problem with the user mode driver feeding the tuntap interface, as there is no feedback to throttle the driver. Of course, here there is an extra copy for each packet, which also reduces performance.

7.3 Reliability and Failure Modes

In general the user-mode drivers are very reliable. Bugs in the drivers that would cause the kernel to crash (for example, a null pointer reference inside an interrupt handler) cause the driver to crash, but the kernel continues. The driver can then be fixed and restarted.

8 Future Work

The main foci of our work now lie in:

1. Reducing the need for context switches and system calls by merging system calls, and by trying new driver structures.

2. A zero-copy implementation of tun/tap.

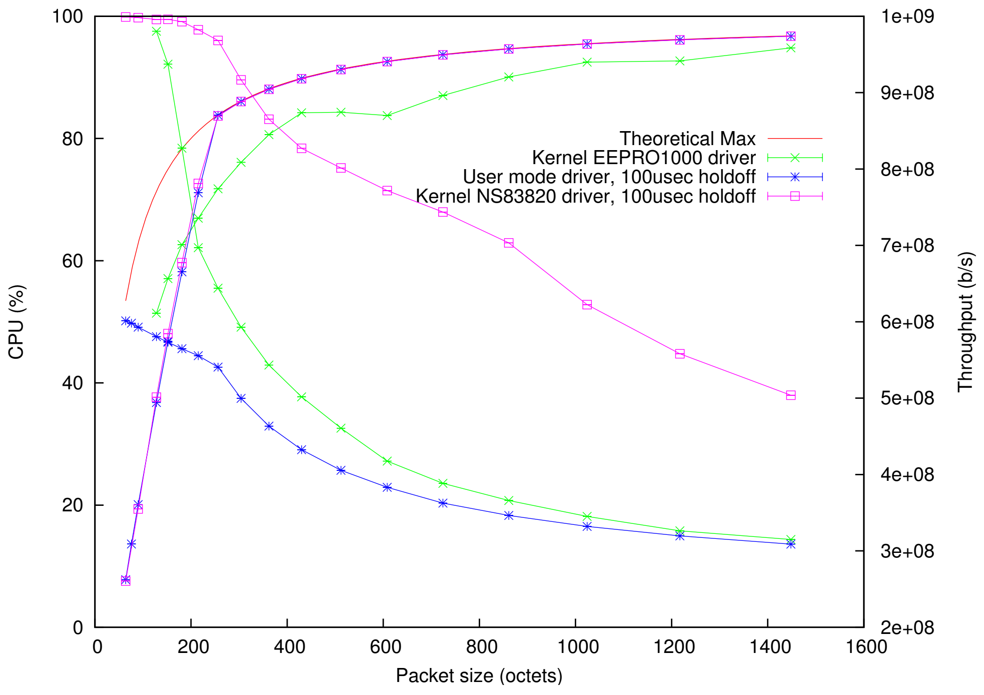


Figure 6: Receive Throughput and CPU usage for Gigabit Ethernet drivers on Itanium-2

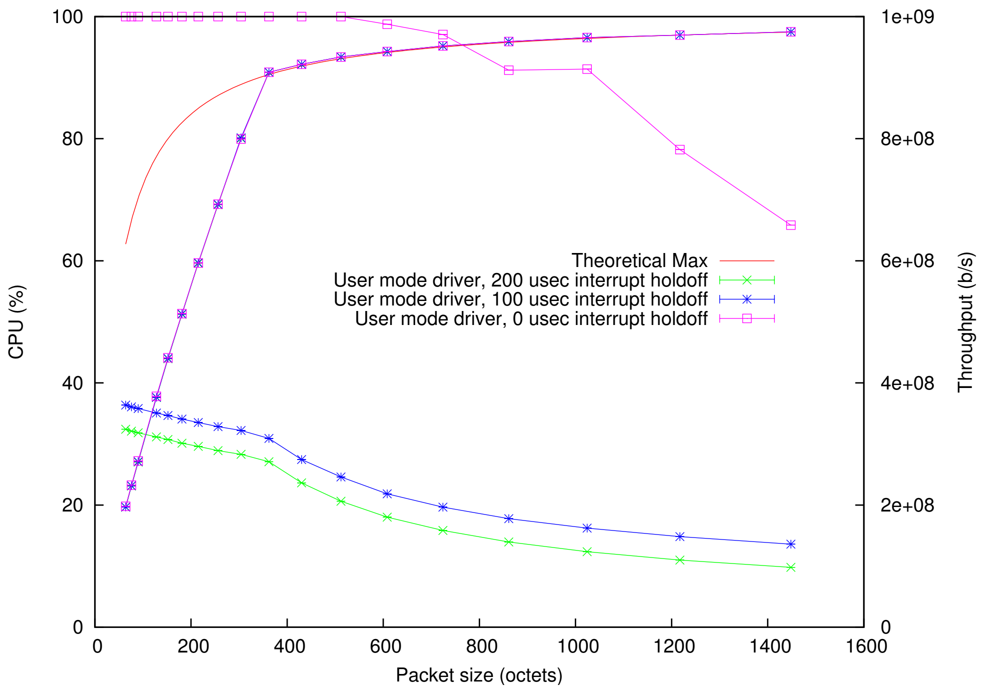


Figure 7: Transmit Throughput and CPU usage for Gigabit Ethernet drivers on Itanium-2

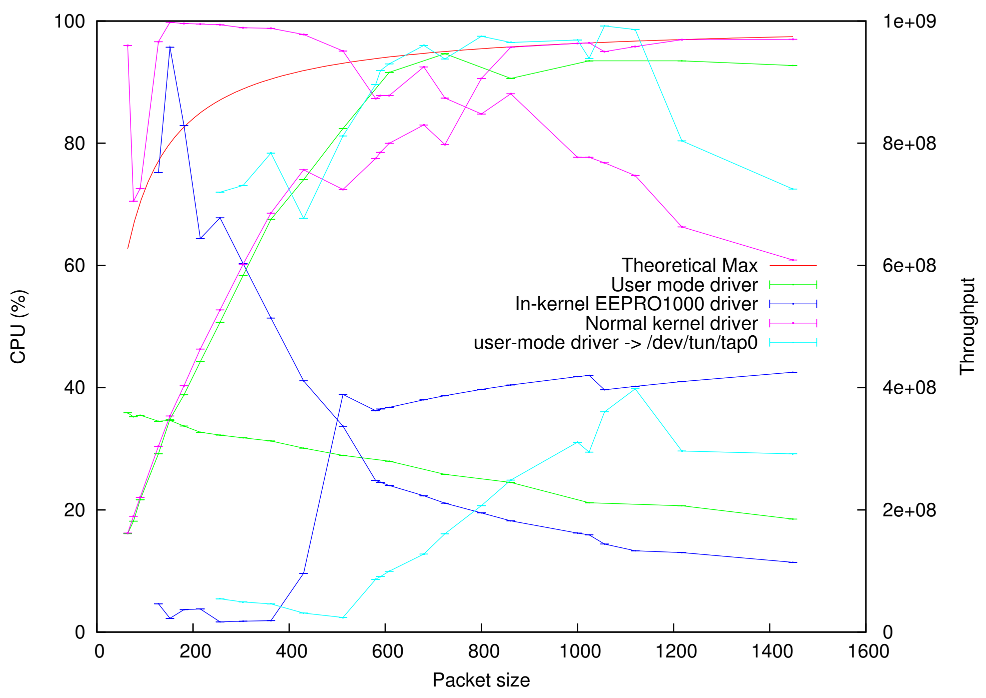


Figure 8: MAC-layer Echo Throughput and CPU usage for Gigabit Ethernet drivers on Itanium-2

3. Improving robustness and reliability of the user-mode drivers, by experimenting with the IOMMU on the ZX1 chipset of our Itanium-2 machines.

4. Measuring the reliability enhancements, by using artificial fault injection to see what problems that cause the kernel to crash are recoverable in user space.

5. User-mode filesystems.

In addition there are some housekeeping tasks to do before this infrastructure is ready for inclusion in a 2.7 kernel:

1. Replace the ad-hoc memory cache with a proper slab allocator.

2. Clean up the system call interface

9 Where d'ya Get It?

Patches against the 2.6 kernel are sent to the Linux kernel mailing list, and are on `http://www.gelato.unsw.edu.au/patches`

Sample drivers will be made available from the same website.

10 Acknowledgements

Other people on the team here did much work on the actual implementation of the user level drivers and on the benchmarking infrastructure. Prominent among them were Ben Leslie (IDE driver, port of our dp83820 into the kernel), Daniel Potts (DP83820 driver, tuntap interface), and Luke McPherson and Ian Wienand (IPbench).

References

[Chou et al., 2001] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. (2001). An empirical study of operating systems errors. In *Symposium on Operating Systems Principles*, pages 73–88. `http://citeseer.nj.nec.com/article/chou01empirical.html`.

[ipbench, 2004] ipbench (2004). ipbench — a distributed framework for network benchmarking. `http://ipbench.sf.net/`.

[Jungo, 2003] Jungo (2003). Windriver. `http://www.jungo.com/windriver.html`.

[Keedy, 1979] Keedy, J. L. (1979). A comparison of two process structuring models. MONADS Report 4, Dept. Computer Science, Monash University.

[Leslie and Heiser, 2003] Leslie, B. and Heiser, G. (2003). Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, Operating Systems and Distributed Systems Group, School of Computer Science and Engineering, The University of NSW. CSE techreports website, `ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/0303.pdf`.

[Nakatani, 2002] Nakatani, B. (2002). ELJOnline: User mode drivers. `http://www.linuxdevices.com/articles/AT5731658926.html`.

[Swift et al., 2002] Swift, M., Martin, S., Leyand, H. M., and Eggers, S. J. (2002). Nooks: an architecture for reliable device drivers. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France.

Big Servers—2.6 compared to 2.4

Wim A. Coekaerts
Oracle Corporation
`wim.coekaerts@oracle.com`

Abstract

Linux 2.4 has been around in production environments at companies for a few years now, we have been able to gather some good data on how well (or not) things scale up. Number of CPU's, amount of memory, number of processes, IO throughput, etc.

Most of the deployments in production today, are on relatively small systems, 4- to 8-ways, 8–16GB of memory, in a few cases 32GB. The architecture of choice has also been IA32. 64-bit systems are picking up in popularity rapidly, however.

Now with 2.6, a lot of the barriers are supposed to be gone. So, have they really? How much memory can be used now, how is cpu scaling these days, how good is IO throughput with multiple controllers in 2.6.

A lot of people have the assumption that 2.6 resolves all of this. We will go into detail on what we have found out, what we have tested and some of the conclusions on how good the move to 2.6 will really be.

1 Introduction

The comparison between the 2.4 and 2.6 kernel trees are not solely based on performance. A large part of the testsuites are performance benchmarks however, as you will see, they have been used to also measure stability. There are a number of features added which improve stability of the kernel under heavy workloads. The goal of comparing the two kernel releases was more to show how well the 2.6 kernel will be able to hold up in a real world production environment. Many companies which have deployed Linux over the last two years are looking forward to rolling out 2.6 and it is important to show the benefits of doing such a move. It will take a few releases before the required stability is there however it's clear so far that the 2.6 kernel has been remarkably solid, so early on.

Most of the 2.4 based tests have been run on Red Hat Enterprise Linux 3, based on Linux 2.4.21. This is the enterprise release of Red Hat's OS distribution; it contains a large number of patches on top of the Linux 2.4 kernel tree. Some of the tests have been run on the `kernel.org` mainstream 2.4 kernel, to show the benefit of having extra functionality. However it is difficult to even just boot up the mainstream kernel on the test hardware due to lack of support for drivers, or lack of stability to complete the testsuite. The interesting thing to keep in mind is that with the current Linux 2.6 main stream kernel, most of the testsuites ran through completition. A number of test runs on Linux 2.6 have been on Novell/SuSE SLES9 beta release.

2 Test Suites

The test suites used to compare the various kernels are based on an IO simulator for Oracle, called OraSim and a TPC-C like workload generator called OAST.

Oracle Simulator (OraSim) is a stand-alone tool designed to emulate the platform-critical activities of the Oracle database kernel. Oracle designed Oracle Simulator to test and characterize the input and output (I/O) software stack, the storage system, memory management, and cluster management of Oracle single instances and clusters. Oracle Simulator supports both pass-fail testing for validation, and analytical testing for debugging and tuning. It runs multiple processes, with each process representing the parameters of a particular type of system load similar to the Oracle database kernel.

OraSim is a relatively straightforward IO stresstest utility, similar to IOzone or tiobench, however it is built to be very flexible and configurable.

It has its own script language which allows one to build very complex IO patterns. The tool is not released under any open source license today because it has some code linked in which is part of the RDBMS itself. The jobfiles used for the testing are available online `http://oss.oracle.com/external/ols/jobfiles/`.

The advantage of using OraSim over a real database benchmark is mainly the simplicity. It does not require large amounts of memory or large installed software components. There is one executable which is started with the jobfile as a parameter.The jobfiles used can be easily modified to turn on certain filesystem features, such as asynchronous IO.

OraSim jobfiles were created to simulate a relatively small database. 10 files are defined as actual database datafiles and two files are used to simulate database journals.

OAST on the other hand is a complete database stress test kit, based on the TPC-C benchmark workloads. It requires a full installation of the database software and relies on an actual database environment to be created. TPC-C is an on-line transaction workload. The numbers represented during the testruns are not actual TPC-C benchmarks results and cannot or should not be used as a measure of TPC-C performance—they are TPC-C-like; however, not the same.

The database engine which runs the OAST benchmark allocates a large shared memory segment which contains the database caches for SQL and for data blocks (shared pool and buffer cache). Every client connection can run on the same server or the connection can be over TCP. In case of a local connection, for each client, 2 processes are spawned on the system. One process is a dedicated database process and the other is the client code which communicates with the database server process through IPC calls. Test run parameters include run time length in seconds and number of client connections. As you can see in the result pages, both remote and local connections have been tested.

3 Hardware

A number of hardware configurations have been used. We tried to include various CPU architectures as well as local SCSI disk versus network storage (NAS) and fibre channel (SAN).

Configuration 1 consists of an 8-way IA32 Xeon 2 GHz with 32GB RAM attached to an EMC CX300 Clariion array with 30 147GB disks using a QLA2300 fibre channel HBA. The network cards are BCM5701 Broadcom Gigabit Ethernet.

Configuration 2 consists of an 8-way Itanium 2 1.3 GHz with 8GB RAM attached to a JBOD fibre channel array with 8 36GB disks using a QLA2300 fibre channel HBA. The network cards are BCM5701 Broadcom Gigabit Ethernet.

Configuration 3 consists of a 2-way AMD64 2 GHz (Opteron 246) with 6GB RAM attached to local SCSI disk (LSI Logic 53c1030).

4 Operating System

The Linux 2.4 test cases were created using Red Hat Enterprise Linux 3 on all architectures. Linux 2.6 was done with SuSE SLES9 on all architectures; however, in a number of tests the kernel was replaced by the 2.6 mainstream kernel for comparison.

The test suites and benchmarks did not have to be recompiled to run on either RHEL3 or SLES9. Of course different executables were used on the three CPU architectures.

5 Test Results

At the time of writing a lot of changes were still happening on the 2.6 kernel. As such, the actual spreadsheets with benchmark data has been published on a website, the data is up-to-date with the current kernel tree and can be found here: `http://oss.oracle.com/external/ols/results/`

5.1 IO

If you want to build a huge database server, which can handle thousands of users, it is important to be able to attach a large number of disks. A very big shortcoming in Linux 2.4 was the fact that it could only handle 128 or 256.

With some patches SuSE got to around 3700 disks in SLES8, however that meant stealing major numbers from other components. Really large database setups which also require very high IO throughput, usually have disks attached ranging from a few hundred to a few thousand.

With the 64-bit `dev_t` in 2.6, it's now possible to attach plenty of disk. Without modifications it can easily handle tens of thousands of devices attached. This opens the world to really large scale datawarehouses, tens of terabytes of storage.

Another important change is the block IO layer, the BIO code is much more efficient when it comes to large IOs being submitted down from the running application. In 2.4, every IO got broken down into small chunks, sometimes causing bottlenecks on allocating accounting structures. Some of the tests compared 1MB `read()` and `write()` calls in 2.4 and 2.6.

5.2 Asynchronous IO and DirectIO

If there is one feature that has always been on top of the Must Have list for large database vendors, it must be async IO. Asynchronous IO allows processes to submit batches of IO operations and continue on doing different tasks in the meantime. It improves CPU utilization and can keep devices more busy. The Enterprise distributions based on Linux 2.4 all ship with the async IO patch applied on top of the mainline kernel.

Linux 2.6 has async IO out of the box. It is implemented a little different from Linux 2.4 however combined with support for direct IO it is very performant. Direct IO is very useful as it eliminates copying the userspace buffers into kernel space. On systems that are constantly overloaded, there is a nice performance improvement to be gained doing direct IO. Linux 2.4 did not have direct IO and async IO combined. As you can see in the performance graph on AIO+DIO, it provides a significant reduction in CPU utilization.

5.3 Virtual Memory

There has been another major VM overhaul in Linux 2.6, in fact, even after 2.6.0 was released a large portion has been re-written. This was due to large scale testing showing weaknesses as it relates to number of users that could be handled on a system. As you can see in the test results, we were able to go from around 3000 users to over 7000 users. In particular on 32-bit systems, the VM has been pretty much a disaster when it comes to deploying a system with more than 16GB of RAM. With the latest VM changes it is now possible to push a 32GB even up to 48GB system pretty reliably.

Support for large pages has also been a big winner. `HUGETLBFS` reduces TLB misses by a decent percentage. In some of the tests it provides up to a 3% performance gain. In our tests `HUGETLBFS` would be used to allocate the shared memory segment.

5.4 NUMA

Linux 2.6 is the first Linux kernel with real NUMA support. As we see high-end customers looking at deploying large SMP boxes running Linux, this became a real requirement. In fact even with the AMD64 design, NUMA support becomes important for performance even when looking at just a dual-CPU system.

NUMA support has two components; however, one is the fact that the kernel VM allocates memory for processes in a more efficient way. On the other hand, it is possible for applications to use the NUMA API and tell the OS where memory should be allocated and how. Oracle has an extention for Itanium2 to support the libnuma API from Andi Kleen. Making use of this extention showed a significant improvement, up to about 20%. It allows the database engine to be smart about memory allocations resulting in a significant performance gain.

6 Conclusion

It is very clear that many of the features that were requested by the larger corporations providing enterprise applications actually help a huge amount. The advantage of having Asynchronous IO or NUMA support in the mainstream kernel is obvious. It takes a lot of effort for distribution vendors to maintain patches on top of the mainline kernel and when functionality makes sense it helps to have it be included in mainline. Micro-optimizations are still being done and in particular the VM subsystem can improve quite a bit. Most of the stability issues are around 32-bit, where the LowMem versus HighMem split wreaks havoc quite frequently. At least with some of the features now in the 2.6 kernel it is possible to run servers with more than 16GB of memory and scale up.

The biggest surprise was the stability. It was very nice to see a new stable tree be so solid out of the box, this in contrast to earlier stable kernel trees where it took quite a few iterations to get to the same point.

The major benefit of 2.6 is being able to run on really large SMP boxes: 32-way Itanium2 or Power4 systems with large amounts of memory. This was the last stronghold of the traditional Unices and now Linux can play alongside with them even there. Very exciting times.

Multi-processor and Frequency Scaling

Making Your Server Behave Like a Laptop

Paul Devriendt
AMD Software Research and Development
paul.devriendt@amd.com

Abstract

This paper will explore a multi-processor implementation of frequency management, using an AMD Opteron™ processor 4-way server as a test vehicle.

Topics will include:

- the benefits of doing this, and why server customers are asking for this,
- the hardware, for case of the AMD Opteron processor,
- the various software components that make this work,
- the issues that arise, and
- some areas of exploration for follow on work.

1 Introduction

Processor frequency management is common on laptops, primarily as a mechanism for improving battery life. Other benefits include a cooler processor and reduced fan noise. Fans also use a non-trivial amount of power.

This technology is spreading to desktop machines, driven both by a desire to reduce power consumption and to reduce fan noise.

Servers and other multiprocessor machines can equally benefit. The multiprocessor frequency management scenario offers more complexity (no surprise there). This paper discusses these complexities, based upon a test implementation on an AMD Opteron processor 4-way server. Details within this paper are AMD processor specific, but the concepts are applicable to other architectures.

The author of this paper would like to make it clear that he is just the maintainer of the AMD frequency driver, supporting the AMD Athlon™ 64 and AMD Opteron processors. This frequency driver fits into, and is totally dependent, on the CPUFreq support. The author has gratefully received much assistance and support from the CPUFreq maintainer (Dominik Brodowski).

2 Abbreviations

BKDG: The BIOS and Kernel Developer's Guide. Document published by AMD containing information needed by system software developers. See the references section, entry 4.

MSR: Model Specific Register. Processor registers, accessable only from kernel space, used for various control functions. These registers are expected to change across processor families. These registers are described in the

BKDG[4].

VRM: Voltage Regulator Module. Hardware external to the processor that controls the voltage supplied to the processor. The VRM has to be capable of supplying different voltages on command. Note that for multiprocessor systems, it is expected that each processor will have its own independent VRM, allowing each processor to change voltage independently. For systems where more than one processor shares a VRM, the processors have to be managed as a group. The current frequency driver does not have this support.

fid: Frequency Identifier. The values written to the control MSR to select a core frequency. These identifiers are processor family specific. Currently, these are six bit codes, allowing the selection of frequencies from 800 MHz to 5 Ghz. See the BKDG[4] for the mappings from fid to frequency. Note that the frequency driver does need to "understand" the mapping of fid to frequency, as frequencies are exposed to other software components.

vid: Voltage Identifier. The values written to the control MSR to select a voltage. These values are then driven to the VRM by processor logic to achieve control of the voltage. These identifiers are processor model specific. Currently these identifiers are five bit codes, of which there are two sets—a standard set and a low-voltage mobile set. The frequency driver does not need to be able to "understand" the mapping of vid to voltage, other than perhaps for debug prints.

VST: Voltage Stabilization Time. The length of time before the voltage has increased and is stable at a newly increased voltage. The driver has to wait for this time period when stepping the voltage up. The voltage has to be stable at the new level before applying a further step up in voltage, or before transitioning to a new frequency that requires the higher voltage.

MVS: Maximum Voltage Step. The maximum voltage step that can be taken when increasing the voltage. The driver has to step up voltage in multiple steps of this value when increasing the voltage. (When decreasing voltage it is not necessary to step, the driver can merely jump to the correct voltage.) A typical MVS value would be 25mV.

RVO: Ramp Voltage Offset. When transitioning frequencies, it is necessary to temporarily increase the nominal voltage by this amount during the frequency transition. A typical RVO value would be 50mV.

IRT: Isochronous Relief Time. During frequency transitions, busmasters briefly lose access to system memory. When making multiple frequency changes, the processor driver must delay the next transition for this time period to allow busmasters access to system memory. The typical value used is 80us.

PLL: Phase Locked Loop. Electronic circuit that controls an oscillator to maintain a constant phase angle relative to a reference signal. Used to synthesize new frequencies which are a multiple of a reference frequency.

PLL Lock Time: The length of time, in microseconds, for the PLL to lock.

pstate: Performance State. A combination of frequency/voltage that is supported for the operation of the processor. A processor will typically have several pstates available, with higher frequencies needing higher voltages. The processor clock can not be set to any arbitrary frequency; it may only be set to one of a limited set of frequencies. For a given frequency, there is a minimum voltage needed to operate reliably at that frequency, and this is the correct voltage, thus forming the frequency/voltage pair.

ACPI: Advanced Configuration and Power Interface Specification. An industry specification, initially developed by Intel, Microsoft, Phoenix and Toshiba. See the reference section, entry 5.

_PSS: Performance Supported States. ACPI object that defines the performance states valid for a processor.

_PPC: Performance Present Capabilities. ACPI object that defines which of the _PSS states are currently available, due to current platform limitations.

PSB Performance State Block. BIOS provided data structure used to pass information, to the driver, concerning the pstates available on the processor. The PSB does not support multiprocessor systems (which use the ACPI _PSS object instead) and is being deprecated. The format of the PSB is defined in the BKDG.

3 Why Does Frequency Management Affect Power Consumption?

Higher frequency requires higher voltage. As an example, data for part number ADA3200AEP4AX:

2.2 GHz @ 1.50 volts, 58 amps max – 89 watts

2.0 GHz @ 1.40 volts, 48 amps max – 69 watts

1.8 GHz @ 1.30 volts, 37 amps max – 50 watts

1.0 GHz @ 1.10 volts, 18 amps max – 22 watts

These figures are worst case current/power figures, at maximum case temperature, and include I/O power of 2.2W.

Actual power usage is determined by:

- code currently executing (idle blocks in the processor consume less power),
- activity from other processors (cache coherency, memory accesses, pass-through traffic on the HyperTransport™ connections),
- processor temperature (current increases with temperature, at constant workload and voltage),
- processor voltage.

Increasing the voltage allows operation at higher frequencies, at the cost of higher power consumption and higher heat generation. Note that relationship between frequency and power consumption is not a linear relationship—a 10% frequency increase will cost more than 10% in power consumption (30% or more).

Total system power usage depends on other devices in the system, such as whether disk drives are spinning or stopped, and on the efficiency of power supplies.

4 Why Should Your Server Behave Like A Laptop?

- Save power. It is the right thing to do for the environment. Note that power consumed is largely converted into heat, which then becomes a load on the air conditioning in the server room.
- Save money. Power costs money. The power savings for a single server are typically regarded as trivial in terms of a corporate budget. However, many large organizations have racks of many thousands of servers. The power bill is then far from trivial.
- Cooler components last longer, and this translates into improved server reliability.
- Government Regulation.

5 Interesting Scenarios

These are real world scenarios, where the application of the technology is appropriate.

5.1 Save power in an idle cluster

A cluster would typically be kept running at all times, allowing remote access on demand. During the periods when the cluster is idle, reducing the CPU frequency is a good way to reduce power consumption (and therefore also air conditioning load), yet be able to quickly transition back up to full speed (<0.1 second) when a job is submitted.

User space code (custom to the management of that cluster) can be used to offer cluster speeds of "fast" and "idle," using the `/proc` or `/sys` file systems to trigger frequency transitions.

5.2 The battery powered server

Or, the server running on a UPS.

Many production servers are connected to a battery backup mechanism (UPS—uninterruptible power supply) in case the mains power fails. Action taken on a mains power failure varies:

- Orderly shutdown.
- Stay up and running for as long as there is battery power, but orderly shutdown if mains power is not restored.
- Stay up and running, mains power will be provided by backup generators as soon as the generators can be started.

In these scenarios, transitioning to lower performance states will maximize battery life, or reduce the amount of generator/battery power capacity required.

UPS notification of mains power loss to the server for administrator alerts is well understood technology. It is not difficult to add the support for transitioning to a lower pstate. This can be done by either a cpufreq governor or by adding the simple user space controls to an existing user space daemon that is monitoring the UPS alerts.

5.3 Server At Less Than Maximum Load

As an example, a busy server may be processing 100 transactions per second, but only 5 transactions per second during quiet periods. Reducing the CPU frequency from 2.2 GHz to 1.0 GHz is not going to impact the ability of that server to process 5 transactions per second.

5.4 Processor is not the bottleneck

The bottleneck may not be the processor speed. Other likely bottlenecks are disk access and network access. Having the processor waiting faster may not improve transaction throughput.

5.5 Thermal cutback to avoid over temperature situations

The processors are the main generators of heat in a system. This becomes very apparent when many processors are in close proximity, such as with blade servers. The effectiveness of the processor cooling is impacted when the processor heat sinks are being cooled with hot air. Reducing processor frequency when idle can dramatically reduce the heat production.

5.6 Smaller Enclosures

The drive to build servers in smaller boxes, whether as standalone machines or slim rackmount machines, means that there is less space for air to circulate. Placing many slim rackmounts together in a rack (of which the most demanding case is a blade server) aggravates the cooling problem as the neighboring boxes are also generating heat.

6 System Power Budget

The processors are only part of the system. We therefore need to understand the power consumption of the entire system to see how significant processor frequency management is on the power consumption of the whole system.

A system power budget is obviously platform specific. This sample DC (direct current) power budget is for a 4-processor AMD Opteron processor based system. The system has three 500W power supplies, of which one is redundant. Analysis shows that for many operating scenarios, the system could run on a single power supply.

This analysis is of DC power. For the system in question, the efficiency of the power supplies are approximately linear across varying loads, and thus the DC power figures expressed as percentages are meaningful as predictors of the AC (alternating current) power consumption. For systems with power supplies that are not linearly efficient across varying loads, the calculations obviously have to be factored to take account of power supply efficiency.

System components:

- 4 processors @ 89W = 356W in the maximum pstate, 4 @ 22W = 88W in the minimum pstate. These are worst case figures, at maximium case temperature, with the worst case instruction mix. The figures in Table1 are reduced from these maximums by approximately 10% to account for a reduced case temperature and for a workload that does not keep all of the processors' internal units busy.
- Two disk drives (Western Digital 250 GByte SATA), 16W read/write, 10W idle (spinning), 1.3W sleep (not spinning). Note SCSI drives typically consume more power.
- DVD Drive, 10W read, 1W idle/sleep.
- PCI 2.2 Slots – absolute max of 25W per slot, system will have a total power budget that may not account for maximum power in all slots. Estimate 2 slots occupied at a total of 20W.
- VGA video card in a PCI slot. 5W. (AGP would be more like 15W+).
- DDR DRAM, 10W max per DIMM, 40W for 4 GBytes configured as 4 DIMMs.
- Network (built in) 5W.
- Motherboard and components 30W.
- 10 fans @ 6W each. 60W.
- Keyboard + Mouse 3W

See Table 1 for the sample power budget under busy and light loads.

The light load without any frequency reduction is baselined as 100%.

The power consumption is shown for the same light load with frequency reduction enabled, and again where the idle loop incorporates the hlt instruction.

Using frequency management, the power consumption drops to 43%, and adding the use of the hlt instruction (assuming 50% time halted), the power consumption drops further to 33%.

These are significant power savings, for systems that are under light load conditions at times. The percentage of time that the system is running under reduced load has to be known to predict actual power savings.

system load	4 cpus	2 disks	dvd	pci	vga	dram	net	planar	fans	kbd mou	total
busy	320 90%	32	10	20	5	40	5	30	60	3	525W
light load	310 87%	22	1	15	5	38	5	20	60	3	479W 100%
light load, using frequency reduction	79 90%	22	1	15	5	38	5	20	20	3	208W 43%
light load, using frequency reduction and using hlt 50% of the time	32 40%	22	1	15	5	38	5	20	15	3	156 33%

Table 1: Sample System Power Budget (DC), in watts

7 Hardware—AMD Opteron

7.1 Software Interface To The Hardware

There are two MSRs, the FIDVID_STATUS MSR and the FIDVID_CONTROL MSR, that are used for frequency voltage transitions. These MSRs are the same for the single processor AMD Athlon 64 processors and for the AMD Opteron MP capable processors. These registers are not compatible with the previous generation of AMD Athlon processors, and will not be compatible with the next generation of processors.

The CPU frequency driver for AMD processors therefore has to change across processor revisions, as do the ACPI _PSS objects that describe pstates.

The status register reports the current fid and vid, as well as the maximum fid, the start fid, the maximum vid and the start vid of the particular processor.

These registers are documented in the BKDG[4].

As MSRs can only be accessed by executing code (the read msr or write msr instructions) on the target processor, the frequency driver has to use the processor affinity support to force execution on the correct processor.

7.2 Multiple Memory Controllers

In PC architectures, the memory controller is a component of the northbridge, which is traditionally a separate component from the processor. With AMD Opteron processors, the northbridge is built into the processor. Thus, in a multi-processor system there are multiple memory controllers.

See Figure 1 for a block diagram of a two processor system.

If a processor is accessing DRAM that is physically attached to a different processor, the DRAM access (and any cache coherency traffic) crosses the coherent HyperTransport interprocessor links. There is a small performance penalty in this case. This penalty is of the order of a DRAM page hit versus a DRAM page miss, about 1.7 times slower than a local access.

This penalty is minimized by the processor caches, where data/code residing in remote DRAM is locally cached. It is also minimized by Linux's NUMA support.

Note that a single threaded application that is memory bandwidth constrained may benefit from multiple memory controllers, due to the increase in memory bandwidth.

When the remote processor is transitioned to a lower frequency, this performance penalty is worse. An upper bound to the penalty may be calculated as proportional to the frequency slowdown. I.e., taking the remote processor from 2.2 GHz to 1.0 GHz would take the 1.7 factor from above to a factor of 2.56. Note that this is an absolute worst case, an upper bound to the factor. Actual impact is workload dependent.

A worst case scenario would be a memory bound task, doing memory reads at addresses that are pathologically the worst case for the caches, with all accesses being to remote memory. A more typical scenario would see this penalty alleviated by:

- processor caches, where 64 bytes will be read and cached for a single access, so applications that walk linearly through memory will only see the penalty on 64 byte boundaries,
- memory writes do not take a penalty (as processor execution continues without waiting for a write to complete),
- memory may be interleaved,
- kernel NUMA optimizations for non-interleaved memory (which allocate memory local to the processor when possible to avoid this penalty).

7.3 DRAM Interface Speed

The DRAM interface speed is impacted by the core clock frequency. A full table is published in the processor data sheet; Table 2 shows a sample of actual DRAM frequencies for the common specified DRAM frequencies, across a range of core frequencies.

This table shows that certain DRAM speed / core speed combinations are suboptimal.

Effective memory performance is influenced by many factors:

- cache hit rates,
- effectiveness of NUMA memory allocation routines,
- load on the memory controller,
- size of penalty for remote memory accesses,
- memory speed,
- other hardware related items, such as types of DRAM accesses.

It is therefore necessary to benchmark the actual workload to get meaningful data for that workload.

7.4 UMA

During frequency transitions, and when HyperTransport LDTSTOP is asserted, DRAM is placed into self refresh mode. UMA graphics devices therefore can not access DRAM. UMA systems therefore need to limit the time that DRAM is in self refresh mode. Time constraints are bandwidth dependent, with high resolution displays needing higher memory bandwidth. This is handled by the IRT delay time during frequency transitions. When transitioning multiple steps, the driver waits an appropriate length of time to allow external devices to access memory.

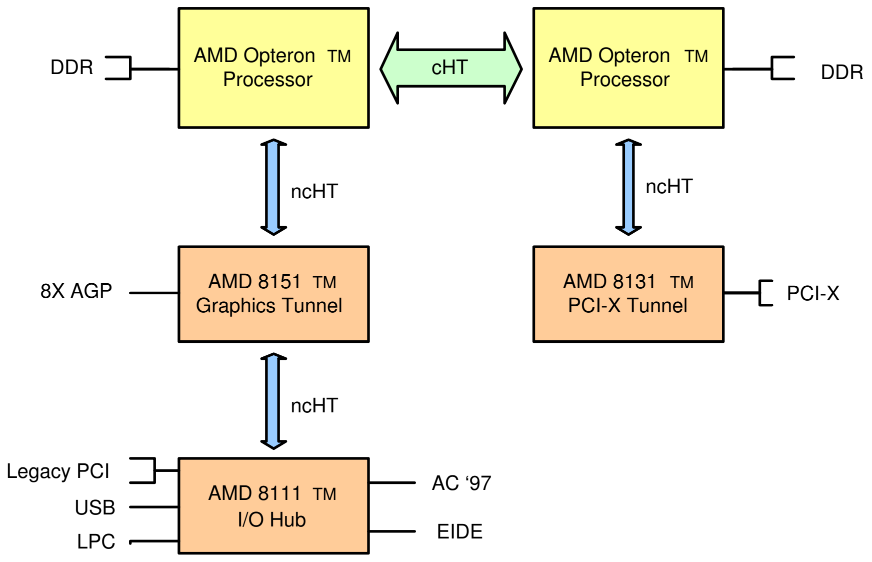


Figure 1: Two Processor System

Processor Core Frequency	100MHz DRAM spec	133MHz DRAM spec	166MHz DRAM spec	200MHz DRAM spec
800MHz	100.00	133.33	160.00	160.00
1000MHz	100.00	125.00	166.66	200.00
2000MHz	100.00	133.33	166.66	200.00
2200MHz	100.00	129.41	157.14	200.00

Table 2: DRAM Frequencies For A Range Of Processor Core Frequencies

7.5 TSC Varying

The Time Stamp Counter (TSC) register is a register that increments with the processor clock. Multiple reads of the register will see increasing values. This register increments on each core clock cycle in the current generation of processors. Thus, the rate of increase of the TSC when compared with "wall clock time" varies as the frequency varies. This causes problems in code that calibrates the TSC increments against an external time source, and then attempts to use the TSC to measure time.

The Linux kernel uses the TSC for such timings, for example when a driver calls udelay(). In this case it is not a disaster if the udelay() call waits for too long as the call is defined to allow this behavior. The case of the udelay() call returning too quickly can be fatal, and this has been demonstrated during experimentation with this code.

This particular problem is resolved by the cpufreq driver correcting the kernel TSC calibration whenever the frequency changes.

This issue may impact other code that uses the TSC register directly. It is interesting to note that it is hard to define a correct behavior. Code that calibrates the TSC against an external clock will be thrown off if the rate of increment of the TSC should change. However, other code may expect a certain code sequence to consistently execute in approximately the same number of cycles, as measured by the TSC, and this code will be thrown off if the behavior of the TSC changes relative to the processor speed.

7.6 Measurement Of Frequency Transition Times

The time required to perform a transition is a combination of the software time to execute the required code, and the hardware time to perform the transition.

Examples of hardware wait time are:

- waiting for the VRM to be stable at a newer voltage,
- waiting for the PLL to lock at the new frequency,
- waiting for DRAM to be placed into and then taken out of self refresh mode around a frequency transition.

The time taken to transition between two states is dependent on both the initial state and the target state. This is due to :

- multiple steps being required in some cases,
- certain operations are lengthier (for example, voltage is stepped up in multiple stages, but stepped down in a single step),
- difference in code execution time dependent on processor speed (although this is minor).

Measurements, taken by calibrating the frequency driver, show that frequency transitions for a processor are taking less than 0.015 seconds.

Further experimentation with multiple processors showed a worst case transition time of less than 0.08 seconds to transition all 4 processors from minimum to maximum frequency, and slightly faster to transition from maximum to minimum frequency.

Note, there is a driver optimization under consideration that would approximately halve these transition times.

7.7 Use of Hardware Enforced Throttling

The southbridge (I/O Hub, example AMD-8111™ HyperTransport I/O Hub) is capable of initiating throttling via the HyperTransport stopclock message, which will ramp down the CPU grid by the programmed amount. This may be initiated by the southbridge for thermal throttling or for other reasons.

This throttling is transparent to software, other than the performance impact.

This throttling is of greatest value in the lowest pstate, due to the reduced voltage.

The hardware enforced throttling is generally not of relevance to the software management of processor frequencies. However, a system designer would need to take care to ensure that the optimal scenarios occur—i.e., transition to a lower frequency/voltage in preference to hardware throttling in high pstates. The BIOS configurations are documented in the BKDG[4].

For maximum power savings, the southbridge would be configured to initiate throttling when the processor executes the `hlt` instruction.

8 Software

The AMD frequency driver is a small part of the software involved. The frequency driver fits into the CPUFreq architecture, which is part of the 2.6 kernel. It is also available as a patch for the 2.4 kernel, and many distributions do include it.

The CPUFreq architecture includes kernel support, the CPUFreq driver itself (`drivers/cpufreq`), an architecture specific driver to control the hardware (powernow-k8.ko is this case), and `/sys` file system code for userland access.

The kernel support code (`linux/kernel/cpufreq.c`) handles timing changes such as updating the kernel constant `loops_per_jiffies`, as well as notifiers (system components that need to be notified of a frequency change).

8.1 History Of The AMD Frequency Driver

The CPU frequency driver for AMD Athlon (the previous generation of processors) was developed by Dave Jones. This driver supports single processor transitions only, as the pstate transition capability was only enabled in mobile processors. This driver used the PSB mechanism to determine valid pstates for the processor. This driver has subsequently been enhanced to add ACPI support.

The initial AMD Athlon 64 and AMD Opteron driver (developed by me, based upon Dave's earlier work, and with much input from Dominik and others), was also PSB based. This was followed by a version of the driver that added ACPI support.

The next release is intended to add a built-in table of pstates that will allow the checking of BIOS supplied data, and also allow an override capability to provide pstate data when not supplied by BIOS.

8.2 User Interface

The deprecated `/proc/cpufreq` (and `/proc/sys`) file system offers control over all processors or individual processors. By echoing values into this file, the root user can change policies and change the limits on available frequencies.

Examples:

Constrain all processors to frequencies between 1.0 GHz and 1.6 GHz, with the performance policy (effectively chooses 1.6 GHz):

```
echo -n "1000000:16000000:
performance" > /proc/cpufreq
```

Constrain processor 2 to run at only 2.0 GHz:

```
echo -n "2:2000000:2000000:
performance" > proc/cpufreq
```

The “performance” refers to a policy, with the other policy available being “powersave.” These policies simply forced the frequency to be at the appropriate extreme of the available range. With the 2.6 kernel, the choice is normally for a “userspace” governor, which allows the (root) user or any user space code (running with root privilege) to dynamically control the frequency.

With the 2.6 kernel, a new interface in the `/sys` filesystem is available to the root user, deprecating the `/proc/cpufreq` method.

The control and status files exist under `/sys/devices/system/cpu/cpuN/cpufreq`, where N varies from 0 upwards, dependent on which processors are online. Among the other files in each processor’s directory, `scaling_min_freq` and `scaling_max_freq` control the minimum and maximum of the ranges in which the frequency may vary. The `scaling_governor` file is used to control the choice of governor. See `linux/Documentation/cpu-freq/userguide.txt` for more information.

Examples:

Constrain processor 2 to run only in the range 1.6 GHz to 2.0 GHz:

```
cd /sys/devices/system/cpu
```

```
cd cpu2/cpufreq
```

```
echo 1600000 > scaling_min_freq
```

```
echo 2000000 > scaling_max_freq
```

8.3 Control From User Space And User Daemons

The interface to the `/sys` filesystem allows userland control and query functionality. Some form of automation of the policy would normally be part of the desired complete implementation.

This automation is dependent on the reason for using frequency management. As an example, for the case of transitioning to a lower pstate when running on a UPS, a daemon will be notified of the failure of mains power, and that daemon will trigger the frequency change by writing to the control files in the `/sys` filesystem.

The CPUFreq architecture has thus split the implementation into multiple parts:

1. user space policy
2. kernel space driver for common functionality
3. kernel space driver for processor specific implementation.

There are multiple user space automation implementations, not all of which currently support multiprocessor systems. One that does, and that has been used in this project is cpufreqd version 1.1.2 (`http://sourceforge.net/projects/cpufreqd`).

This daemon is controlled by a configuration file. Other than making changes to the configuration file, the author of this paper has not been involved in any of the development work on cpufreqd, and is a mere user of this tool.

The configuration file specifies profiles and rules. A profile is a description of the system settings in that state, and my configuration file is setup to map the profiles to the processor pstates. Rules are used to dynamically choose which profile to use, and my rules are setup to transition profiles based on total processor load.

My simple configuration file to change processor frequency dependent on system load is:

```
[General]
pidfile=/var/run/cpufreqd.pid
poll_interval=2
pm_type=acpi

# 2.2 GHz processor speed
[Profile]
name=hi_boost
minfreq=95%
maxfreq=100%
policy=performance

# 2.0 GHz processor speed
[Profile]
name=medium_boost
minfreq=90%
maxfreq=93%
policy=performance

# 1.0 GHz processor Speed
[Profile]
name=lo_boost
minfreq=40%
maxfreq=50%
policy=powersave

[Profile]
name=lo_power
minfreq=40%
maxfreq=50%
policy=powersave

[Rule]
#not busy 0%-40%
name=conservative
ac=on
battery_interval=0-100
cpu_interval=0-40
profile=lo_boost

#medium busy 30%-80%
[Rule]
name=lo_cpu_boost
ac=on
battery_interval=0-100
cpu_interval=30-80
profile=medium_boost

#really busy 70%-100%
[Rule]
name=hi_cpu_boost
ac=on
battery_interval=50-100
cpu_interval=70-100
profile=hi_boost
```

This approach actually works very well for multiple small tasks, for transitioning the frequencies of all the processors together based on a collective loading statistic.

For a long running, single threaded task, this approach does not work well as the load is only high on a single processor, with the others being idle. The average load is thus low, and all processors are kept at a slow speed. Such a workload scenario would require an implementation that looked at the loading of individual processors, rather than the average. See the section below on future work.

8.4 The Drivers Involved

`powernow-k8.ko` `arch/i386/kernel/cpu/cpufreq/powernow-k8.c` (the same source code is built as a 32-bit driver in the `i386` tree and as a 64-bit driver in the `x86_64` tree)

`drivers/acpi`

`drivers/cpufreq`

The Test Driver

Note that the `powernow-k8.ko` driver does not export any read, write, or ioctl interfaces. For test purposes, a second driver exists with an ioctl interface for test application use. The test driver was a big part of the test effort on `powernow-k8.ko` prior to release.

8.5 Frequency Driver Entry Points

powernowk8_init()

Driver `late_initcall`. Initialization is late as the acpi driver needs to be initialized first. Verifies that all processors in the system are capable of frequency transitions, and that all processors are supported processors. Builds a data structure with the addresses of the four entry points for cpufreq use (listed below), and calls `cpufreq_register_driver()`.

powernowk8_exit()

Called when the driver is to be unloaded. Calls `cpufreq_unregister_driver()`.

8.6 Frequency Driver Entry Points For Use By The CPUFreq driver

powernowk8_cpu_init()

This is a per-processor initialization routine. As we are not guaranteed to be executing on the processor in question, and as the driver needs access to MSRs, the driver needs to force itself to run on the correct processor by using `set_cpus_allowed()`.

This pre-processor initialization allows for processors to be taken offline or brought online dynamically. I.e., this is part of the software support that would be needed for processor hotplug, although this is not supported in the hardware.

This routine finds the ACPI pstate data for this processor, and extracts the (proprietary) data from the ACPI `_PSS` objects. This data is verified as far as is reasonable. Per-processor data tables for use during frequency transitions are constructed from this information.

powernowk8_cpu_exit()

Per-processor cleanup routine.

powernowk8_verify()

When the root user (or an application running on behalf of the root user) requests a change to the minimum/maximum frequencies, or to the policy or governor, the frequency driver's verification routine is called to verify (and correct if necessary) the input values. For example, if the maximum speed of the processor is 2.4 GHz and the user requests that the maximum range be set to 3.0 GHz, the verify routine will correct the maximum value to a value that is actually possible. The user can, however, chose a value that is less than the hardware maximum, for example 2.0 GHz in this case.

As this routine just needs to access the per-processor data, and not any MSRs, it does not matter which processor executes this code.

powernowk8_target()

This is the driver entry point that actually performs a transition to a new frequency/voltage. This entry point is called for each processor that needs to transition to a new frequency.

There is therefore an optimization possible by enhancing the interface between the frequency driver and the CPUFreq driver for the case where all processors are to be transitioned to a new, common frequency. However, it is not clear that such an optimization is worth the complexity, as the functionality to transition a single processor would still be needed.

This routine is invoked with the processor number as a parameter, and there is no guarantee as to which processor we are currently executing on. As the mechanism for changing the frequency involves accessing MSRs, it is necessary to execute on the target processor, and the driver forces its execution onto the target processor by using `set_cpus_allowed()`.

The CPUFreq helpers are then used to determine the correct target frequency. Once a chosen target `fid` and `vid` are identified:

- the cpufreq driver is called to warn that a transition is about to occur,
- the actual transition code within powernow-k8 is called, and then
- the cpufreq driver is called again to confirm that the transition was successful.

The actual transition is protected with a semaphore that is used across all processors. This is to prevent transitions on one processor from interfering with transitions on other processors. This is due to the inter-processor communication that occurs at a hardware level when a frequency transition occurs.

8.7 CPUFreq Interface

The CPUFreq interface provides entry points, that are required to make the system function.

It also provides helper functions, which need not be used, but are there to provide common functionality across the set of all architecture specific drivers. Elimination of duplicate good is a good thing! An architecture specific driver can build a table of available frequencies, and pass this table to the CPUFreq driver. The helper functions then simplify the architecture driver code by manipulating this table.

cpufreq_register_driver()

Registers the frequency driver as being the driver capable of performing frequency transitions on this platform. Only one driver may be registered.

cpufreq_unregister_driver()

Unregisters the driver, when it is being unloaded.

cpufreq_notify_transition()

Used to notify the CPUFreq driver, and thus the kernel, that a frequency transition is occurring, and triggering recalibration of timing specific code.

cpufreq_frequency_table_target()

Helper function to find an appropriate table entry for a given target frequency. Used in the driver's target function.

cpufreq_frequency_table_verify()

Helper function to verify that an input frequency is valid. This helper is effectively a complete implementation of the driver's verify function.

cpufreq_frequency_table_cpuinfo()

Supplies the frequency table data that is used on subsequent helper function calls. Also aids with providing information as to the capabilities of the processors.

8.8 Calls To The ACPI Driver

acpi_processor_register_performance()

acpi_processor_unregister_performance()

Helper functions used at per-processor initialization time to gain access to the data from the _PSS object for that processor. This is a preferable solution to the frequency driver having to walk the ACPI namespace itself.

8.9 The Single Processor Solution

Many of the kernel system calls collapse to constants when the kernel is built without multiprocessor support. For example, `num_online_cpus()` becomes a macro with the value 1. By the careful use of the definitions in smp.h, the same driver code handles both multiprocessor and single processor machines without the use of conditional compilation. The multiprocessor support obviously adds complexity to the code for a single processor code, but this code is negligible in the case of transitioning frequencies. The driver initialization and termination code is made more complex and lengthy, but this is not frequently executed code. There is also a small penalty in terms of code space.

The author does not feel that the penalty of the multiple processor support code is noticeable on a single processor system, but this is obviously debatable. The current choice is to have a single driver that supports both single processor and multiple processor systems.

As the primary performance cost is in terms of additional code space, it is true that a single processor machine with highly constrained memory may benefit from a simplified driver without the additional multi-processor support code. However, such a machine would see greater benefit by eliminating other code that would not be necessary on a chosen platform. For example, the PSB support code could be removed from a memory constrained single processor machine that was using ACPI.

This approach of removing code unnecessary for a particular platform is not a wonderful approach when it leads to multiple variants of the driver, all of which have to be supported and enhanced, and which makes Kconfig even more complex.

8.10 Stages Of Development, Test And Debug Of The Driver

The algorithm for transitioning to a new frequency is complex. See the BKDG[4] for a good description of the steps required, including flowcharts. In order to test and debug the frequency/voltage transition code thoroughly, the author first wrote a simple simulation of the processor. This simulation maintained a state machine, verified that fid/vid MSR control activity was legal, provided fid/vid status MSR results, and wrote a log file of all activity. The core driver code was then written as an application and linked with this simulation code to allow testing of all combinations.

The driver was then developed as a skeleton using printk to develop and test the BIOS/ACPI interfaces without having the frequency/voltage transition code present. This is because attempts to actually transition to an invalid pstate often result in total system lockups that offer no debug output—if the processor voltage is too low for the frequency, successful code execution ceases.

When the skeleton was working correctly, the actual transition code was dropped into place, and tested on real hardware, both single processor and multiple processor. (The single processor driver was released many months before the multi-processor capable driver as the multiprocessor capable hardware was not available in the marketplace.) The functional driver was tested, using printk to trace activity, and using external hardware to track power usage, and using a test driver to independently verify register settings.

The functional driver was then made available to various people in the community for their feedback. The author is grateful for the extensive feedback received, which included the changed code to implement suggestions. The driver as it exists today is considerably improved from the initial release, due to this feedback mechanism.

9 How To Determine Valid PStates For A Given Processor

AMD defines pstates for each processor. A performance state is a frequency/voltage pair that is valid for operation of that processor. These are specified as fid/vid (frequency identifier/voltage identifier values) pairs, and are documented in the Processor Thermal and Data Sheets (see references). The worst case processor power consumption for each pstate is also characterized. The BKDG[4] contains tables for mapping fid to frequency and vid to voltage.

Pstates are processor specific. I.e., 2.0 GHz at 1.45V may be correct for one model/revision of processor, but is not necessarily correct for a different/revision model of processor.

Code can determine whether a processor supports or does not support pstate transitions by executing the cpuid instruction. (For details, see the BKDG[4] or the source code for the Linux frequency driver). This needs to be done for each processor in an MP system.

Each processor in an MP system could theoretically have different pstates.

Ideally, the processor frequency driver would not contain hardcoded pstate tables, as the driver would then need to be revised for new processor revisions. The chosen solution is to have the BIOS provide the tables of pstates, and have the driver retrieve the pstate data from the BIOS. There are two such tables defined for use by BIOSs for AMD systems:

1. PSB, AMD's original proprietary mechanism, which does not support MP. This mechanism is being deprecated.
2. ACPI `_PSS` objects. Whereas the ACPI specification is a standard, the data within the `_PSS` objects is AMD specific (and, in fact, processor family specific), and thus there is still a proprietary nature of this solution.

The current AMD frequency driver obtains data from the ACPI objects. ACPI does introduce some limitations, which are discussed later. Experimentation is ongoing with a built-in database approach to the problem in an attempt to bypass these issues, and also to allow checking of validity of the ACPI provided data.

10 ACPI And Frequency Restrictions

ACPI[5] provides the `_PPC` object, that is used to constrain the pstates available. This object is dynamic, and can therefore be used in platforms for purposes such as:

- forcing frequency restrictions when operating on battery power,
- forcing frequency restrictions due to thermal conditions.

For battery / mains power transitions, an ACPI-compliant GPE (General Purpose Event) input to the chipset (I/O hub) is dedicated to assigning a SCI (System Control Interrupt) when the power source changes. The ACPI driver will then execute the ACPI control method (see the `_PSR` power source ACPI object), which issues a notify to the `_CPUn` object, which triggers the ACPI driver to re-evaluate the `_PPC` object. If the current pstate exceeds that allowed by this new evaluation of the `_PPC` object, the CPU frequency driver will be called to transition to a lower pstate.

11 ACPI Issues

ACPI as a standard is not perfect. There is variation among different implementations, and Linux ACPI support does not work on all machines.

ACPI does introduce some overhead, and some users are not willing to enable ACPI.

ACPI requires that pstates be of equivalent power usage and frequency across all processors. In a system with processors that are capable of different maximum frequencies (for example, one processor capable of 2.0 GHz and a second processor capable of 2.2 GHz), compliance with the ACPI specification means that the faster processor(s) will be restricted to the maximum speed of the slowest processor. Also, if one processor has 5 available pstates, the presence of processor with only 4 available pstates will restrict all processors to 4 pstates.

12 What Is There Today?

AMD is shipping pstate capable AMD Opteron processors (revision CG). Server processors prior to revision CG were not pstate capable. All AMD Athlon 64 processors for mobile and desktop are pstate capable.

BKDG[4] enhancements to describe the capability are in progress.

AMD internal BIOSs have the enhancements. These enhancements are rolling out to the publicly available BIOSs along with the BKDG enhancements.

The multi-processor capable Linux frequency driver has released under GPL.

The cpufreqd user-mode daemon, available for download from `http://sourceforge.net/projects/cpufreqd` supports multiple processors.

13 Other Software-directed Power Saving Mechanisms

13.1 Use Of The `HLT` Instruction

The hlt instruction is normally used when the operating system has no code for the processor to execute. This is the ACPI C1 state. Execution of instructions ceases, until the processor is restarted with an interrupt. The power savings are maximized when the hlt state is entered in the minimum pstate, due to the lower voltage. The alternative to the use of the hlt instruction is a do nothing loop.

13.2 Use of Power Managed Chipset Drivers

Devices on the planar board, such as a PCI-X bridge or an AGP tunnel, may have the capability to operate in lower power modes. Entering and leaving the lower power modes is under the control of the driver for that device.

Note that HyperTransport attached devices can transition themselves to lower power modes when certain messages are seen on the bus. However, this functionality is typically configurable, so a chipset driver (or the system BIOS during bootup) would need to enable this capability.

14 Items For Future Exploration

14.1 A Built-in Database

The theory is that the driver could have a built-in database of processors and the pstates that they support. The driver could then use this database to obtain the pstate data without dependencies on ACPI, or use it for enhanced checking of the ACPI provided data. The disadvantage of this is the need to update the database for new processor revisions. The advantages are the ability to overcome the ACPI imposed restrictions, and also to allow the use of the technology on systems where the ACPI support is not enabled.

14.2 Kernel Scheduler—CPU Power

An enhanced scheduler for the 2.6 kernel (2.6.6-bk1) is aware of groups of processors with different processing power. The power tating of each CPU group should be dynamically adjusted using a cpufreq transition notifier as the processor frequencies are changed.

See `http://lwn.net/Articles/80601/` for a detailed acount of the scheduler changes.

14.3 Thermal Management, ACPI Thermal Zones

Publicly available BIOSs for AMD machines do not implement thermal zones. Obviously this is one way to provide the input control for frequency management based on thermal conditions.

14.4 Thermal Management, Service Processor

Servers typically have a service processor, which may be compliant to the IPMI specification. This service processor is able to accurately monitor temperature at different locations within the chassis. The 2.6 kernel includes an IPMI driver. User space code may use these thermal readings to control fan speeds and generate administrator alerts. It may make sense to also use these accurate thermal readings to trigger frequency transitions.

The interaction between thermal events from the service processor and ACPI thermal zones may be a problem.

Hiding Thermal Conditions

One concern with the use of CPU frequency manipulation to avoid overheating is that hardware problems may not be noticed. Over temperature conditions would normally cause administrator alerts, but if the processor is first taken to a lower frequency to hold temperature down, then the alert may not be generated. A failing fan (not spinning at full speed) could therefore be missed. Some hardware components fail gradually, and early warning of imminent failures is needed to perform planned maintenance. Losing this data would be badness.

15 Legal Information

16 References

1. AMD Opteron™ Processor Data Sheet, publication 23932, available from `www.amd.com`

2. AMD Opteron™ Processor Power And

Thermal Data Sheet, publication 30417, available from `www.amd.com`

3. AMD Athlon™ 64 Processor Power And Thermal Data Sheet, publication 30430, available from `www.amd.com`

4. BIOS and Kernel Developer's Guide (the BKDG) for AMD Athlon™ 64 and AMD Opteron™ Processors, publication 26094, available from `www.amd.com`. Chapter 9 covers frequency management.

5. ACPI 2.0b Specification, from `www.acpi.info`

6. Text documentation files in the kernel `linux/Documentation/cpu-freq/` directory:

   - `index.txt`
   - `user-guide.txt`
   - `core.txt`
   - `cpu-drivers.txt`
   - `governors.txt`

Dynamic Kernel Module Support: From Theory to Practice

Matt Domsch & Gary Lerhaupt
Dell Linux Engineering
`Matt_Domsch@dell.com, Gary_Lerhaupt@dell.com`

Abstract

DKMS is a framework which allows individual kernel modules to be upgraded without changing your whole kernel. Its primary audience is fourfold: system administrators who want to update a single device driver rather than wait for a new kernel from elsewhere with it included; distribution maintainers, who want to release a single targeted bugfix in between larger scheduled updates; system manufacturers who need single modules changed to support new hardware or to fix bugs, but do not wish to test whole new kernels; and driver developers, who must provide updated device drivers for testing and general use on a wide variety of kernels, as well as submit drivers to kernel.org.

Since OLS2003, DKMS has gone from a good idea to deployed and used. Based on end user feedback, additional features have been added: precompiled module tarball support to speed factory installation; driver disks for Red Hat distributions; 2.6 kernel support; SuSE kernel support. Planned features include cross-architecture build support and additional distribution driver disk methods.

In addition to overviewing DKMS and its features, we explain how to create a dkms.conf file to DKMS-ify your kernel module source.

1 History

Historically, Linux distributions bundle device drivers into essentially one large kernel package, for several primary reasons:

- Completeness: The Linux kernel as distributed on kernel.org includes all the device drivers packaged neatly together in the same kernel tarball. Distro kernels follow kernel.org in this respect.
- Maintainer simplicity: With over 4000 files in the kernel `drivers/` directory, each possibly separately versioned, it would be impractical for the kernel maintainer(s) to provide a separate package for each driver.
- Quality Assurance / Support organization simplicity: It is easiest to ask a user "what kernel version are you running," and to compare this against the list of approved kernel versions released by the QA team, rather than requiring the customer to provide a long and extensive list of package versions, possibly one per module.
- End user install experience: End users don't care about which of the 4000 possible drivers they need to install, they just want it to work.

This works well as long as you are able to make the "top of the tree" contain the most current and most stable device driver, and you are able to convince your end users to always run the "top of the tree." The `kernel.org` development processes tend to follow this model with great success.

But widely used distros cannot ask their users to always update to the top of the kernel.org tree. Instead, they start their products from the top of the kernel.org tree at some point in time, essentially freezing with that, to begin their test cycles. The duration of these test cycles can be as short as a few weeks, and as long as a few years, but 3-6 months is not unusual. During this time, the kernel.org kernels march forward, and some (but not all) of these changes are backported into the distro's kernel. They then apply the minimal patches necessary for them to declare the product finished, and move the project into the sustaining phase, where changes are very closely scrutinized before releasing them to the end users.

1.1 Backporting

It is this sustaining phase that DKMS targets. DKMS can be used to backport newer device driver versions from the "top of the tree" kernels where most development takes place to the now-historical kernels of released products.

The `PATCH_MATCH` mechanism was specifically designed to allow the application of patches to a "top of the tree" device driver to make it work with older kernels. This allows driver developers to continue to focus their efforts on keeping kernel.org up to date, while allowing that same effort to be used on existing products with minimal changes. See Section 6 for a further explanation of this feature.

1.2 Driver developers' packaging

Driver developers have recognized for a long time that they needed to provide backported versions of their drivers to match their end users' needs. Often these requirements are imposed on them by system vendors such as Dell in support of a given distro release. However, each driver developer was free to provide the backport mechanism in any way they chose. Some provided architecture-specific RPMs which contained only precompiled modules. Some provided source RPMs which could be rebuilt for the running kernel. Some provided driver disks with precompiled modules. Some provided just source code patches, and expected the end user to rebuild the kernel themselves to obtain the desired device driver version. All provided their own Makefiles rather than use the kernel-provided build system.

As a result, different problems were encountered with each developers' solution. Some developers had not included their drivers in the kernel.org tree for so long that that there were discrepancies, e.g. `CONFIG_SMP` vs `__SMP__`, `CONFIG_2G` vs. `CONFIG_3G`, and compiler option differences which went unnoticed and resulted in hard-to-debug issues.

Needless to say, with so many different mechanisms, all done differently, and all with different problems, it was a nightmare for end users.

A new mechanism was needed to cleanly handle applying updated device drivers onto an end user's system. Hence DKMS was created as the one module update mechanism to replace all previous methods.

2 Goals

DKMS has several design goals.

- Implement only mechanism, not policy.
- Allow system administrators to easily know what modules, what versions, for

what kernels, and in what state, they have on their system.

- Keep module source as it would be found in the "top of the tree" on kernel.org. Apply patches to backport the modules to earlier kernels as necessary.

- Use the kernel-provided build mechanism. This reduces the Makefile magic that driver developers need to know, thus the likelihood of getting it wrong.

- Keep additional DKMS knowledge a driver developer must have to a minimum. Only a small per-driver dkms.conf file is needed.

- Allow multiple versions of any one module to be present on the system, with only one active at any given time.

- Allow DKMS-aware drivers to be packaged in the Linux Standard Base-conformant RPM format.

- Ease of use by multiple audiences: driver developers, system administrators, Linux distros, and system vendors.

We discuss DKMS as it applies to each of these four audiences.

3 Distributions

All present Linux distributions distribute device drivers bundled into essentially one large kernel package, for reasons outlined in Section 1. It makes the most sense, most of the time.

However, there are cases where it does not make sense.

- Severity 1 bugs are discovered in a single device driver between larger scheduled updates. Ideally you'd like your affected users to be able to get the single module update without having to release and Q/A a whole new kernel. Only customers who are affected by the particular bug need to update "off-cycle."

- Solutions vendors, for change control reasons, often certify their solution on a particular distribution, scheduled update release, and sometimes specific kernel version. The latter, combined with releasing device driver bug fixes as whole new kernels, puts the customer in the untenable position of either updating to the new kernel (and losing the certification of the solution vendor), or forgoing the bug fix and possibly putting their data at risk.

- Some device drivers are not (yet) included in kernel.org nor a distro kernel, however one may be required for a functional software solution. The current support models require that the add-on driver "taint" the kernel or in some way flag to the support organization that the user is running an unsupported kernel module. Tainting, while valid, only has three dimensions to it at present: Proprietary—non-GPL licensed; Forced—loaded via `insmod -f`; and Unsafe SMP—for some CPUs which are not designed to be SMP-capable. A GPL-licensed device driver which is not yet in kernel.org or provided by the distribution may trigger none of these taints, yet the support organization needs to be aware of this module's presence. To avoid this, we expect to see the distros begin to cryptographically sign kernel modules that they produce, and taint on load of an unsigned module. This would help reduce the support organization's work for calls about "unsupported" configurations. With DKMS in use, there is less a need for such methods, as it's easy to see which modules have been changed.

Note: this is not to suggest that driver authors should not submit their drivers to `kernel.org`*—absolutely they should.*

- The distro QA team would like to test updates to specific drivers without waiting for the kernel maintenance team to rebuild the kernel package (which can take many hours in some cases). Likewise, individual end users may be willing (and often be required, e.g. if the distro QA team can't reproduce the users's hardware and software environment exactly) to show that a particular bug is fixed in a driver, prior to releasing the fix to *all* of that distro's users.

- New hardware support via driver disks: Hardware vendors release new hardware asynchronously to any software vendor schedule, no matter how hard companies may try to synchronize releases. OS distributions provide install methods which use driver diskettes to enable new hardware for previously-released versions of the OS. Generating driver disks has always been a difficult and error-prone procedure, different for each OS distribution, not something that the casual end-user would dare attempt.

DKMS was designed to address all of these concerns.

DKMS aims to provide a clear separation between mechanism (how one updates individual kernel modules and tracks such activity) and policy (when should one update individual kernel modules).

3.1 Mechanism

DKMS provides only the mechanism for updating individual kernel modules, not policy. As such, it can be used by distributions (per their policy) for updating individual device drivers for individual users affected by Severity 1 bugs, without releasing a whole new kernel.

The first mechanism critical to a system administrator or support organization is the `status` command, which reports the name, version, and state of each kernel module under DKMS control. By querying DKMS for this information, system administrators and distribution support organizations may quickly understand when an updated device driver is in use to speed resolution when issues are seen.

DKMS's ability to generate driver diskettes gives control to both novice and seasoned system administrators alike, as they can now perform work which otherwise they would have to wait for a support organization to do for them. They can get their new hardware systems up-and-running quickly by themselves, leaving the support organizations with time to do other more interesting value-added work.

3.2 Policy

Suggested policy items include:

- Updates must pass QA. This seems obvious, but it reduces broken updates (designed to fix other problems) from being released.

- Updates must be submitted, and ideally be included already, upstream. For this we expect kernel.org and the OS distribution to include the update in their next larger scheduled update. This ensures that when the next kernel.org kernel or distro update comes out, the short-term fix provided via DKMS is incorporated already.

- The `AUTOINSTALL` mechanism is set to `NO` for all modules which are shipped with the target distro's kernel. This prevents the DKMS autoinstaller from installing a (possibly older) kernel module onto a newer kernel without being explicitly told to do so by the system administrator. This follows from the "all DKMS updates must be in the next larger release" rule above.

- All issues for which DKMS is used are tracked in the appropriate bug tracking databases until they are included upstream, and are reviewed regularly.

- All DKMS packages are provided as DKMS-enabled RPMs for easy installation and removal, per the Linux Standard Base specification.

- All DKMS packages are posted to the distro's support web site for download by system administrators affected by the partiular issue.

4 System Vendors

DKMS is useful to System Vendors such as Dell for many of the same reasons it's useful to the Linux distributions. In addition, system vendors face additional issues:

- Critical bug fixes for distro-provided drivers: While we hope to never need such, and we test extensively with distro-provided drivers, occasionally we have discovered a critical bug after the distribution has cut their gold CDs. We use DKMS to update just the affected device drivers.

- Alternate drivers: Dell occasionally needs to provide an alternate driver for a piece of hardware rather than that provided by the distribution natively. For example, Dell provides the Intel iANS network channel bonding and failover driver for customers who have used iANS in the past, and wish to continue using it rather than upgrading to the native channel bonding driver resident in the distribution.

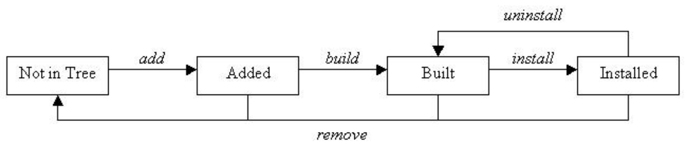
- Factory installation: Dell installs various OS distribution releases onto new hardware in its factories. We try not to update from the gold release of a distribution version to any of the scheduled updates, as customers expect to receive gold. We use DKMS to enable newer device drivers to handle newer hardware than was supported natively in the gold release, while keeping the gold kernel the same.

We briefly describe the policy Dell uses, in addition to the above rules suggested to OS distributions:

- Prebuilt DKMS tarballs are required for factory installation use, for all kernels used in the factory install process. This prevents the need for the compiler to be run, saving time through the factories. Dell rarely changes the factory install images for a given OS release, so this is not a huge burden on the DKMS packager.

- All DKMS packages are posted to support.dell.com for download by system administrators purchasing systems without Linux factory-installed.

Figure 1: DKMS state diagram.



5 System Administrators

5.1 Understanding the DKMS Life Cycle

Before diving into using DKMS to manage kernel modules, it is helpful to understand the life cycle by which DKMS maintains your kernel modules. In Figure 1, each rectangle represents a state your module can be in and each italicized word represents a DKMS action that can used to switch between the various DKMS states. In the following section we will look further into each of these DKMS actions and then continue on to discuss auxiliary DKMS functionality that extends and improves upon your ability to utilize these basic commands.

5.2 RPM and DKMS

DKMS was designed to work well with Red Hat Package Manger (RPM). Many times using DKMS to install a kernel module is as easy as installing a DKMS-enabled module RPM. Internally in these RPMs, DKMS is used to `add`, `build`, and `install` a module. By wrapping DKMS commands inside of an RPM, you get the benefits of RPM (package versioning, security, dependency resolution, and package distribution methodologies) while DKMS handles the work RPM does not, versioning and building of individual kernel modules. For reference, a sample DKMS-enabled RPM specfile can be found in the DKMS package.

5.3 Using DKMS

5.3.1 Add

DKMS manages kernel module versions at the source code level. The first requirement of using DKMS is that the module source be located on the build system and that it be located in the directory `/usr/src/<module>-<module-version>/`. It also requires that a dkms.conf file exists with the appropriately formatted directives within this configuration file to tell DKMS such things as where to install the module and how to build it. Once these two requirements have been met and DKMS has been installed on your system, you can begin using DKMS by adding a module/module-version to the DKMS tree. For example:

```
# dkms add -m megaraid2 -v 2.10.3
```

This example `add` command would add megaraid2/2.10.3 to the already existent `/var/dkms` tree, leaving it in the Added state.

5.3.2 Build

Once in the Added state, the module is ready to be built. This occurs through the DKMS `build` command and requires that the proper kernel sources are located on the system from the `/lib/module/<kernel-version>/build` symlink. The make command that is used to compile the module is specified in the dkms.conf configuration file. Continuing with the megaraid2/2.10.3 example:

```
# dkms build -m megaraid2
   -v 2.10.3 -k 2.4.21-4.ELsmp
```

The `build` command compiles the module but stops short of installing it. As can be seen in the above example, `build` expects a kernel-version parameter. If this kernel name is left out, it assumes the currently running kernel. However, it functions perfectly well to build modules for kernels that are not currently running. This functionality is assured through use of a kernel preparation subroutine that runs before any module build is performed in order to ensure that the module being built is linked against the proper kernel symbols.

Successful completion of a `build` creates, for this example, the `/var/dkms/megaraid2/2.10.3/2.4.21-4.ELsmp/` directory as well as the log and module subdirectories within this directory. The log directory holds a log file of the module make and the module directory holds copies of the resultant binaries.

5.3.3 Install

With the completion of a `build`, the module can now be installed on the kernel for which it was built. Installation copies the compiled module binary to the correct location in the `/lib/modules/` tree as specified in the dkms.conf file. If a module by that name is already found in that location, DKMS saves it in its tree as an original module so at a later time it can be put back into place if the newer module is uninstalled. An example `install` command:

```
# dkms install -m megaraid2
   -v 2.10.3 -k 2.4.21-4.ELsmp
```

If a module by the same name is already installed, DKMS saves a copy in its tree and does so in the `/var/dkms/<module-name>/original_module/` directory. In this case, it would be saved to `/var/dkms/megaraid2/original_module/2.4.21-4.ELsmp/`.

5.3.4 Uninstall and Remove

To complete the DKMS cycle, you can also uninstall or remove your module from the tree. The `uninstall` command deletes from `/lib/modules` the module you installed and, if applicable, replaces it with its original module. In scenarios where multiple versions of a module are located within the DKMS tree, when one version is uninstalled, DKMS does not try to understand or assume which of these other versions to put in its place. Instead, if a true "original_module" was saved from the very first DKMS installation, it will be put back into the kernel and all of the other module versions for that module will be left in the Built state. An example `uninstall` would be:

```
# dkms uninstall -m megaraid2
   -v 2.10.3 -k 2.4.21-4.ELsmp
```

Again, if the kernel version parameter is unset, the currently running kernel is assumed, although, the same behavior does not occur with the `remove` command. The `remove` and `uninstall` are very similar in that `remove` will do all of the same steps as `uninstall`. However, when `remove` is employed, if the module-version being removed is the last instance of that module-version for all kernels on your system, after the uninstall portion of the remove completes, it will delete all traces of that module from the DKMS tree. To put it another way, when an `uninstall` command completes, your modules are left in the Built state. However, when a `remove` completes, you would be left in the Not in Tree state. Here are two sample `remove` commands:

```
# dkms remove -m megaraid2
   -v 2.10.3 -k 2.4.21-4.ELsmp
# dkms remove -m megaraid2
   -v 2.10.3 --all
```

With the first example `remove` command, your module would be uninstalled and if this module/module-version were not installed on any other kernel, all traces of it would be removed from the DKMS tree all together. If, say, megaraid2/2.10.3 was also installed on the 2.4.21-4.ELhugemem kernel, the first `remove` command would leave it alone and it would remain intact in the DKMS tree. In the second example, that would not be the case. It would uninstall all versions of the megaraid2/2.10.3 module from all kernels and then completely expunge all references of megaraid2/2.10.3 from the DKMS tree. Thus, `remove` is what cleans your DKMS tree.

5.4 Miscellaneous DKMS Commands

5.4.1 Status

DKMS also comes with a fully functional status command that returns information about what is currently located in your tree. If no parameters are set, it will return all information found. Logically, the specificity of information returned depends on which parameters are passed to your status command. Each status entry returned will be of the state: "added," "built," or "installed," and if an original module has been saved, this information will also be displayed. Some example status commands include:

```
# dkms status
# dkms status -m megaraid2
# dkms status -m megaraid2 -v 2.10.3
# dkms status -k 2.4.21-4.ELsmp
# dkms status -m megaraid2
   -v 2.10.3 -k 2.4.21-4.ELsmp
```

5.4.2 Match

Another major feature of DKMS is the match command. The match command takes the configuration of DKMS installed modules for one kernel and applies this same configuration to some other kernel. When the match completes, the same module/module-versions that were installed for one kernel are also then installed on the other kernel. This is helpful when you are upgrading from one kernel to the next, but would like to keep the same DKMS modules in place for the new kernel. Here is an example:

```
# dkms match
   --templatekernel 2.4.21-4.ELsmp
   -k 2.4.21-5.ELsmp
```

As can be seen in the example, the `−−templatekernel` is the "match-er" kernel from which the configuration is based, while the `-k kernel` is the "match-ee" upon which the configuration is instated.

5.4.3 dkms_autoinstaller

Similar in nature to the match command is the dkms_autoinstaller service. This service gets installed as part of the DKMS RPM in the /etc/init.d directory. Depending on whether an `AUTOINSTALL` directive is set within a module's dkms.conf configuration file, the dkms_autoinstaller will automatically build and install that module as you boot your system into new kernels which do not already have this module installed.

5.4.4 mkdriverdisk

The last miscellaneous DKMS command is `mkdriverdisk`. As can be inferred from its name, `mkdriverdisk` will take the proper sources in your DKMS tree and create a driver disk image for use in providing updated drivers to Linux distribution installations. A sample `mkdriverdisk` might look like:

```
# dkms mkdriverdisk -d redhat
   -m megaraid2 -v 2.10.3
   -k 2.4.21-4.ELBOOT
```

Currently, the only supported distribution driver disk format is Red Hat. For more information on the extra necessary files and their formats for DKMS to create Red Hat driver disks, see `http://people.redhat.com/dledford`. These files should be placed in your module source directory.

5.5 Systems Management with DKMS Tarballs

As we have seen, DKMS provides a simple mechanism to build, install, and track device driver updates. So far, all these actions have related to a single machine. But what if you've got many similar machines under your administrative control? What if you have a compiler and kernel source on only one system (your master build system), but you need to deploy your newly built driver to all your other systems? DKMS provides a solution to this as well—in the `mktarball` and `ldtarball` commands.

The `mktarball` command rolls up copies of each device driver module file which you've built using DKMS into a compressed tarball. You may then copy this tarball to each of your target systems, and use the DKMS `ldtarball` command to load those into your DKMS tree, leaving each module in the Built state, ready to be installed. This avoids the need for both kernel source and compilers to be on every target system.

For example:

You have built the megaraid2 device driver, version 2.10.3, for two different kernel families (here 2.4.20-9 and 2.4.21-4.EL), on your master build system.

```
# dkms status
megaraid2, 2.10.3, 2.4.20-9: built
megaraid2, 2.10.3, 2.4.20-9bigmem: built
megaraid2, 2.10.3, 2.4.20-9BOOT: built
megaraid2, 2.10.3, 2.4.20-9smp: built
megaraid2, 2.10.3, 2.4.21-4.EL: built
megaraid2, 2.10.3, 2.4.21-4.ELBOOT: built
megaraid2, 2.10.3, 2.4.21-4.ELhugemem: built
megaraid2, 2.10.3, 2.4.21-4.ELsmp: built
```

You wish to deploy this version of the driver to several systems, without rebuilding from source each time. You can use the `mktarball` command to generate one tarball for each kernel family:

```
# dkms mktarball -m megaraid2
   -v 2.10.3
   -k 2.4.21-4.EL,2.4.21-4.ELsmp,
   2.4.21-4.ELBOOT,2.4.21-4.ELhugemem

Marking /usr/src/megaraid2-2.10.3 for archiving...
Marking kernel 2.4.21-4.EL for archiving...
Marking kernel 2.4.21-4.ELBOOT for archiving...
Marking kernel 2.4.21-4.ELhugemem for archiving...
Marking kernel 2.4.21-4.ELsmp for archiving...
Tarball location:
  /var/dkms/megaraid2/2.10.3/tarball/
megaraid2-2.10.3-manykernels.tgz
Done.
```

You can make one big tarball containing modules for both families by omitting the -k argument and kernel list; DKMS will include a module for every kernel version found.

You may then copy the tarball (renaming it if you wish) to each of your target systems using any mechanism you wish, and load the modules in. First, see that the target DKMS tree does not contain the modules you're loading:

```
# dkms status
Nothing found within the DKMS tree for
this status command. If your modules were
not installed with DKMS, they will not show
up here.
```

Then, load the tarball on your target system:

```
# dkms ldtarball
   --archive=megaraid2-2.10.3-manykernels.tgz

Loading tarball for module:
 megaraid2 / version: 2.10.3
Loading /usr/src/megaraid2-2.10.3...
Loading /var/dkms/megaraid2/2.10.3/2.4.21-4.EL...
Loading /var/dkms/megaraid2/2.10.3/2.4.21-4.ELBOOT...
Loading /var/dkms/megaraid2/2.10.3/2.4.21-4.ELhugemem...
Loading /var/dkms/megaraid2/2.10.3/2.4.21-4.ELsmp...
Creating /var/dkms/megaraid2/2.10.3/source symlink...
```

Finally, verify the modules are present, and in the Built state:

```
# dkms status
megaraid2, 2.10.3, 2.4.21-4.EL: built
megaraid2, 2.10.3, 2.4.21-4.ELBOOT: built
megaraid2, 2.10.3, 2.4.21-4.ELhugemem: built
megaraid2, 2.10.3, 2.4.21-4.ELsmp: built
```

DKMS `ldtarball` leaves the modules in the Built state, not the Installed state. For each kernel version you want your modules to be installed into, follow the install steps as above.

6 Driver Developers

As the maintainer of a kernel module, the only thing you need to do to get DKMS interoperability is place a small dkms.conf file in your driver source tarball. Once this has been done, any user of DKMS can simply do:

```
dkms ldtarball --archive /path/to/foo-1.0.tgz
```

That's it. We could discuss at length (which we will not rehash in this paper) the best methods to utilizing DKMS within a dkms-enabled module RPM, but for simple DKMS usability, the buck stops here. With the dkms.conf file in place, you have now positioned your source tarball to be usable by all manner and skill level of Linux users utilizing your driver. Effectively, you have widely increased your testing base without having to wade into package management or pre-compiled binaries. DKMS will handle this all for you. Along the same line, by leveraging DKMS you can now easily allow more widespread testing of your driver. Since driver versions can now be cleanly tracked outside of the kernel tree, you no longer must wait for the next kernel release in order for the community to register the necessary debugging cycles against your code. Instead, DKMS can be counted on to manage various versions of your kernel module such that any catastrophic errors in your code can be easily mitigated by a singular dkms uninstall command.

This leaves the composition of the dkms.conf as the only interesting piece left to discuss for the driver developer audience. With that in mind, we will now explicate over two dkms.conf examples ranging from that which is minimally required (Figure 2) to that which expresses maximal configuration (Figure 3).

6.1 Minimal dkms.conf for 2.4 kernels

Referring to Figure 2, the first thing that is distinguishable is the definition of the version of the package and the make command to be used to compile your module. This is only necessary for 2.4-based kernels, and lets the developer specify their desired make incantation.

Reviewing the rest of the dkms.conf, `PACKAGE_NAME` and `BUILT_MODULE_NAME[0]` appear to be duplicate in nature, but this is only the case for a package which contains only one kernel module within it. Had this example been for something like ALSA, the name of the package would be "alsa," but the `BUILT_MODULE_NAME` array would instead be populated with the names of the kernel modules within the ALSA package.

The final required piece of this minimal example is the `DEST_MODULE_LOCATION` array. This simply tells DKMS where in the /lib/modules tree it should install your module.

```
PACKAGE_NAME="megaraid2"
PACKAGE_VERSION="2.10.3"

MAKE[0]="make -C ${kernel_source_dir}
  SUBDIRS=${dkms_tree}/${PACKAGE_NAME}/${PACKAGE_VERSION}/build modules"

BUILT_MODULE_NAME[0]="megaraid2"
DEST_MODULE_LOCATION[0]="/kernel/drivers/scsi/"
```

Figure 2: A minimal dkms.conf

6.2 Minimal dkms.conf for 2.6 kernels

In the current version of DKMS, for 2.6 kernels the MAKE command listed in the dkms.conf is wholly ignored, and instead DKMS will always use:

```
make -C /lib/modules/$kernel_version/build \
  M=$dkms_tree/$module/$module_version/build
```

This jibes with the new external module build infrastructure supported by Sam Ravnborg's kernel Makefile improvements, as DKMS will always build your module in a build subdirectory it creates for each version you have installed. Similarly, an impending future version of DKMS will also begin to ignore the `PACKAGE_VERSION` as specified in dkms.conf in favor of the new modinfo provided information as implemented by Rusty Russell.

With regard to removing the requirement for `DEST_MODULE_LOCATION` for 2.6 kernels, given that similar information should be located in the install target of the Makefile provided with your package, it is theoretically possible that DKMS could one day glean such information from the Makefile instead. In fact, in a simple scenario as this example, it is further theoretically possible that the name of the package and of the built module could also be determined from the package Makefile. In effect, this would completely remove any need for a dkms.conf whatsoever, thus enabling all simple module tarballs to be automatically DKMS enabled.

Though, as these features have not been explored and as package maintainers would likely want to use some of the other dkms.conf directive features which are about to be elaborated upon, it is likely that requiring a dkms.conf will continue for the foreseeable future.

6.3 Optional dkms.conf directives

In the real-world version of the Dell's DKMS-enabled megaraid2 package, we also specify the optional directives:

```
MODULES_CONF_ALIAS_TYPE[0]=
   "scsi_hostadapter"
MODULES_CONF_OBSOLETES[0]=
   "megaraid,megaraid_2002"
REMAKE_INITRD="yes"
```

These directives tell DKMS to remake the kernel's initial ramdisk after every DKMS install or uninstall of this module. They further specify that before this happens, /etc/modules.conf (or /etc/sysconfig/kernel) should be edited intelligently so that the initrd is properly assembled. In this case, if /etc/modules.conf already contains a reference to either "megaraid" or "megaraid_2002," these will be switched to "megaraid2." If no such references are found, then a new "scsi_hostadapter" entry will be added as the last such scsi_hostadapter number.

On the other hand, if it had also included:

```
MODULES_CONF_OBSOLETES_ONLY="yes"
```

then had no obsolete references been found, a new "scsi_hostadapter" line would not have been added. This would be useful in scenarios where you instead want to rely on something like Red Hat's kudzu program for adding references for your kernel modules.

As well one could hypothetically also specify within the dkms.conf:

```
DEST_MODULE_NAME[0]="megaraid"
```

This would cause the resultant megaraid2 kernel module to be renamed to "megaraid" before being installed. Rather than having to propagate various one-off naming mechanisms which include the version as part of the module name in /lib/modules as has been previous common practice, DKMS could instead be relied upon to manage all module versioning to avoid such clutter. Was megaraid_2002 a version or just a special year in the hearts of the megaraid developers? While you and I might know the answer to this, it certainly confused Dell's customers.

Continuing with hypothetical additions to the dkms.conf in Figure 2, one could also include:

```
BUILD_EXCLUSIVE_KERNEL="^2\.4.*"
BUILD_EXCLUSIVE_ARCH="i.86"
```

In the event that you know the code you produced is not portable, this is how you can tell DKMS to keep people from trying to build it elsewhere. The above restrictions would only allow the kernel module to be built on 2.4 kernels on x86 architectures.

Continuing with optional dkms.conf directives, the ALSA example in Figure 3 is taken directly from a DKMS-enabled package that Dell released to address sound issues on the Precision 360 workstation. It is slightly abridged as the alsa-driver as delivered actually installs 13 separate kernel modules, but for the sake of this example, only 9 are shown.

In this example, we have:

```
AUTOINSTALL="yes"
```

This tells the boot-time service dkms_autoinstaller that this package should be built and installed as you boot into a new kernel that DKMS has not already installed this package upon. By general policy, Dell only allows `AUTOINSTALL` to be set if the kernel modules are not already natively included with the kernel. This is to avoid the scenario where DKMS might automatically install over a newer version of the kernel module as provided by some newer version of the kernel. However, given the 2.6 modinfo changes, DKMS can now be modified to intelligently check the version of a native kernel module before clobbering it with some older version. This will likely result in a future policy change within Dell with regard to this feature.

In this example, we also have:

```
PATCH[0]="adriver.h.patch"
PATCH_MATCH[0]="2.4.[2-9][2-9]"
```

These two directives indicate to DKMS that if the kernel that the kernel module is being built for is >=2.4.22 (but still of the 2.4 family), the included adriver.h.patch should first be

```
PACKAGE_NAME="alsa-driver"
PACKAGE_VERSION="0.9.0rc6"

MAKE="sh configure --with-cards=intel8x0 --with-sequencer=yes \
  --with-kernel=/lib/modules/$kernelver/build \
  --with-moddir=/lib/modules/$kernelver/kernel/sound > /dev/null; make"
AUTOINSTALL="yes"

PATCH[0]="adriver.h.patch"
PATCH_MATCH[0]="2.4.[2-9][2-9]"

POST_INSTALL="alsa-driver-dkms-post.sh"
MODULES_CONF[0]="alias char-major-116 snd"
MODULES_CONF[1]="alias snd-card-0 snd-intel8x0"
MODULES_CONF[2]="alias char-major-14 soundcore"
MODULES_CONF[3]="alias sound-slot-0 snd-card-0"
MODULES_CONF[4]="alias sound-service-0-0 snd-mixer-oss"
MODULES_CONF[5]="alias sound-service-0-1 snd-seq-oss"
MODULES_CONF[6]="alias sound-service-0-3 snd-pcm-oss"
MODULES_CONF[7]="alias sound-service-0-8 snd-seq-oss"
MODULES_CONF[8]="alias sound-service-0-12 snd-pcm-oss"
MODULES_CONF[9]="post-install snd-card-0 /usr/sbin/alsactl restore >/dev/null 2>&1 || :"
MODULES_CONF[10]="pre-remove snd-card-0 /usr/sbin/alsactl store >/dev/null 2>&1 || :"

BUILT_MODULE_NAME[0]="snd-pcm"
BUILT_MODULE_LOCATION[0]="acore"
DEST_MODULE_LOCATION[0]="/kernel/sound/acore"

BUILT_MODULE_NAME[1]="snd-rawmidi"
BUILT_MODULE_LOCATION[1]="acore"
DEST_MODULE_LOCATION[1]="/kernel/sound/acore"

BUILT_MODULE_NAME[2]="snd-timer"
BUILT_MODULE_LOCATION[2]="acore"
DEST_MODULE_LOCATION[2]="/kernel/sound/acore"

BUILT_MODULE_NAME[3]="snd"
BUILT_MODULE_LOCATION[3]="acore"
DEST_MODULE_LOCATION[3]="/kernel/sound/acore"

BUILT_MODULE_NAME[4]="snd-mixer-oss"
BUILT_MODULE_LOCATION[4]="acore/oss"
DEST_MODULE_LOCATION[4]="/kernel/sound/acore/oss"

BUILT_MODULE_NAME[5]="snd-pcm-oss"
BUILT_MODULE_LOCATION[5]="acore/oss"
DEST_MODULE_LOCATION[5]="/kernel/sound/acore/oss"

BUILT_MODULE_NAME[6]="snd-seq-device"
BUILT_MODULE_LOCATION[6]="acore/seq"
DEST_MODULE_LOCATION[6]="/kernel/sound/acore/seq"

BUILT_MODULE_NAME[7]="snd-seq-midi-event"
BUILT_MODULE_LOCATION[7]="acore/seq"
DEST_MODULE_LOCATION[7]="/kernel/sound/acore/seq"

BUILT_MODULE_NAME[8]="snd-seq-midi"
BUILT_MODULE_LOCATION[8]="acore/seq"
DEST_MODULE_LOCATION[8]="/kernel/sound/acore/seq"

BUILT_MODULE_NAME[9]="snd-seq"
BUILT_MODULE_LOCATION[9]="acore/seq"
DEST_MODULE_LOCATION[9]="/kernel/sound/acore/seq"
```

Figure 3: An elaborate dkms.conf

applied to the module source before a module build occurs. In this way, by including various patches needed for various kernel versions, you can distribute one source tarball and ensure it will always properly build regardless of the end user target kernel. If no corresponding `PATCH_MATCH[0]` entry were specified for `PATCH[0]`, then the adriver.h.patch would always get applied before a module build. As DKMS always starts off each module build with pristine module source, you can always ensure the right patches are being applied.

Also seen in this example is:

```
MODULES_CONF[0]=
  "alias char-major-116 snd"
MODULES_CONF[1]=
  "alias snd-card-0 snd-intel8x0"
```

Unlike the previous discussion of /etc/modules.conf changes, any entries placed into the `MODULES_CONF` array are automatically added into /etc/modules.conf during a module install. These are later only removed during the final module uninstall.

Lastly, we have:

```
POST_INSTALL="alsa-driver-dkms-post.sh"
```

In the event that you have other scripts that must be run during various DKMS events, DKMS includes `POST_ADD`, `POST_BUILD`, `POST_INSTALL` and `POST_REMOVE` functionality.

7 Future

As you can tell from the above, DKMS is very much ready for deployment now. However, as with all software projects, there's room for improvement.

7.1 Cross-Architecture Builds

DKMS today has no concept of a platform architecture such as i386, x86_64, ia64, sparc, and the like. It expects that it is building kernel modules with a native compiler, not a cross compiler, and that the target architecture is the native architecture. While this works in practice, it would be convenient if DKMS were able to be used to build kernel modules for non-native architectures.

Today DKMS handles the cross-architecture build process by having separate /var/dkms directory trees for each architecture, and using the `dkmstree` option to specify a using a different tree, and the `config` option to specify to use a different kernel configuration file.

Going forward, we plan to add an −−`arch` option to DKMS, or have it glean it from the kernel config file and act accordingly.

7.2 Additional distribution driver disks

DKMS today supports generating driver disks in the Red Hat format only. We recognize that other distributions accomplish the same goal using other driver disk formats. This should be relatively simple to add once we understand what the additional formats are.

8 Conclusion

DKMS provides a simple and unified mechanism for driver authors, Linux distributions, system vendors, and system administrators to update the device drivers on a target system without updating the whole kernel. It allows driver developers to keep their work aimed at the "top of the tree," and to backport that work to older kernels painlessly. It allows Linux distributions to provide updates to single device drivers asynchronous to the release of a larger scheduled update, and to know what drivers have been updated. It lets system vendors ship newer hardware than was supported in a distribution's "gold" release without invalidating any test or certification work done on the "gold" release. It lets system administrators update individual drivers to match their environment and their needs, regardless of whose kernel they are running. It lets end users track which module versions have been added to their system.

We believe DKMS is a project whose time has come, and encourage everyone to use it.

9 References

DKMS is licensed under the GNU General Public License. It is available at

`http://linux.dell.com/dkms/`,

and has a mailing list `dkms-devel@lists.us.dell.com` to which you may subscribe at `http://lists.us.dell.com/`.

e100 Weight Reduction Program

Writing for Maintainability

Scott Feldman
Intel Corporation
`scott.feldman@intel.com`

Abstract

Corporate-authored device drivers are bloated/buggy with dead code, HW and OS abstraction layers, non-standard user controls, and support for complicated HW features that provide little or no value. e100 in 2.6.4 has been rewritten to address these issues and in the process lost 75% of the lines of code, with no loss of functionality. This paper gives guidelines to other corporate driver authors.

Introduction

This paper gives some basic guidelines to corporate device driver maintainers based on experiences I had while re-writing the e100 network device driver for Intel's PRO/100+ Ethernet controllers. By corporate maintainer, I mean someone employed by a corporation to provide Linux driver support for that corporation's device. Of course, these guidelines may apply to non-corporate individuals as well, but the intended audience is the corporate driver author.

The assumption behind these guidelines is that the device driver is intended for inclusion in the Linux kernel. For a driver to be accepted into the Linux kernel, it must meet both technical and non-technical requirements. This paper focuses on the non-technical requirements, specifically maintainability.

Guideline #1: Maintainability over Everything Else

Corporate marketing requirements documents specify priority order to features and performance and schedule (time-to-market), but rarely specify maintainability. However, maintainability is the *most* important requirement for Linux kernel drivers.

Why?

- You will not be the long-term driver maintainer.
- Your company will not be the long-term driver maintainer.
- Your driver will out-live your interest in it.

Driver code should be written so a like-skilled kernel maintainer can fix a problem in a reasonable amount of time without you or your resources. Here are a few items to keep in mind to improve maintainability.

- Use kernel coding style over corporate coding style
- Document how the driver/device works, at a high level, in a "Theory of Operation" comment section

old driver v2	new driver v3
VLANs tagging/ stripping	use SW VLAN support in kernel
Tx/Rx checksum offloading	use SW checksum support in kernel
interrupt moderation	use NAPI support in kernel

Table 1: Feature migration in e100

- Document hardware workarounds

Guideline #2: Don't Add Features for Feature's Sake

Consider the code complexity to support the feature versus the user's benefit. Is the device still usable without the feature? Is the device performing reasonably for the 80% use-case without the feature? Is the hardware offload feature working against ever increasing CPU/memory/IO speeds? Is there a software equivalent to the feature already provided in the OS?

If the answer is yes to any of these questions, it is better to not implement the feature, keeping the complexity in the driver low and maintainability high.

Table 1 shows features removed from the driver during the re-write of e100 because the OS already provides software equivalents.

Guideline #3: Limit User-Controls—Use What's Built into the OS

Most users will use the default settings, so before adding a user-control, consider:

1. If the driver model for your device class already provides a mechanism for the user-control, enable that support in the driver rather than adding a custom user-control.
2. If the driver model doesn't provide a user-control, but the user-control is potentially useful to other drivers, extend the driver model to include user-control.
3. If the user-control is to enable/disable a workaround, enable the workaround without the use of a user-control. (Solve the problem without requiring a decision from the user).
4. If the user-control is to tune performance, tune the driver for the 80% use-case and remove the user-control.

old driver v2	new driver v3
BundleMax	not needed – NAPI
BundleSmallFr	not needed – NAPI
IntDelay	not needed – NAPI
ucode	not needed – NAPI
RxDescriptors	ethtool -G
TxDescriptors	ethtool -G
XsumRX	not needed – checksum in OS
IFS	always enabled
e100_speed_duplex	ethtool -s

Table 2: User-control migration in e100

Table 2 shows user-controls (implemented as module parameters) removed from the driver during the re-write of e100 because the OS already provides built-in user-controls, or the user-control was no longer needed.

Guideline #4: Don't Write Code that's Already in the Kernel

Look for library code that's already used by other drivers and adapt that to your driver. Common hardware is often used between vendors' devices, so shared code will work for all (and be debugged by all).

For example, e100 has a highly MDI-compliant PHY interface, so use `mii.c` for standard PHY access and remove custom code from the driver.

For another example, e100 v2 used `/proc/net/IntelPROAdapter` to report driver information. This functionality was replaced with `ethtool`, `sysfs`, `lspci`, etc.

Look for opportunities to move code out of the driver into generic code.

Guideline #5: Don't Use OS-abstraction Layers

A common corporate design goal is to reuse driver code as much as possible between OSes. This allows a driver to be brought up on one OS and "ported" to another OS with little work. After all, the hardware interface to the device didn't change from one OS to the next, so all that is required is an OS-abstraction layer that wraps the OS's native driver model with a generic driver model. The driver is then written to the generic driver model and it's just a matter of porting the OS-abstraction layer to each target OS.

There are problems when doing this with Linux:

1. The OS-abstraction wrapper code means nothing to an outside Linux maintainer and just obfuscates the real meaning behind the code. This makes your code harder to follow and therefore harder to maintain.

2. The generic driver model may not map 1:1 with the native driver model leaving gaps in compatibility that you'll need to fix up with OS-specific code.

3. Limits your ability to back-port contributions given under GPL to non-GPL OSes.

Guideline #6: Use kcompat Techniques to Move Legacy Kernel Support out of the Driver (and Kernel)

Users may not be able to move to the latest `kernel.org` kernel, so there is a need to provide updated device drivers that can be installed against legacy kernels. The need is driven by 1) bug fixes, 2) new hardware support that wasn't included in the driver when the driver was included in the legacy kernel.

The best strategy is to:

1. Maintain your driver code to work against the latest `kernel.org` development kernel API. This will make it easier to keep the driver in the `kernel.org` kernel synchronized with your code base as changes (patches) are almost always in reference to the latest `kernel.org` kernel.

2. Provide a kernel-compat-layer (kcompat) to translate the latest API to the supported legacy kernel API. The driver code is void of any `ifdef` code for legacy kernel support. All of the `ifdef` logic moves to the kcompat layer. The kcompat layer is not included in the latest `kernel.org` kernel (by definition).

Here is an example with e100.

In driver code, use the latest API:

```
s = pci_name(pdev);
...
free_netdev(netdev);
```

In kcompat code, translate to legacy kernel API:

```
#if ( LINUX_VERSION_CODE < \
      KERNEL_VERSION(2,4,22) )
#define pci_name(x)  ((x)->slot_name)
#endif

#ifndef HAVE_FREE_NETDEV
#define free_netdev(x) kfree(x)
#endif
```

Guideline #7: Plan to Re-write the Driver at Least Once

You will not get it right the first time. Plan on rewriting the driver from scratch at least once. This will cleanse the code, removing dead code and organizing/consolidating functionality.

For example, the last e100 re-write reduced the driver size by 75% without loss of functionality.

Conclusion

Following these guidelines will result in more maintainable device drivers with better acceptance into the Linux kernel tree. The basic idea is to remove as much as possible from the driver without loss of functionality.

References

- The latest e100 driver code is available at `linux/driver/net/e100.c` (2.6.4 kernel or higher).
- An example of kcompat is here: `http://sf.net/projects/gkernel`

NFSv4 and `rpcsec_gss` for linux

J. Bruce Fields
University of Michigan
`bfields@umich.edu`

Abstract

The 2.6 Linux kernels now include support for version 4 of NFS. In addition to built-in locking and ACL support, and features designed to improve performance over the Internet, NFSv4 also mandates the implementation of strong cryptographic security. This security is provided by rpcsec_gss, a standard, widely implemented protocol that operates at the rpc level, and hence can also provide security for NFS versions 2 and 3.

1 The rpcsec_gss protocol

The rpc protocol, which all version of NFS and related protocols are built upon, includes generic support for authentication mechanisms: each rpc call has two fields, the credential and the verifier, each consisting of a 32-bit integer, designating a "security flavor," followed by 400 bytes of opaque data whose structure depends on the specified flavor. Similarly, each reply includes a single "verifier."

Until recently, the only widely implemented security flavor has been the auth_unix flavor, which uses the credential to pass uid's and gid's and simply asks the server to trust them. This may be satisfactory given physical security over the clients and the network, but for many situations (including use over the Internet), it is inadequate.

Thus rfc 2203 defines the rpcsec_gss protocol, which uses rpc's opaque security fields to carry cryptographically secure tokens. The cryptographic services are provided by the GSS-API ("Generic Security Service Application Program Interface," defined by rfc 2743), allowing the use of a wide variety of security mechanisms, including, for example, Kerberos.

Three levels of security are provided by rpcsec_gss:

1. Authentication only: The rpc header of each request and response is signed.
2. Integrity: The header and body of each request and response is signed.
3. Privacy: The header of each request is signed, and the body is encrypted.

The combination of a security level with a GSS-API mechanism can be designated by a 32-bit "pseudoflavor." The mount protocol used with NFS versions 2 and 3 uses a list of pseudoflavors to communicate the security capabilities of a server. NFSv4 does not use pseudoflavors on the wire, but they are still useful in internal interfaces.

Security protocols generally require some initial negotiation, to determine the capabilities of the systems involved and to choose session keys. The rpcsec_gss protocol uses calls with procedure number 0 for this purpose. Normally such a call is a simple "ping" with no side-effects, useful for measuring round-trip latency or testing whether a certain service is running. However a call with procedure number 0, if made with authentication flavor rpcsec_gss, may use certain fields in the credential to indicate that it is part of a context-initiation exchange.

2 Linux implementation of rpcsec_gss

The Linux implementation of rpcsec_gss consists of several pieces:

1. Mechanism-specific code, currently for two mechanisms: krb5 and spkm3.
2. A stripped-down in-kernel version of the GSS-API interface, with an interface that allows mechanism-specific code to register support for various pseudoflavors.
3. Client and server code which uses the GSS-API interface to encode and decode rpc calls and replies.
4. A userland daemon, gssd, which performs context initiation.

2.1 Mechanism-specific code

The NFSv4 RFC mandates the implementation (though not the use) of three GSS-API mechanisms: krb5, spkm3, and lipkey.

Our krb5 implementation supports three pseudoflavors: krb5, krb5i, and krb5p, providing authentication only, integrity, and privacy, respectively. The code is derived from MIT's Kerberos implementation, somewhat simplified, and not currently supporting the variety of encryption algorithms that MIT's does. The krb5 mechanism is also supported by NFS implementations from Sun, Network Appliance, and others, which it interoperates with.

The Low Infrastructure Public Key Mechanism ("lipkey," specified by rfc 2847), is a public key mechanism built on top of the Simple Public Key Mechanism (spkm), which provides functionality similar to that of TLS, allowing a secure channel to be established using a server-side certificate and a client-side password.

We have a preliminary implementation of spkm3 (without privacy), but none yet of lipkey. Other NFS implementors have not yet implemented either of these mechanisms, but there appears to be sufficient interest from the grid community for us to continue implementation even if it is Linux-only for now.

2.2 GSS-API

The GSS-API interface as specified is very complex. Fortunately, rpcsec_gss only requires a subset of the GSS-API, and even less is required for per-packet processing.

Our implementation is derived by the implementation in MIT Kerberos, and initially stayed fairly close the the GSS-API specification; but over time we have pared it down to something quite a bit simpler.

The kernel gss interface also provides APIs by which code implementing particular mechanisms can register itself to the gss-api code and hence can be safely provided by modules loaded at runtime.

2.3 RPC code

The RPC code has been enhanced by the addition of a new rpcsec_gss mechanism which authenticates calls and replies and which wraps and unwraps rpc bodies in the case of integrity and privacy.

This is relatively straightforward, though somewhat complicated by the need to handle discontiguous buffers containing page data.

Caches for session state are also required on both client and server; on the client a preexisting rpc credentials cache is used, and on the server we use the same caching infrastructure used for caching of client and export information.

2.4 Userland daemon

We had no desire to put a complete implementation of Kerberos version 5 or the other mechanisms into the kernel. Fortunately, the work performed by the various GSS-API mechanisms can be divided neatly into context initiation and per-packet processing. The former is complex and is performed only once per session, while the latter is simple by comparison and needs to be performed on every packet. Therefore it makes sense to put the packet processing in the kernel, and have the context initiation performed in userspace.

Since it is the kernel that knows when context initiation is necessary, we require a mechanism allowing the kernel to pass the necessary parameters to a userspace daemon whenever context initiation is needed, and allowing the daemon to respond with the completed security context.

This problem was solved in different ways on the client and server, but both use special files (the former in a dedicated filesystem, rpc_pipefs, and the latter in the proc filesystem), which our userspace daemon, gssd, can poll for requests and then write responses back to.

In the case of Kerberos, the sequence of events will be something like this:

1. The user gets Kerberos credentials using kinit, which are cached on a local filesystem.

2. The user attempts to perform an operation on an NFS filesystem mounted with krb5 security.

3. The kernel rpc client looks for the a security context for the user in its cache; not finding any, it does an upcall to gssd to request one.

4. Gssd, on receiving the upcall, reads the user's Kerberos credentials from the local filesystem and uses them to construct a null rpc request which it sends to the server.

5. The server kernel makes an upcall which passes the null request to its gssd.

6. At this point, the server gssd has all it needs to construct a security context for this session, consisting mainly of a session key. It passes this context down to the kernel rpc server, which stores it in its context cache.

7. The server's gssd then constructs the null rpc reply, which it gives to the kernel to return to the client gssd.

8. The client gssd uses this reply to construct its own security context, and passes this context to the kernel rpc client.

9. The kernel rpc client then uses this context to send the first real rpc request to the server.

10. The server uses the new context in its cache to verify the rpc request, and to compose its reply.

3 The NFSv4 protocol

While rpcsec_gss works equally well on all existing versions of NFS, much of the work on rpcsec_gss has been motivated by NFS version 4, which is the first version of NFS to make rpcsec_gss mandatory to implement.

This new version of NFS is specified by rfc 3530, which says:

"Unlike earlier versions, the NFS version 4 protocol supports traditional file access while integrating support for file locking and the mount protocol. In addition, support for strong security (and its negotiation), compound operations, client caching, and internationalization have been added. Of course, attention has been applied to making NFS version 4 operate well in an Internet environment."

Descriptions of some of these features follow, with some notes about their implementation in Linux.

3.1 Compound operations

Each rpc request includes a procedure number, which describes the operation to be performed. The format of the body of the rpc request (the arguments) and of the reply depend on the program number. Procedure 0 is reserved as a no-op (except when it is used for rpcsec_gss context initiation, as described above).

The NFSv4 protocol only supports one non-zero procedure, procedure 1, the compound procedure.

The body of a compound is a list of operations, each with its own arguments. For example, a compound request performing a lookup might consist of 3 operations: a PUTFH, with a filehandle, which sets the "current filehandle" to the provided filehandle; a LOOKUP, with a name, which looks up the name in the directory given by the current filehandle and then modifies the current filehandle to be the filehandle of the result; a GETFH, with no arguments, which returns the new value of the current filehandle; and a GETATTR, with a bitmask specifying a set of attributes to return for the looked-up file.

The server processes these operations in order, but with no guarantee of atomicity. On encountering any error, it stops and returns the results of the operations up to and including the operation that failed.

In theory complex operations could therefore be done by long compounds which perform complex series of operations.

In practice, the compounds sent by the Linux client correspond very closely to NFSv2/v3 procedures—the VFS and the POSIX filesystem API make it difficult to do otherwise—and our server, like most NFSv4 servers we know of, rejects overly long or complex compounds.

3.2 Well-known port for NFS

RPC allows services to be run on different ports, using the "portmap" service to map program numbers to ports. While flexible, this system complicates firewall management; so NFSv4 recommends the use of port 2049.

In addition, the use of sideband protocols for mounting, locking, etc. also complicates firewall management, as multiple connections to multiple ports are required for a single NFS mount. NFSv4 eliminates these extra protocols, allowing all traffic to pass over a single connection using one protocol.

3.3 No more mount protocol

Earlier versions of NFS use a separate protocol for mount. The mount protocol exists primarily to map path names, presented to the server as strings, to filehandles, which may then be used in the NFS protocol.

NFSv4 instead uses a single operation, PUTROOTFH, that returns a filehandle; clients can then use ordinary lookups to traverse to the filesystem they wish to mount. This changes the behavior of NFS in a few subtle ways: for example, the special status of mounts in the old protocol meant that mounting `/usr` and then looking up `local` might get you a different object than would mounting `/usr/local`; under NFSv4 this can no longer happen.

A server that exports multiple filesystems must knit them together using a single "pseudofilesystem" which links them to a common root.

On Linux's nfsd the pseudofilesystem is a real filesystem, marked by the export option "fsid=0". An adminstrator that is content to export a single filesystem can export it with "fsid=0", and clients will find it just by mounting the path "/".

The expected use for "fsid=0", however, is to designate a filesystem that is used just a collection of empty directories used as mountpoints for exported filesystems, which are mounted using `mount ---bind`; thus an administrator could export `/bin` and `/local/src` by:

```
mkdir -p /exports/home
mkdir -p /exports/bin/
mount --bind /home /exports/home
mount --bind /bin/ /exports/bin
```

and then using an exports file something like:

```
/exports *.foo.com(fsid=0,crossmnt)
/exports/home *.foo.com
/exports/bin *.foo.com
```

Clients in `foo.com` can then mount `server.foo.com:/bin` or `server.foo.com:/home`. However the relationship between the original mountpoint on the server and the mountpoint under `/exports` (which determines the path seen by the client) is arbitrary, so the administrator could just as well export `/home` as `/some/other/path` if desired.

This gives maximum flexibility at the expense of some confusion for adminstrators used to earlier NFS versions.

3.4 No more lock protocol

Locking has also been absorbed into the NFSv4 protocol. In addition to advantages enumerated above, this allows servers to support mandatory locking if desired. Previously this was impossible because it was impossible to tell whether a given read or write should be ordered before or after a lock request. NFSv4 enforces such sequencing by providing a stateid field on each read or write which identifies the locking state that the operation was performed under; thus for example a write that occurred while a lock was held, but that appeared on the server to have occurred after an unlock, can be identified as belonging to a previous locking context, and can therefore be correctly rejected.

The additional state required to manage locking is the source of much of the additional complexity in NFSv4.

3.5 String representations of user and group names

Previous versions of NFS use integers to represent users and groups; while simple to handle, they can make NFS installations difficult to manage, particularly across adminstrative domains. Version 4, therefore, uses string names of the form `user@domain`.

This poses some challenges for the kernel implementation. In particular, while the protocol may use string names, the kernel still needs to deal with uid's, so it must map between NFSv4 string names and integers.

As with rpcsec_gss context initation, we solve this problem by making upcalls to a userspace daemon; with the mapping in userspace, it is easy to use mechanisms such as NIS or LDAP to do the actual mapping without introducing large amounts of code into the kernel. So as not to degrade performance by requiring a context switch every time we process a packet carrying a name, we cache the results of this mapping in the kernel.

3.6 Delegations

NFSv4, like previous versions of NFS, does not attempt to provide full cache consistency. Instead, all that is guaranteed is that if an open follows a close of the same file, then data read after the open will reflect any modifications performed before the close. This makes both open and close potentially high latency operations, since they must wait for at least one round trip before returning–in the close case, to flush out any pending writes, and in the open case, to check the attributes of the file in question to determine whether the local cache should be invalidated.

Locks provide similar semantics—writes are flushed on unlock, and cache consistency is verified on lock—and hence lock operations are also prone to high latencies.

To mitigate these concerns, and to encourage the use of NFS's locking features, delegations have been added to NFSv4. Delegations are granted or denied by the server in response to open calls, and give the client the right to perform later locks and opens locally, without the need to contact the server. A set of callbacks is provided so that the server can notify the client when another client requests an open that would confict with the open originally obtained by the client.

Thus locks and opens may be performed quickly by the client in the common case when files are not being shared, but callbacks ensure that correct close-to-open (and unlock-to-lock) semantics may be enforced when there is contention.

To allow other clients to proceed when a client holding a delegation reboots, clients are required to periodically send a "renew" operation to the server, indicating that it is still alive; a client that fails to send a renew operation within a given lease time (established when the client first contacts the server) may have all of its delegations and other locking state revoked.

Most implementations of NFSv4 delegations, including Linux's, are still young, and we haven't yet gathered good data on the performance impact.

Nevertheless, further extensions, including delegations over directories, are under consideration for future versions of the protocol.

3.7 ACLs

ACL support is integrated into the protocol, with ACLs that are more similar to those found in NT than to the POSIX ACLs supported by Linux.

Thus while it is possible to translate an arbitrary Linux ACL to an NFS4 ACL with nearly identical meaning, most NFS ACLs have no reasonable representation as Linux ACLs.

Marius Eriksen has written a draft describing the POSIX to NFS4 ACL translation. Currently the Linux implementation uses this mapping, and rejects any NFS4 ACL that isn't exactly in the image of this mapping. This ensures userland support from all tools that currently support POSIX ACLs, and simplifies ACL management when an exported filesystem is also used by local users, since both nfsd and the local users can use the backend filesystem's POSIX ACL implementation. However it makes it difficult to interoperate with NFSv4 implementations that support the full ACL protocol. For that reason we will eventually also want to add support for NFSv4 ACLs.

4 Acknowledgements and Further Information

This work has been sponsored by Sun Microsystems, Network Appliance, and the Accelerated Strategic Computing Initiative (ASCI). For further information, see `www.citi.umich.edu/projects/nfsv4/`.

Comparing and Evaluating epoll, select, and poll Event Mechanisms

Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag
University of Waterloo
`{lgammo,brecht,ashukla,db2pariag}@cs.uwaterloo.ca`

Abstract

This paper uses a high-performance, event-driven, HTTP server (the μserver) to compare the performance of the select, poll, and epoll event mechanisms. We subject the μserver to a variety of workloads that allow us to expose the relative strengths and weaknesses of each event mechanism.

Interestingly, initial results show that the select and poll event mechanisms perform comparably to the epoll event mechanism in the absence of idle connections. Profiling data shows a significant amount of time spent in executing a large number of `epoll_ctl` system calls. As a result, we examine a variety of techniques for reducing `epoll_ctl` overhead including edge-triggered notification, and introducing a new system call (`epoll_ctlv`) that aggregates several `epoll_ctl` calls into a single call. Our experiments indicate that although these techniques are successful at reducing `epoll_ctl` overhead, they only improve performance slightly.

1 Introduction

The Internet is expanding in size, number of users, and in volume of content, thus it is imperative to be able to support these changes with faster and more efficient HTTP servers.

A common problem in HTTP server scalability is how to ensure that the server handles a large number of connections simultaneously without degrading the performance. An event-driven approach is often implemented in high-performance network servers [14] to multiplex a large number of concurrent connections over a few server processes. In event-driven servers it is important that the server focuses on connections that can be serviced without blocking its main process. An event dispatch mechanism such as `select` is used to determine the connections on which forward progress can be made without invoking a blocking system call. Many different event dispatch mechanisms have been used and studied in the context of network applications. These mechanisms range from `select`, `poll`, `/dev/poll`, RT signals, and epoll [2, 3, 15, 6, 18, 10, 12, 4].

The epoll event mechanism [18, 10, 12] is designed to scale to larger numbers of connections than `select` and `poll`. One of the problems with `select` and `poll` is that in a single call they must both inform the kernel of all of the events of interest and obtain new events. This can result in large overheads, particularly in environments with large numbers of connections and relatively few new events occurring. In a fashion similar to that described by Banga et al. [3] epoll separates mechanisms for obtaining events (`epoll_wait`) from those used to declare and control interest in events (`epoll_ctl`).

Further reductions in the number of generated events can be obtained by using edge-triggered epoll semantics. In this mode events are only provided when there is a change in the state of the socket descriptor of interest. For compatibility with the semantics offered by `select` and `poll`, epoll also provides level-triggered event mechanisms.

To compare the performance of epoll with `select` and `poll`, we use the μserver [4, 7] web server. The μserver facilitates comparative analysis of different event dispatch mechanisms within the same code base through command-line parameters. Recently, a highly tuned version of the single process event driven μserver using `select` has shown promising results that rival the performance of the in-kernel TUX web server [4].

Interestingly, in this paper, we found that for some of the workloads considered `select` and `poll` perform as well as or slightly better than epoll. One such result is shown in Figure 1. This motivated further investigation with the goal of obtaining a better understanding of epoll's behaviour. In this paper, we describe our experience in trying to determine how to best use epoll, and examine techniques designed to improve its performance.

The rest of the paper is organized as follows: In Section 2 we summarize some existing work that led to the development of epoll as a scalable replacement for `select`. In Section 3 we describe the techniques we have tried to improve epoll's performance. In Section 4 we describe our experimental methodology, including the workloads used in the evaluation. In Section 5 we describe and analyze the results of our experiments. In Section 6 we summarize our findings and outline some ideas for future work.

2 Background and Related Work

Event-notification mechanisms have a long history in operating systems research and development, and have been a central issue in many performance studies. These studies have sought to improve mechanisms and interfaces for obtaining information about the state of socket and file descriptors from the operating system [2, 1, 3, 13, 15, 6, 18, 10, 12]. Some of these studies have developed improvements to `select`, `poll` and `sigwaitinfo` by reducing the amount of data copied between the application and kernel. Other studies have reduced the number of events delivered by the kernel, for example, the signal-per-fd scheme proposed by Chandra et al. [6]. Much of the aforementioned work is tracked and discussed on the web site, "The C10K Problem" [8].

Early work by Banga and Mogul [2] found that despite performing well under laboratory conditions, popular event-driven servers performed poorly under real-world conditions. They demonstrated that the discrepancy is due the inability of the `select` system call to scale to the large number of simultaneous connections that are found in WAN environments.

Subsequent work by Banga et al. [3] sought to improve on `select`'s performance by (among other things) separating the declaration of interest in events from the retrieval of events on that interest set. Event mechanisms like select and poll have traditionally combined these tasks into a single system call. However, this amalgamation requires the server to re-declare its interest set every time it wishes to retrieve events, since the kernel does not remember the interest sets from previous calls. This results in unnecessary data copying between the application and the kernel.

The `/dev/poll` mechanism was adapted from Sun Solaris to Linux by Provos et al. [15], and improved on poll's performance by introducing a new interface that separated the declaration of interest in events from retrieval. Their /dev/poll mechanism further reduced data copying by using a shared memory region to return events to the application.

The kqueue event mechanism [9] addressed many of the deficiencies of select and poll for FreeBSD systems. In addition to separating the declaration of interest from retrieval, kqueue allows an application to retrieve events from a variety of sources including file/socket descriptors, signals, AIO completions, file system changes, and changes in process state.

The epoll event mechanism [18, 10, 12] investigated in this paper also separates the declaration of interest in events from their retrieval. The epoll_create system call instructs the kernel to create an event data structure that can be used to track events on a number of descriptors. Thereafter, the epoll_ctl call is used to modify interest sets, while the epoll_wait call is used to retrieve events.

Another drawback of select and poll is that they perform work that depends on the size of the interest set, rather than the number of events returned. This leads to poor performance when the interest set is much larger than the active set. The epoll mechanisms avoid this pitfall and provide performance that is largely independent of the size of the interest set.

3 Improving epoll Performance

Figure 1 in Section 5 shows the throughput obtained when using the μserver with the select, poll, and level-triggered epoll (epoll-LT) mechanisms. In this graph the x-axis shows increasing request rates and the y-axis shows the reply rate as measured by the clients that are inducing the load. This graph shows results for the one-byte workload. These results demonstrate that the μserver with level-triggered epoll does not perform as well as select under conditions that stress the event mechanisms. This led us to more closely examine these results. Using gprof, we observed that epoll_ctl was responsible for a large percentage of the run-time. As can been seen in Table 1 in Section 5 over 16% of the time is spent in epoll_ctl. The gprof output also indicates (not shown in the table) that epoll_ctl was being called a large number of times because it is called for every state change for each socket descriptor. We examine several approaches designed to reduce the number of epoll_ctl calls. These are outlined in the following paragraphs.

The first method uses epoll in an edge-triggered fashion, which requires the μserver to keep track of the current state of the socket descriptor. This is required because with the edge-triggered semantics, events are only received for transitions on the socket descriptor state. For example, once the server reads data from a socket, it needs to keep track of whether or not that socket is still readable, or if it will get another event from epoll_wait indicating that the socket is readable. Similar state information is maintained by the server regarding whether or not the socket can be written. This method is referred to in our graphs and the rest of the paper epoll-ET.

The second method, which we refer to as epoll2, simply calls epoll_ctl twice per socket descriptor. The first to register with the kernel that the server is interested in read and write events on the socket. The second call occurs when the socket is closed. It is used to tell epoll that we are no longer interested in events on that socket. All events are handled in a level-triggered fashion. Although this approach will reduce the number of epoll_ctl calls, it does have potential disadvantages.

One disadvantage of the epoll2 method is that because many of the sockets will continue to be readable or writable `epoll_wait` will return sooner, possibly with events that are currently not of interest to the server. For example, if the server is waiting for a read event on a socket it will not be interested in the fact that the socket is writable until later. Another disadvantage is that these calls return sooner, with fewer events being returned per call, resulting in a larger number of calls. Lastly, because many of the events will not be of interest to the server, the server is required to spend a bit of time to determine if it is or is not interested in each event and in discarding events that are not of interest.

The third method uses a new system call named `epoll_ctlv`. This system call is designed to reduce the overhead of multiple `epoll_ctl` system calls by aggregating several calls to `epoll_ctl` into one call to `epoll_ctlv`. This is achieved by passing an array of epoll events structures to `epoll_ctlv`, which then calls `epoll_ctl` for each element of the array. Events are generated in level-triggered fashion. This method is referred to in the figures and the remainder of the paper as epoll-ctlv.

We use `epoll_ctlv` to add socket descriptors to the interest set, and for modifying the interest sets for existing socket descriptors. However, removal of socket descriptors from the interest set is done by explicitly calling `epoll_ctl` just before the descriptor is closed. We do not aggregate deletion operations because by the time `epoll_ctlv` is invoked, the μserver has closed the descriptor and the `epoll_ctl` invoked on that descriptor will fail.

The μserver does not attempt to batch the closing of descriptors because it can run out of available file descriptors. Hence, the epoll-ctlv method uses both the `epoll_ctlv` and the `epoll_ctl` system calls. Alternatively, we could rely on the `close` system call to remove the socket descriptor from the interest set (and we did try this). However, this increases the time spent by the μserver in `close`, and does not alter performance. We verified this empirically and decided to explicitly call `epoll_ctl` to perform the deletion of descriptors from the epoll interest set.

4 Experimental Environment

The experimental environment consists of a single server and eight clients. The server contains dual 2.4 GHz Xeon processors, 1 GB of RAM, a 10,000 rpm SCSI disk, and two one Gigabit Ethernet cards. The clients are identical to the server with the exception of their disks which are EIDE. The server and clients are connected with a 24-port Gigabit switch. To avoid network bottlenecks, the first four clients communicate with the server's first Ethernet card, while the remaining four use a different IP address linked to the second Ethernet card. The server machine runs a slightly modified version of the 2.6.5 Linux kernel in uniprocessor mode.

4.1 Workloads

This section describes the workloads that we used to evaluate performance of the μserver with the different event notification mechanisms. In all experiments, we generate HTTP loads using *httperf* [11], an open-loop workload generator that uses connection timeouts to generate loads that can exceed the capacity of the server.

Our first workload is based on the widely used SPECweb99 benchmarking suite [17]. We use httperf in conjunction with a SPECweb99 file set and synthetic HTTP traces. Our traces have been carefully generated to recreate the file classes, access patterns, and number of requests issued per (HTTP 1.1) connection that are used in the static portion of SPECweb99. The file set and server caches are sized so that the entire file set fits in the server's cache. This ensures that differences in cache hit rates do not affect performance.

Our second workload is called the one-byte workload. In this workload, the clients repeatedly request the same one byte file from the server's cache. We believe that this workload stresses the event dispatch mechanism by minimizing the amount of work that needs to be done by the server in completing a particular request. By reducing the effect of system calls such as `read` and `write`, this workload isolates the differences due to the event dispatch mechanisms.

To study the scalability of the event dispatch mechanisms as the number of socket descriptors (connections) is increased, we use *idleconn*, a program that comes as part of the httperf suite. This program maintains a steady number of idle connections to the server (in addition to the active connections maintained by httperf). If any of these connections are closed idleconn immediately re-establishes them. We first examine the behaviour of the event dispatch mechanisms without any idle connections to study scenarios where all of the connections present in a server are active. We then pre-load the server with a number of idle connections and then run experiments. The idle connections are used to increase the number of simultaneous connections in order to simulate a WAN environment. In this paper we present experiments using 10,000 idle connections, our findings with other numbers of idle connections were similar and they are not presented here.

4.2 Server Configuration

For all of our experiments, the μserver is run with the same set of configuration parameters except for the event dispatch mechanism. The μserver is configured to use `sendfile` to take advantage of zero-copy socket I/O while writing replies. We use TCP_CORK in conjunction with `sendfile`. The same server options are used for all experiments even though the use of TCP_CORK and `sendfile` may not provide benefits for the one-byte workload when compared with simply using `writev`.

4.3 Experimental Methodology

We measure the throughput of the μserver using different event dispatch mechanisms. In our graphs, each data point is the result of a two minute experiment. Trial and error revealed that two minutes is sufficient for the server to achieve a stable state of operation. A two minute delay is used between consecutive experiments, which allows the TIME_WAIT state on all sockets to be cleared before the subsequent run. All non-essential services are terminated prior to running any experiment.

5 Experimental Results

In this section we first compare the throughput achieved when using level-triggered epoll with that observed when using `select` and `poll` under both the one-byte and SPECweb99-like workloads with no idle connections. We then examine the effectiveness of the different methods described for reducing the number of `epoll_ctl` calls under these same workloads. This is followed by a comparison of the performance of the event dispatch mechanisms when the server is pre-loaded with 10,000 idle connections. Finally, we describe the results of experiments in which we tune the accept strategy used in conjunction with epoll-LT and epoll-ctlv to further improve their performance.

We initially ran the one byte and the SPECweb99-like workloads to compare the performance of the select, poll and level-triggered epoll mechanisms.

As shown in Figure 1 and Figure 2, for both of these workloads select and poll perform as well as epoll-LT. It is important to note that because there are no idle connections for these experiments the number of socket descriptors tracked by each mechanism is not very high. As expected, the gap between epoll-LT and select is more pronounced for the one byte workload because it places more stress on the event dispatch mechanism.

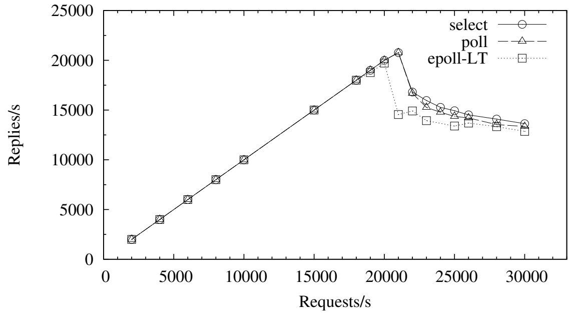


Figure 1: *µserver performance on one byte workload using select, poll, and epoll-LT*

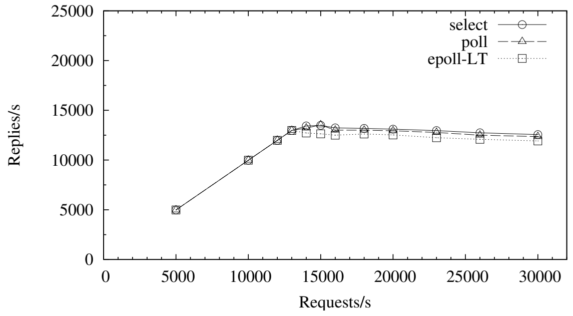


Figure 2: *µserver performance on SPECweb99-like workload using select, poll, and epoll-LT*

We tried to improve the performance of the server by exploring different techniques for using epoll as described in Section 3. The effect of these techniques on the one-byte workload is shown in Figure 3. The graphs in this figure show that for this workload the techniques used to reduce the number of `epoll_ctl` calls do not provide significant benefits when compared with their level-triggered counterpart (epoll-LT). Additionally, the performance of select and poll is equal to or slightly better than each of the epoll techniques. Note that we omit the line for poll from Figures 3 and 4 because it is nearly identical to the select line.

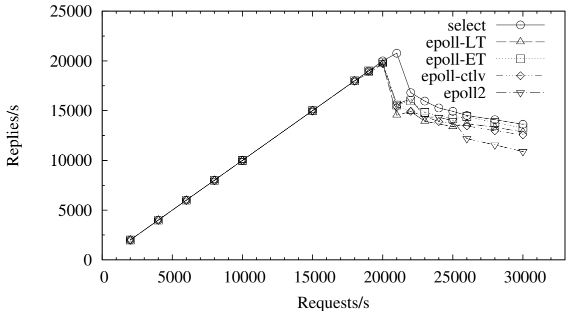


Figure 3: *µserver performance on one byte workload with no idle connections*

We further analyze the results from Figure 3 by profiling the µserver using gprof at the request rate of 22,000 requests per second. Table 1 shows the percentage of time spent in system calls (rows) under the various event dispatch methods (columns). The output for system calls and µserver functions which do not contribute significantly to the total run-time is left out of the table for clarity.

If we compare the select and poll columns we see that they have a similar breakdown including spending about 13% of their time indicating to the kernel events of interest and obtaining events. In contrast the epoll-LT, epoll-ctlv, and epoll2 approaches spend about 21 – 23% of their time on their equivalent functions (`epoll_ctl`, `epoll_ctlv` and `epoll_wait`). Despite these extra overheads the throughputs obtained using the epoll techniques compare favourably with those obtained

	select	**epoll-LT**	**epoll-ctlv**	**epoll2**	**epoll-ET**	**poll**
read	21.51	20.95	21.41	20.08	22.19	20.97
close	14.90	14.05	14.90	13.02	14.14	14.79
select	13.33	-	-	-	-	-
poll	-	-	-	-	-	13.32
epoll_ctl	-	16.34	5.98	10.27	11.06	-
epoll_wait	-	7.15	6.01	12.56	6.52	-
epoll_ctlv	-	-	9.28	-	-	-
setsockopt	11.17	9.13	9.13	7.57	9.08	10.68
accept	10.08	9.51	9.76	9.05	9.30	10.20
write	5.98	5.06	5.10	4.13	5.31	5.70
fcntl	3.66	3.34	3.37	3.14	3.34	3.61
sendfile	3.43	2.70	2.71	3.00	3.91	3.43

Table 1: gprof profile data for the μserver under the one-byte workload at 22,000 requests/sec

using `select` and `poll`. We note that when using `select` and `poll` the application requires extra manipulation, copying, and event scanning code that is not required in the epoll case (and does not appear in the gprof data).

The results in Table 1 also show that the overhead due to `epoll_ctl` calls is reduced in epoll-ctlv, epoll2 and epoll-ET, when compared with epoll-LT. However, in each case these improvements are offset by increased costs in other portions of the code. The epoll2 technique spends twice as much time in `epoll_wait` when compared with epoll-LT. With epoll2 the number of calls to `epoll_wait` is significantly higher, the average number of descriptors returned is lower, and only a very small proportion of the calls (less than 1%) return events that need to be acted upon by the server. On the other hand, when compared with epoll-LT the epoll2 technique spends about 6% less time on `epoll_ctl` calls so the total amount of time spent dealing with events is comparable with that of epoll-LT. Despite the significant `epoll_wait` overheads epoll2 performance compares favourably with the other methods on this workload.

Using the epoll-ctlv technique, gprof indicates that `epoll_ctlv` and `epoll_ctl` combine for a total of 1,949,404 calls compared with 3,947,769 `epoll_ctl` calls when using epoll-LT. While epoll-ctlv helps to reduce the number of user-kernel boundary crossings, the net result is no better than epoll-LT. The amount of time taken by epoll-ctlv in `epoll_ctlv` and `epoll_ctl` system calls is about the same (around 16%) as that spent by level-triggered epoll in invoking `epoll_ctl`.

When comparing the percentage of time epoll-LT and epoll-ET spend in `epoll_ctl` we see that it has been reduced using epoll-ET from 16% to 11%. Although the `epoll_ctl` time has been reduced it does not result in an appreciable improvement in throughput. We also note that about 2% of the run-time (which is not shown in the table) is also spent in the epoll-ET case checking, and tracking the state of the request (i.e., whether the server should be reading or writing) and the state of the socket (i.e., whether it is readable or writable). We expect that this can be reduced but that it wouldn't noticeably impact performance.

Results for the SPECweb99-like workload are shown in Figure 4. Here the graph shows that all techniques produce very similar results with a very slight performance advantage going to epoll-ET after the saturation point is reached.

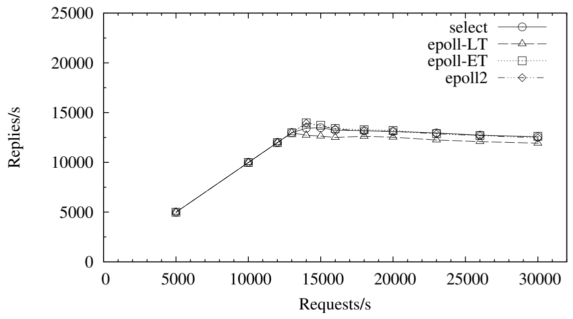


Figure 4: *µserver performance on SPECweb99-like workload with no idle connections*

5.1 Results With Idle Connections

We now compare the performance of the event mechanisms with 10,000 idle connections. The idle connections are intended to simulate the presence of larger numbers of simultaneous connections (as might occur in a WAN environment). Thus, the event dispatch mechanism has to keep track of a large number of descriptors even though only a very small portion of them are active.

By comparing results in Figures 3 and 5 one can see that the performance of select and poll degrade by up to 79% when the 10,000 idle connections are added. The performance of epoll2 with idle connections suffers similarly to select and poll. In this case, epoll2 suffers from the overheads incurred by making a large number of `epoll_wait` calls the vast majority of which return events that are not of current interest to the server. Throughput with level-triggered epoll is slightly reduced with the addition of the idle connections while edge-triggered epoll is not impacted.

The results for the SPECweb99-like workload with 10,000 idle connections are shown in Figure 6. In this case each of the event mechanisms is impacted in a manner similar to that in which they are impacted by idle connections in the one-byte workload case.

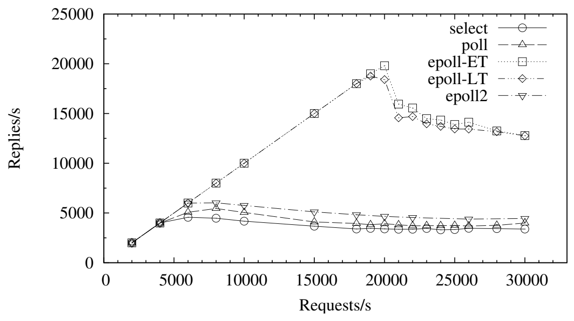


Figure 5: *µserver performance on one byte workload and 10,000 idle connections*

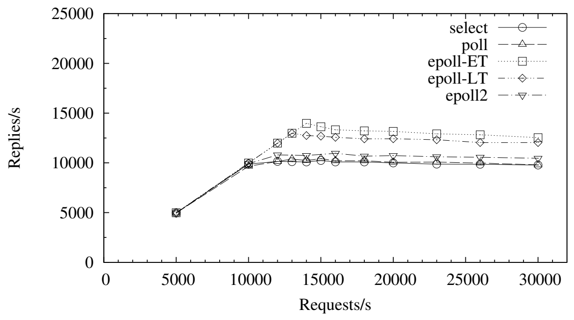


Figure 6: *µserver performance on SPECweb99-like workload and 10,000 idle connections*

5.2 Tuning Accept Strategy for epoll

The *µ*server's accept strategy has been tuned for use with `select`. The *µ*server includes a parameter that controls the number of connections that are accepted consecutively. We call this parameter the accept-limit. Parameter values range from one to infinity (Inf). A value of one limits the server to accepting at most one connection when notified of a pending connection request, while Inf causes the server to consecutively accept all currently pending connections.

To this point we have used the accept strategy that was shown to be effective for `select` by

Brecht et al. [4] (i.e., accept-limit is Inf). In order to verify whether the same strategy performs well with the epoll-based methods we explored their performance under different accept strategies.

Figure 7 examines the performance of level-triggered epoll after the accept-limit has been tuned for the one-byte workload (other values were explored but only the best values are shown). Level-triggered epoll with an accept limit of 10 shows a marked improvement over the previous accept-limit of Inf, and now matches the performance of select on this workload. The accept-limit of 10 also improves peak throughput for the epoll-ctlv model by 7%. This gap widens to 32% at 21,000 requests/sec. In fact the best accept strategy for epoll-ctlv fares slightly better than the best accept strategy for select.

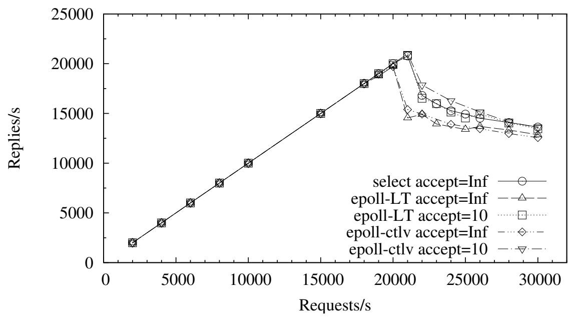


Figure 7: *µserver performance on one byte workload with different accept strategies and no idle connections*

Varying the accept-limit did not improve the performance of the edge-triggered epoll technique under this workload and it is not shown in the graph. However, we believe that the effects of the accept strategy on the various epoll techniques warrants further study as the efficacy of the strategy may be workload dependent.

6 Discussion

In this paper we use a high-performance event-driven HTTP server, the µserver, to compare and evaluate the performance of select, poll, and epoll event mechanisms. Interestingly, we observe that under some of the workloads examined the throughput obtained using `select` and `poll` is as good or slightly better than that obtained with epoll. While these workloads may not utilize representative numbers of simultaneous connections they do stress the event mechanisms being tested.

Our results also show that a main source of overhead when using level-triggered epoll is the large number of `epoll_ctl` calls. We explore techniques which significantly reduce the number of `epoll_ctl` calls, including the use of edge-triggered events and a system call, `epoll_ctlv`, which allows the µserver to aggregate large numbers of `epoll_ctl` calls into a single system call. While these techniques are successful in reducing the number of `epoll_ctl` calls they do not appear to provide appreciable improvements in performance.

As expected, the introduction of idle connections results in dramatic performance degradation when using `select` and `poll`, while not noticeably impacting the performance when using epoll. Although it is not clear that the use of idle connections to simulate larger numbers of connections is representative of real workloads, we find that the addition of idle connections does not significantly alter the performance of the edge-triggered and level-triggered epoll mechanisms. The edge-triggered epoll mechanism performs best with the level-triggered epoll mechanism offering performance that is very close to edge-triggered.

In the future we plan to re-evaluate some of the mechanisms explored in this paper under more representative workloads that include more representative wide area network conditions. The problem with the technique of using idle connections is that the idle connections simply inflate the number of connections without doing any useful work. We plan to explore tools similar to Dummynet [16] and NIST Net [5] in order to more accurately simulate traffic delays, packet loss, and other wide area network traffic characteristics, and to re-examine the performance of Internet servers using different event dispatch mechanisms and a wider variety of workloads.

7 Acknowledgments

We gratefully acknowledge Hewlett Packard, the Ontario Research and Development Challenge Fund, and the National Sciences and Engineering Research Council of Canada for financial support for this project.

References

[1] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[2] G. Banga and J.C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998.

[3] G. Banga, J.C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[4] Tim Brecht, David Pariag, and Louay Gammo. accept()able strategies for improving web server performance. In *Proceedings of the 2004 USENIX Annual Technical Conference (to appear)*, June 2004.

[5] M. Carson and D. Santay. NIST Net – a Linux-based network emulation tool. *Computer Communication Review*, to appear.

[6] A. Chandra and D. Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, 2001.

[7] HP Labs. The userver home page, 2004. Available at `http://hpl.hp.com/research/linux/userver`.

[8] Dan Kegel. The C10K problem, 2004. Available at `http://www.kegel.com/c10k.html`.

[9] Jonathon Lemon. Kqueue—a generic and scalable event notification facility. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2001.

[10] Davide Libenzi. Improving (network) I/O performance. Available at `http://www.xmailserver.org/linux-patches/nio-improve.html`.

[11] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *The First Workshop on Internet Server Performance*, pages 59–67, Madison, WI, June 1998.

[12] Shailabh Nagar, Paul Larson, Hanna Linder, and David Stevens. epoll scalability web page. Available at `http://lse.sourceforge.net/epoll/index.html`.

[13] M. Ostrowski. A mechanism for scalable event notification and delivery in Linux. Master's thesis, Department of Computer Science, University of Waterloo, November 2000.

[14] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999. `http://citeseer.nj.nec.com/article/pai99flash.html`.

[15] N. Provos and C. Lever. Scalable network I/O in Linux. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2000.

[16] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997. `http://citeseer.ist.psu.edu/rizzo97dummynet.html`.

[17] Standard Performance Evaluation Corporation. *SPECWeb99 Benchmark*, 1999. Available at `http://www.specbench.org/osg/web99`.

[18] David Weekly. /dev/epoll – a highspeed Linux kernel patch. Available at `http://epoll.hackerdojo.com`.

The (Re)Architecture of the X Window System

James Gettys
`jim.gettys@hp.com`
Keith Packard
`keithp@keithp.com`
HP Cambridge Research Laboratory

Abstract

The X Window System, Version 11, is the standard window system on Linux and UNIX systems. X11, designed in 1987, was "state of the art" at that time. From its inception, X has been a network transparent window system in which X client applications can run on any machine in a network using an X server running on any display. While there have been some significant extensions to X over its history (e.g. OpenGL support), X's design lay fallow over much of the 1990's. With the increasing interest in open source systems, it was no longer sufficient for modern applications and a significant overhaul is now well underway. This paper describes revisions to the architecture of the window system used in a growing fraction of desktops and embedded systems

1 Introduction

While part of this work on the X window system [SG92] is "good citizenship" required by open source, some of the architectural problems solved ease the ability of open source applications to print their results, and some of the techniques developed are believed to be in advance of the commercial computer industry. The challenges being faced include:

- X's fundamentally flawed font architecture made it difficult to implement good WYSIWYG systems
- Inadequate 2D graphics, which had always been intended to be augmented and/or replaced
- Developers are loathe to adopt any new technology that limits the distribution of their applications
- Legal requirements for accessibility for screen magnifiers are difficult to implement
- Users desire modern user interface eye candy, which sport translucent graphics and windows, drop shadows, etc.
- Full integration of applications into 3 D environments
- Collaborative shared use of X (e.g. multiple simultaneous use of projector walls or other shared applications)

While some of this work has been published elsewhere, there has never been any overview paper describing this work as an integrated whole, and the compositing manager work described below is novel as of fall 2003. This work represents a long term effort that started in 1999, and will continue for several years more.

2 Text and Graphics

X's obsolete 2D bit-blit based text and graphics system problems were most urgent. The development of the Gnome and KDE GUI environments in the period 1997-2000 had shown X11's fundamental soundness, but confirmed the authors' belief that the rendering system in X was woefully inadequate. One of us participated in the original X11 design meetings where the intent was to augment the rendering design at a later date; but the "GUI Wars" of the late 1980's doomed effort in this area. Good printing support has been particularly difficult to implement in X applications, as fonts have were opaque X server side objects not directly accessible by applications.

Most applications now composite images in sophisticated ways, whether it be in Flash media players, or subtly as part of anti-aliased characters. Bit-Blit is not sufficient for these applications, and these modern applications were (if only by their use of modern toolkits) all resorting to pixel based image manipulation. The screen pixels are retrieved from the window system, composited in clients, and then restored to the screen, rather than directly composited in hardware, resulting in poor performance. Inspired by the model first implemented in the Plan 9 window system, a graphics model based on Porter/Duff [PD84] image compositing was chosen. This work resulted in the X Render extension [Pac01a].

X11's core graphics exposed fonts as a server side abstraction. This font model was, at best, marginally adequate by 1987 standards. Even WYSIWYG systems of that era found them insufficient. Much additional information embedded in fonts (e.g. kerning tables) were not available from X whatsoever. Current competitive systems implement anti-aliased outline fonts. Discovering the Unicode coverage of a font, required by current toolkits for internationalization, was causing major performance problems. Deploying new server side font technology is slow, as X is a distributed system, and many X servers are seldom (or never) updated.

Therefore, a more fundamental change in X's architecture was undertaken: to no longer use server side fonts at all, but to allow applications direct access to font files and have the window system cache and composite glyphs onto the screen.

The first implementation of the new font system [Pac01b] taught a vital lesson. Xft1 provided anti-aliased text and proper font naming/substitution support, but reverted to the core X11 bitmap fonts if the Render extension was not present. Xft1 included the first implementation what is called "subpixel decimation," which provides higher quality subpixel based rendering than Microsoft's ClearType [Pla00] technology in a completely general algorithm.

Despite these advances, Xft1 received at best a lukewarm reception. If an application developer wanted anti-aliased text universally, Xft1 did not help them, since it relied on the Render extension which had not yet been widely deployed; instead, the developer would be faced with two implementations, and higher maintenance costs. This (in retrospect obvious) rational behavior of application developers shows the high importance of backwards compatibility; X extensions intended for application developers' use must be designed in a downward compatible form whenever possible, and should enable a complete conversion to a new facility, so that multiple code paths in applications do not need testing and maintenance. These principles have guided later development.

The font installation, naming, substitution, and internationalization problems were separated from Xft into a library named Fontconfig [Pac02], (since some printer only applications need this functionality independent of the window system.) Fontconfig provides internationalization features in advance of those in commercial systems such as Windows or OS X, and enables trivial font installation with good performance even when using thousands of fonts. Xft2 was also modified to operate against legacy X servers lacking the Render extension.

Xft2 and Fontconfig's solving of several major problems and lack of deployment barriers enabled rapid acceptance and deployment in the open source community, seeing almost universal use and uptake in less than one calendar year. They have been widely deployed on Linux systems since the end of 2002. They also "future proof" open source systems against coming improvements in font systems (e.g. OpenType), as the window system is no longer a gating item for font technology.

Sun Microsystems implemented a server side font extension over the last several years; for the reasons outlined in this section, it has not been adopted by open source developers.

While Xft2 and Fontconfig finally freed application developers from the tyranny of X11's core font system, improved performance [PG03], and at a stroke simplified their printing problems, it has still left a substantial burden on applications. The X11 core graphics, even augmented by the Render extension, lack convenient facilities for many applications for even simple primitives like splines, tasteful wide lines, stroking paths, etc, much less provide simple ways for applications to print the results on paper.

3 Cairo

The Cairo library [WP03], developed by one of the authors in conjunction with by Carl Worth of ISI, is designed to solve this problem. Cairo provides a state full user-level API with support for the PDF 1.4 imaging model. Cairo provides operations including stroking and filling Bézier cubic splines, transforming and compositing translucent images, and anti-aliased text rendering. The PostScript drawing model has been adapted for use within applications. Extensions needed to support much of the PDF 1.4 imaging operations have been included. This integration of the familiar PostScript operational model within the native application language environments provides a simple and powerful new tool for graphics application development.

Cairo's rendering algorithms use work done in the 1980's by Guibas, Ramshaw, and Stolfi [GRS83] along with work by John Hobby [Hob85], which has never been exploited in Postscript or in Windows. The implementation is fast, precise, and numerically stable, supports hardware acceleration, and is in advance of commercial systems.

Of particular note is the current development of Glitz [NR04], an OpenGL backend for Cairo, being developed by a pair of master's students in Sweden. Not only is it showing that a high speed implementation of Cairo is possible, it implements an interface very similar to the X Render extension's interface. More about this in the OpenGL section below.

Cairo is in the late stages of development and is being widely adopted in the open source community. It includes the ability to render to Postscript and a PDF back end is planned, which should greatly improve applications' printing support. Work to incorporate Cairo in the Gnome and KDE desktop environments is well underway, as are ports to Windows and Apple's MacIntosh, and it is being used by the Mono project. As with Xft2, Cairo works with all X servers, even those without the Render extension.

4 Accessibility and Eye-Candy

Several years ago, one of us implemented a prototype X system that used image compositing as the fundamental primitive for constructing the screen representation of the window hierarchy contents. Child window contents were composited to their parent windows which were incrementally composed to their parents until the final screen image was formed, enabling translucent windows. The problem with this simplistic model was twofold—first, a naïve implementation consumed enormous resources as each window required two complete off screen buffers (one for the window contents themselves, and one for the window contents composited with the children) and took huge amounts of time to build the final screen image as it recursively composited windows together. Secondly, the policy governing the compositing was hardwired into the X server. An architecture for exposing the same semantics with less overhead seemed almost possible, and pieces of it were implemented (miext/layer). However, no complete system was fielded, and every copy of the code tracked down and destroyed to prevent its escape into the wild.

Both Mac OS X and DirectFB [Hun04] perform window-level compositing by creating off-screen buffers for each top-level window (in OS X, the window system is not nested, so there are only top-level windows). The screen image is then formed by taking the resulting images and blending them together on the screen. Without handling the nested window case, both of these systems provide the desired functionality with a simple implementation. This simple approach is inadequate for X as some desktop environments nest the whole system inside a single top-level window to allow panning, and X's long history has shown the value of separating mechanism from policy (Gnome and KDE were developed over 10 years after X11's design). The fix is pretty easy—allow applications to select which pieces of the window hierarchy are to be stored off-screen and which are to be drawn to their parent storage.

With window hierarchy contents stored in off-screen buffers, an external application can now control how the screen contents are constructed from the constituent sub-windows and whatever other graphical elements are desired. This eliminated the complexities surrounding precisely what semantics would be offered in window-level compositing within the X server and the design of the underlying X extensions. They were replaced by some concerns over the performance implications of using an external agent (the "Compositing Manager") to execute the requests needed to present the screen image. Note that every visible pixel is under the control of the compositing manager, so screen updates are limited to how fast that application can get the bits painted to the screen.

The architecture is split across three new extensions:

- Composite, which controls which sub-hierarchies within the window tree are rendered to separate buffers.
- Damage, which tracks modified areas with windows, informing the Composting Manager which areas of the off-screen hierarchy components have changed.
- Xfixes, which includes new Region objects permitting all of the above computation to be performed indirectly within the

X server, avoiding round trips.

Multiple applications can take advantage of the off screen window contents, allowing thumbnail or screen magnifier applications to be included in the desktop environment.

To allow applications other than the compositing manager to present alpha-blended content to the screen, a new X Visual was added to the server. At 32 bits deep, it provides 8 bits of red, green and blue along with 8 bits of alpha value. Applications can create windows using this visual and the compositing manager can composite them onto the screen.

Nothing in this fundamental design indicates that it is used for constructing translucent windows; redirection of window contents and notification of window content change seems pretty far removed from one of the final goals. But note the compositing manger can use whatever X requests it likes to paint the combined image, including requests from the Render extension, which does know how to blend translucent images together. The final image is constructed programmatically so the possible presentation on the screen is limited only by the fertile imagination of the numerous eye-candy developers, and not restricted to any policy imposed by the base window system. And vital to rapid deployment, most applications can be completely oblivious to this background legerdemain.

In this design, such sophisticated effects need only be applied at frame update rates on only modified sections of the screen rather than at the rate applications perform graphics; this constant behavior is highly desirable in systems.

There is very strong "pull" from both commercial and non-commercial users of X for this work and the current early version will likely be shipped as part of the next X.org Foundation X Window System release, sometime this summer. Since there has not been sufficient exposure through widespread use, further changes will certainly be required further experience with the facilities are gained in a much larger audience; as these can be made without affecting existing applications, immediate deployment is both possible and extremely desirable.

The mechanisms described above realize a fundamentally more interesting architecture than either Windows or Mac OSX, where the compositing policy is hardwired into the window system. We expect a fertile explosion of experimentation, experience (both good and bad), and a winnowing of ideas as these facilities gain wider exposure.

5 Input Transformation

In the "naïve," eye-candy use of the new compositing functions, no transformation of input events are required, as input to windows remains at the same geometric position on the screen, even though the windows are first rendered off screen. More sophisticated use, for example, screen readers or immersive environments such as Croquet [SRRK02], or Sun's Looking Glass [KJ04] requires transformation of input events from where they first occur on the visible screen to the actual position in the windows (being rendered from off screen), since the window's contents may have been arbitrarily transformed or even texture mapped onto shapes on the screen.

As part of Sun Microsystem's award winning work on accessibility in open source for screen readers, Sun has developed the XEvIE extension [Kre], which allows external clients to transform input events. This looks like a good starting point for the somewhat more general problem that 3D systems pose, and with some modification can serve both the accessibility needs and those of more sophisticated applications.

6 Synchronization

Synchronization is probably the largest remaining challenge posed by compositing. While composite has eliminated much flashing of the screen since window exposure is eliminated, this does not solve the challenge of the compositing manager happening to copy an application's window to the frame buffer in the middle of an application painting a sequence of updates. No "tearing" of single graphics operations take place since the X server is single threaded, and all graphics operations are run to completion.

The X Synchronization extension (XSync) [GCGW92], widely available but to date seldom used, provides a general set of mechanisms for applications to synchronize with each other, with real time, and potentially with other system provided counters. XSync's original design intent intended system provided counters for vertical retrace interrupts, audio sample clocks, and similar system facilities, enabling very tight synchronization of graphics operations with these time bases. Work has begun on Linux to provide these counters at long last, when available, to flesh out the design originally put in place and tested in the early 1990's.

A possible design for solving the application synchronization problem at low overhead may be to mark sections of requests with increments of XSync counters: if the count is odd (or even) the window would be unstable/stable. The compositing manager might then copy the window only if the window is in a stable state. Some details and possibly extensions to XSync will need to be worked out, if this approach is pursued.

7 Next Steps

We believe we are slightly more than half way through the process of rearchitecting and reimplementing the X Window System. The existing prototype needs to become a production system requiring significant infrastructure work as described in this section.

7.1 OpenGL based X

Current X-based systems which support OpenGL do so by encapsulating the OpenGL environment within X windows. As such, an OpenGL application cannot manipulate X objects with OpenGL drawing commands.

Using OpenGL as the basis for the X server itself will place X objects such as pixmaps and off-screen window contents inside OpenGL objects allowing applications to use the full OpenGL command set to manipulate them.

A "proof of concept" of implementation of the X Render extension is being done as part of the Glitz back-end for Cairo, which is showing very good performance for render based applications. Whether the "core" X graphics will require any OpenGL extensions is still somewhat an open question.

In concert with the new compositing extensions, conventional X applications can then be integrated into 3D environments such as Croquet, or Sun's Looking Glass. X application contents can be used as textures and mapped onto any surface desired in those environments.

This work is underway, but not demonstrable at this date.

7.2 Kernel support for graphics cards

In current open source systems, graphics cards are supported in a manner totally unlike that of any other operating system, and unlike previous device drivers for the X Window System on commercial UNIX systems. There is no single central kernel driver responsible for managing access to the hardware. Instead, a large set of cooperating user and kernel mode systems are involved in mutual support of the hardware, including the X server (for 2D graphic), the direct-rendering infrastructure (DRI) (for accelerated 3D graphics), the kernel frame buffer driver (for text console emulation), the General ATI TV and Overlay Software (GATOS) (for video input and output) and alternate 2D graphics systems like DirectFB.

Two of these systems, the kernel frame buffer driver and the X server both include code to configure the graphics card "video mode"—the settings needed to send the correct video signals to monitors connected to the card. Three of these systems, DRI, the X server and GATOS, all include code for managing the memory space within the graphics card. All of these systems directly manipulate hardware registers without any coordination among them.

The X server has no kernel component for 2D graphics. Long-latency operations cannot use interrupts, instead the X server spins while polling status registers. DMA is difficult or impossible to configure in this environment. Perhaps the most egregious problem is that the X server reconfigures the PCI bus to correct BIOS mapping errors without informing the operating system kernel. Kernel access to devices while this remapping is going on may find the related devices mismapped.

To rationalize this situation, various groups and vendors are coordinating efforts to create a single kernel-level entity responsible for basic device management, but this effort has just begun.

7.3 Housecleaning and Latency Elimination and Latency Hiding

Serious attempts were made in the early 1990's to multi-thread the X server itself, with the discovery that the threading overhead in the X server is a net performance loss [Smi92].

Applications, however, often need to be multithreaded. The primary C binding to the X protocol is called Xlib, and its current implementation by one of us dates from 1987. While it was partially developed on a Firefly multiprocessor workstation of that era, something almost unheard of at that date, and some consideration of multi-threaded applications were taken in its implementation, its internal transport facilities were never expected/intended to be preserved when serious multi-threaded operating systems became available. Unfortunately, rather than a full rewrite as one of us expected, multi-threaded support was debugged into existence using the original code base and the resulting code is very bug-prone and hard to maintain. Additionally, over the years, Xlib became a "kitchen sink" library, including functionality well beyond its primary use as a binding to the X protocol. We have both seriously regretted the precedents both of us set introducing extraneous functionality into Xlib, causing it to be one of the largest libraries on UNIX/Linux systems. Due to better facilities in modern toolkits and system libraries, more than half of Xlib's current footprint is obsolete code or data.

While serious work was done in X11's design to mitigate latency, X's performance, particularly over low speed networks, is often limited by round trip latency, and with retrospect much more can be done [PG03]. As this work shows, client side fonts have made a significant improvement in startup latency, and work has already been completed in toolkits to mitigate some of the other hot spots. Much of the latency can be retrieved by some simple techniques already underway, but some require more sophisticated techniques that the current Xlib implementation is not capable of. Potentially 90the latency as of 2003 can be recovered by various techniques. The XCB library [MS01] by Bart Massey and Jamey Sharp is both carefully engineered to be multithreaded and to expose interfaces that will allow for latency hiding.

Since libraries linked against different basic X transport systems would cause havoc in the same address space, a Xlib compatibility layer (XCL) has been developed that provides the "traditional" X library API, using the original Xlib stubs, but replacing the internal transport and locking system, which will allow for much more useful latency hiding interfaces. The XCB/XCL version of Xlib is now able to run essentially all applications, and after a shakedown period, should be able to replace the existing Xlib transport soon. Other bindings than the traditional Xlib bindings then become possible in the same address space, and we may see toolkits adopt those bindings at substantial savings in space.

7.4 Mobility, Collaboration, and Other Topics

X's original intended environment included highly mobile students, and a hope, never generally realized for X, was the migration of applications between X servers.

The user should be able to travel between systems running X and retrieve your running applications (with suitable authentication and authorization). The user should be able to log out and "park" applications somewhere for later retrieval, either on the same display, or elsewhere. Users should be able to replicate an application's display on a wall projector for presentation. Applications should be able to easily survive the loss of the X server (most commonly caused by the loss of the underlying TCP connection, when running remotely).

Toolkit implementers typically did not understand and share this poorly enunciated vision and were primarily driven by pressing immediate needs, and X's design and implementation made migration or replication difficult to implement as an afterthought. As a result, migration (and replication) was seldom implemented, and early toolkits such as Xt made it even more difficult. Emacs is the only widespread application capable of both migration and replication, and it avoided using any toolkit. A more detailed description of this vision is available in [Get02].

Recent work in some of the modern toolkits (e.g. GTK+) and evolution of X itself make much of this vision demonstrable in current applications. Some work in the X infrastructure (Xlib) is underway to enable the prototype in GTK+ to be finished.

Similarly, input devices need to become full-fledged network data sources, to enable much looser coupling of keyboards, mice, game consoles and projectors and displays; the challenge here will be the authentication, authorization and security issues this will raise. The HAL and DBUS projects hosted at freedesktop.org are working on at least part of the solutions for the user interface challenges posed by hotplug of input devices.

7.5 Color Management

The existing color management facilities in X are over 10 years old, have never seen widespread use, and do not meet current needs. This area is ripe for revisiting. Marti Maria Saguer's LittleCMS [Mar] may be of use here. For the first time, we have the opportunity to "get it right" from one end to the other if we choose to make the investment.

7.6 Security and Authentication

Transport security has become an burning issue; X is network transparent (applications can run on any system in a network, using remote displays), yet we dare no longer use X over the network directly due to password grabbing kits in the hands of script kiddies. SSH [BS01] provides such facilities via port forwarding and is being used as a temporary stopgap. Urgent work on something better is vital to enable scaling and avoid the performance and latency issues introduced by transit of extra processes, particularly on (Linux Terminal Server Project (LTSP [McQ02]) servers, which are beginning break out of their initial use in schools and other non security sensitive environments into very sensitive commercial environments.

Another aspect of security arises between applications sharing a display. In the early and mid 1990's efforts were made as a result of the compartmented mode workstation projects to make it much more difficult for applications to share or steal data from each other on a X display. These facilities are very inflexible, and have gone almost unused.

As projectors and other shared displays become common over the next five years, applications from multiple users sharing a display will become commonplace. In such environments, different people may be using the same display at the same time and would like some level of assurance that their application's data is not being grabbed by the other user's application.

Eamon Walsh has, as part of the SELinux project [Wal04], been working to replace the existing X Security extension with an extension that, as in SELinux, will allow multiple different security policies to be developed external to the X server. This should allow multiple different policies to be available to suit the varied uses: normal workstations, secure workstations, shared displays in conference rooms, etc.

7.7 Compression and Image Transport

Many/most modern applications and desktops, including the most commonly used application (a web browser) are now intensive users of synthetic and natural images. The previous attempt (XIE [SSF$^+$96]) to provide compressed image transport failed due to excessive complexity and over ambition of the designers, has never been significantly used, and is now in fact not even shipped as part of current X distributions.

Today, many images are being read from disk or the network in compressed form, uncompressed into memory in the X client, moved to the X server (where they often occupy another copy of the uncompressed data). If we add general data compression to X (or run X over ssh with compression enabled) the data would be both compressed and uncompressed on its way to the X server. A simple replacement for XIE (if the complexity slippery slope can be avoided in a second attempt) would be worthwhile, along with other general compression of the X protocol.

Results in our 2003 Usenix X Network Performance paper show that, in real application workloads (the startup of a Gnome desktop), using even simple GZIP [Gai93] style compression can make a tremendous difference in a network environment, with a factor of 300(!) savings in bandwidth. Apparently the synthetic images used in many current UI's are extremely good candidates for compression. A simple X extension that could encapsulate one or more X requests into the extension request would avoid multiple compression/uncompression of the same data in the system where an image transport extension was also present. The basic X protocol framework is actually very byte efficient relative to most conventional RPC systems, with a basic X request only occupying 4 bytes (contrast this with HTTP or CORBA, in which a simple request is more than 100 bytes).

With the great recent interest in LTSP in commercial environments, work here would be extremely well spent, saving both memory and CPU, and network bandwidth.

We are more than happy to hear from anyone interested in helping in this effort to bring X into the new millennium.

References

[BS01] Daniel J. Barrett and Richard Silverman. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly & Associates, Inc., 2001.

[Gai93] Jean-Loup Gailly. *Gzip: The Data Compression Program*. iUniverse.com, 1.2.4 edition, 1993.

[GCGW92] Tim Glauert, Dave Carver, James Gettys, and David Wiggins. X Synchronization Extension Protocol, Version 3.0. X consortium standard, 1992.

[Get02] James Gettys. The Future is Coming, Where the X Window System Should Go. In *FREENIX Track, 2002 Usenix Annual Technical Conference*, Monterey, CA, June 2002. USENIX.

[GRS83] Leo Guibas, Lyle Ramshaw, and Jorge Stolfi. A kinetic framework for computational geometry. In *Proceedings of the IEEE 1983 24th Annual Symposium on the Foundations of Computer Science*, pages 100–111. IEEE Computer Society Press, 1983.

[Hob85] John D. Hobby. *Digitized Brush Trajectories*. PhD thesis, Stanford University, 1985. Also *Stanford Report STAN-CS-85-1070*.

[Hun04] A. Hundt. DirectFB Overview (v0.2 for DirectFB 0.9.21), February 2004. `http://www.directfb.org/documentation`.

[KJ04] H. Kawahara and D. Johnson. Project Looking Glass: 3D Desktop Exploration. In *X Developers Conference*, Cambridge, MA, April 2004.

[Kre] S. Kreitman. XEvIE - X Event Interception Extension. `http://freedesktop.org/~stukreit/xevie.html`.

[Mar] M. Maria. Little CMS Engine 1.12 API Definition. Technical report. `http://www.littlecms.com/lcmsapi.txt`.

[McQ02] Jim McQuillan. LTSP - Linux Terminal Server Project, Version 3.0. Technical report, March 2002. `http://www.ltsp.org/documentation/ltsp-3.0-4-en.html`.

[MS01] Bart Massey and Jamey Sharp. XCB: An X protocol c binding.

In *XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.

[NR04] Peter Nilsson and David Reveman. Glitz: Hardware Accelerated Image Compositing using OpenGL. In *FREENIX Track, 2004 Usenix Annual Technical Conference*, Boston, MA, July 2004. USENIX.

[Pac01a] Keith Packard. Design and Implementation of the X Rendering Extension. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.

[Pac01b] Keith Packard. The Xft Font Library: Architecture and Users Guide. In *XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.

[Pac02] Keith Packard. Font Configuration and Customization for Open Source Systems. In *2002 Gnome User's and Developers European Conference*, Seville, Spain, April 2002. Gnome.

[PD84] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.

[PG03] Keith Packard and James Gettys. X Window System Network Performance. In *FREENIX Track, 2003 Usenix Annual Technical Conference*, San Antonio, TX, June 2003. USENIX.

[Pla00] J. Platt. Optimal filtering for patterned displays. *IEEE Signal Processing Letters*, 7(7):179–180, 2000.

[SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

[Smi92] John Smith. The Multi-Threaded X Server. *The X Resource*, 1:73–89, Winter 1992.

[SRRK02] D. Smith, A. Raab, D. Reed, and A. Kay. Croquet: The Users Manual, October 2002. `http://glab.cs.uni-magdeburg.de/~croquet/downloads/Croquet0.1.pdf`.

[SSF$^+$96] Robert N.C. Shelley, Robert W. Scheifler, Ben Fahy, Jim Fulton, Keith Packard, Joe Mauro, Richard Hennessy, and Tom Vaughn. X Image Extension Protocol Version 5.02. X consortium standard, 1996.

[Wal04] Eamon Walsh. Integrating XFree86 With Security-Enhanced Linux. In *X Developers Conference*, Cambridge, MA, April 2004. `http://freedesktop.org/Software/XDevConf/x-security-walsh.pdf`.

[WP03] Carl Worth and Keith Packard. Xr: Cross-device Rendering for Vector Graphics. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, ON, July 2003. OLS.

IA64-Linux perf tools for IO dorks

Examples of IA-64 PMU usage

Grant Grundler
Hewlett-Packard
`iod00d@hp.com`
`grundler@parisc-linux.org`

Abstract

Itanium processors have very sophisticated performance monitoring tools integrated into the CPU. McKinley and Madison Itanium CPUs have over three hundred different types of events they can filter, trigger on, and count. The restrictions on which combinations of triggers are allowed is daunting and varies across CPU implementations. Fortunately, the tools hide this complicated mess. While the tools prevent us from shooting ourselves in the foot, it's not obvious how to use those tools for measuring kernel device driver behaviors.

IO driver writers can use pfmon to measure two key areas generally not obvious from the code: MMIO read and write frequency and precise addresses of instructions regularly causing L3 data cache misses. Measuring MMIO reads has some nuances related to instruction execution which are relevant to understanding ia64 and likely ia32 platforms. Similarly, the ability to pinpoint exactly which data is being accessed by drivers enables driver writers to either modify the algorithms or add prefetching directives where feasible. I include some examples on how I used pfmon to measure NIC drivers and give some guidelines on use.

q-syscollect is a "gprof without the pain" kind of tool. While q-syscollect uses the same kernel perfmon subsystem as pfmon, the former works at a higher level. With some knowledge about how the kernel operates, q-syscollect can collect call-graphs, function call counts, and percentage of time spent in particular routines. In other words, pfmon can tell us how much time the CPU spends stalled on d-cache misses and q-syscollect can give us the call-graph for the worst offenders.

Updated versions of this paper will be available from `http://iou.parisc-linux.org/ols2004/`

1 Introduction

Improving the performance of IO drivers is really not that easy. It usually goes something like:

1. Determine which workload is relevant
2. Set up the test environment
3. Collect metrics
4. Analyze the metrics
5. Change the code based on theories about the metrics
6. Iterate on Collect metrics

This paper attempts to make the collect-analyze-change loop more efficient for three obvious things: MMIO reads, MMIO writes, and cache line misses.

MMIO reads and writes are easier to locate in Linux code than for other OSs which support memory-mapped IO—just search for *readl()* and *writel()* calls. But `pfmon` [1] can provide statistics of actual behavior and not just where in the code MMIO space is touched.

Cache line misses are hard to detect. None of the regular performance tools I've used can precisely tell where CPU stalls are taking place. We can guess some of them based on data usage—like spin locks ping-ponging between CPUs. This requires a level of understanding that most of us mere mortals don't possess. Again, `pfmon` can help out here.

Lastly, getting an overview of system performance and getting run-time call graph usually requires compiler support that gcc doesn't provide. q-tools[4] can provide that information. Driver writers can then manually adjust the code knowing where the "hot spots" are.

1.1 `pfmon`

The author of `pfmon`, Stephane Eranian [2], describes `pfmon` as "the performance tool for IA64-Linux which exploits all the features of the IA-64 Performance Monitoring Unit (PMU)." `pfmon` uses a command line interface and does not require any special privilege to run. `pfmon` can monitor a single process, a multi-threaded process, multi-processes workloads and the entire system.

`pfmon` is the user command line interface to the kernel perfmon subsystem. perfmon does the ugly work of programming the PMU. Perfmon is versioned separately from `pfmon` command. When in doubt, use the perfmon in the latest 2.6 kernel.

There are two major types of measurements: counting and sampling. For counting, `pfmon` simply reports the number of occurrences of the desired events during the monitoring period. `pfmon` can also be configured to sample at certain intervals information about the execution of a command or for the entire system. It is possible to sample any events provided by the underlying PMU.

The information recorded by the PMU depends on what the user wants. `pfmon` contains a few preset measurements but for the most part the user is free to set up custom measurements. On Itanium2, `pfmon` provides access to all the PMU advanced features such as opcode matching, range restrictions, the Event Address Registers (EAR) and the Branch Trace Buffer.

1.2 `pfmon` command line options

Here is a summary of command line options used in the examples later in this paper:

–us-c use the US-style comma separator for large numbers.

–cpu-list=0 bind `pfmon` to CPU 0 and only count on CPU 0

–pin-command bind the command at the end of the command line to the same CPU as `pfmon`.

–resolve-addr look up addresses and print the symbols

–long-smpl-periods=2000 take a sample of every 2000th event.

–smpl-periods-random=0xfff:10 randomize the sampling period. This is necessary to avoid bias when sampling repetitive behaviors. The first value is the mask of bits to randomize (e.g., 0xfff) and the second value is initial seed (e.g., 10).

-k kernel only.

–system-wide measure the entire system (all processes and kernel)

Parameters only available on a to-be-released `pfmon` v3.1:

–smpl-module=dear-hist-itanium2 This particular module is to be used ONLY in conjunction with the Data EAR (Event Address Registers) and presents recorded samples as histograms about the cache misses. By default, the information is presented in the instruction view but it is possible to get the data view of the misses also.

-e data_ear_cache_lat64 pseudo event for memory loads with latency $\geq$ 64 cycles. The real event is `DATA_EAR_EVENT` (counts the number of times Data EAR has recorded something) and the pseudo event expresses the latency filter for the event. Use "`pfmon -ldata_ear_cache*`" to list all valid values. Valid values with McKinley CPU are powers of two (4 – 4096).

1.3 q-tools

The author of q-tools, David Mosberger [5], has described q-tools as "gprof without the pain."

q-tools package contains `q-syscollect`, `q-view`, `qprof`, and `q-dot`. `q-syscollect` collects profile information using kernel perfmon subsystem to sample the PMU. `q-view` will present the data collected in both flat-profile and call graph form. `q-dot` displays the call-graph in graphical form. Please see the `qprof` [6] website for details on `qprof`.

`q-syscollect` depends on the kernel perfmon subsystem which is included in all 2.6 Linux kernels. Because `q-syscollect` uses the PMU, it has the following advantages over other tools:

- no special kernel support needed (besides perfmon subsystem).
- provides call-graph of kernel functions
- can collect call-graphs of the kernel while interrupts are blocked.
- measures multi-threaded applications
- data is collected per-CPU and can be merged
- instruction level granularity (not bundles)

2 Measuring MMIO Reads

Nearly every driver uses MMIO reads to either flush MMIO writes, flush in-flight DMA, or (most obviously) collect status data from the IO device directly. While use of MMIO read is necessary in most cases, it should be avoided where possible.

2.1 Why worry about MMIO Reads?

MMIO reads are expensive—how expensive depends on speed of the IO bus, the number bridges the read (and its corresponding read return) has to cross, how "busy" each bus is, and finally how quickly the device responds to the read request. On most architectures, one can precisely measure the cost by measuring a loop of MMIO reads and calling `get_cycles()` before/after the loop.

I've measured anywhere from 1μs to 2μs per read. In practical terms:

- $\sim$ 500–600 cycles on an otherwise-idle 400 MHz PA-RISC machine.

- $\sim$ 1000 cycles on a 450 MHz Pentium machine which included crossing a PCI-PCI bridge.
- $\sim$ 900–1000 cycles on a 800 MHz IA64 HP ZX1 machine.

And for those who still don't believe me, try watching a DVD movie after turning DMA off for an IDE DVD player:

```
hdparm -d 0 /dev/cdrom
```

By switching the IDE controller to use PIO (Programmed I/O) mode, all data will be transferred to/from host memory under CPU control, byte (or word) at a time. `pfmon` can measure this. And `pfmon` looks broken when it displays three and four digit "Average Cycles Per Instruction" (CPI) output.

2.2 Eh? Memory Reads don't stall?

They do. But the CPU and PMU don't "realize" the stall until the next memory reference. The CPU continues execution until memory order is enforced by the acquire semantics in the MMIO read. This means the **Data Event Address Registers record the next stalled memory reference due to memory ordering constraints, not the MMIO read**. One has to look at the instruction stream carefully to determine which instruction actually caused the stall.

This also means the following sequence doesn't work exactly like we expect:

```
writel(CMD,addr);
readl(addr);
udelay(1);
y = buf->member;
```

The problem is the value returned by `read(x)` is never consumed. **Memory ordering imposes no constraint on non-load/store instructions.** Hence `udelay(1)` begins before the CPU stalls. The CPU will stall on `buf->member` because of memory ordering restrictions if the `udelay(1)` completes before `readl(x)` is retired. Drop the `udelay(1)` call and `pfmon` will always see the stall caused by MMIO reads on the next memory reference.

Unfortunately, the IA32 Software Developer's Manual[3] Volume 3, Chapter 7.2 "MEMORY ORDERING" is silent on the issue of how MMIO (uncached accesses) will (or will not) stall the instruction stream. This document is very clear on how "IO Operations" (e.g., IN/OUT) will stall the instruction pipeline until the read return arrives at the CPU. A direct response from Intel(R) indicated `readl()` does not stall like IN or OUT do and IA32 has the same problem. The Intel® architect who responded did hedge the above statement claiming a "udelay(10) will be as close as expected" for an example similar to mine. Anyone who has access to a frontside bus analyzer can verify the above statement by measuring timing loops between uncached accesses. I'm not that privileged and have to trust Intel® in this case.

For IA64, we considered putting an extra burden on `udelay` to stall the instruction stream until previous memory references were retired. We could use dummy loads/stores before and after the actual delay loop so memory ordering could be used to stall the instruction pipeline. That seemed excessive for something that we didn't have a bug report for.

Consensus was adding `mf.a` (memory fence) instruction to `readl()` should be sufficient. The architecture only requires `mf.a` serve as an ordering token and need not cause any delays of its own. In other words, the implementation is platform specific. `mf.a` has not been added to `readl()` yet because everything was working without so far.

2.3 `pfmon -e uc_loads_retired`

IO accesses are generally the only uncached references made on IA64-linux and normally will represent MMIO reads. The basic measurement will tell us roughly how many cycles the CPU stalls for MMIO reads. Get the number of MMIO reads per sample period and then multiply by the actual cycle counts a MMIO read takes for the given device. One needs to measure MMIO read cost by using a CPU internal cycle counter and hacking the kernel to read a harmless address from the target device a few thousand times.

In order to make statements about per transaction or per interrupt, we need to know the cumulative number of transactions or interrupts processed for the sample period. `pktgen` is straightforward in this regard since `pktgen` will print transaction statistics when a run is terminated. And one can record `/proc/interrupts` contents before and after each `pfmon` run to collect interrupt events as well.

Drawbacks to the above are one assumes a homogeneous driver environment; i.e., only one type of driver is under load during the test. I think that's a fair assumption for development in most cases. Bridges (e.g., routing traffic across different interconnects) are probably the one case it's not true. One has to work a bit harder to figure out what the counts mean in that case.

For other benchmarks, like SpecWeb, we want to grab `/proc/interrupt` and networking stats before/after `pfmon` runs.

2.4 tg3 Memory Reads

In summary, Figure 1 shows tg3 is doing $2749675/(1834959 - 918505) \approx 3$ MMIO reads per interrupt and averaging about $5000000/(1834959 - 918505) \approx 5$ packets per interrupt. This is with the BCM5701 chip running in PCI mode at 66MHz:64-bit.

Based on code inspection, here is a break down of where the MMIO reads occur in temporal order:

1. `tg3_interrupt()` flushes MMIO write to `MAILBOX_INTERRUPT_0`
2. `tg3_poll()` → `tg3_enable_ints()` → `tw32(TG3PCI_MISC_HOST_CTRL)`
3. `tg3_enable_ints()` flushes MMIO write to `MAILBOX_INTERRUPT_0`

It's obvious when inspecting `tw32()`, the BCM5701 chip has a serious bug. Every call to `tw32()` on BCM5701 requires a MMIO read to follow the MMIO write. Only writes to mailbox registers don't require this and a different routine is used for mailbox writes.

Given the NIC was designed for zero MMIO reads, this is pretty poor performance. Using a BCM5703 or BCM5704 would avoid the MMIO read in tw32().

I've exchanged email with David Miller and Jeff Garzik (tg3 driver maintainers). They have valid concerns with portability. We agree tg3 could be reduced to one MMIO read after the last MMIO write (to guarantee interrupts get re-enabled).

One would need to use the "tag" field in the status block when writing the mail box register to indicate which "tag" the CPU most recently

```
gsyprf3:~# pfmon -e uc_loads_retired -k --system-wide \
-- /usr/src/pktgen-testing/pktgen-single-tg3
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:     918505          0  IO-SAPIC-level  eth1
Result: OK: 7613693(c7613006+d687) usec, 5000000 (64byte) 656771pps 320Mb/sec
(336266752bps)  errors: 0
 57:    1834959          0  IO-SAPIC-level  eth1
CPU0                     2749675 UC_LOADS_RETIRED
CPU1                        1175 UC_LOADS_RETIRED
}
```

Figure 1: tg3 v3.6 MMIO reads with pktgen/IRQ on same CPU

saw. Using Message Signaled Interrupts (MSI) instead of Line based IRQs would guarantee the most recent status block update (transferred via DMA writes) would be visible to the CPU before `tg3_interrupt()` gets called.

The protocol would allow correct operation without using MSI, too.

2.5 Benchmarking, `pfmon`, and CPU bindings

The purpose of binding `pktgen` to CPU1 is to verify the transmit code path is NOT doing any MMIO reads. We split the transmit code path and interrupt handler across CPUs to narrow down which code path is performing the MMIO reads. This change is not obvious from Figure 2 output since tg3 only performs MMIO reads from CPU 0 (`tg3_interrupt()`).

But in Figure 2, performance goes up 30%! Offhand, I don’t know if this is due to CPU utilization (pktgen and `tg3_interrupt()` contending for CPU cycles) or if DMA is more efficient because of cache-line flows. When I don’t have any deadlines looming, I’d like to determine the difference.

2.6 e1000 Memory Reads

e1000 version 5.2.52-k4 has a more efficient implementation than tg3 driver. In a nut shell, MMIO reads are pretty much irrelevant to the pktgen workload with e1000 driver using default values.

Figure 3 shows e1000 performs $173315/(703829 - 622143) \approx 2$ MMIO reads per interrupt and $5000000/(703829 - 622143) \approx 61$ packets per interrupt.

Being the curious soul I am, I tracked down the two MMIO reads anyway. One is in the interrupt handler and the second when interrupts are re-enabled. It looks like e1000 will always need at least 2 MMIO reads per interrupt.

3 Measuring MMIO Writes

3.1 Why worry about MMIO Writes?

MMIO writes are clearly not as significant as MMIO reads. Nonetheless, every time a driver writes to MMIO space, some subtle things happen. There are four minor issues to think about: memory ordering, PCI bus utilization, filling outbound write queues, and stalling MMIO reads longer than necessary.

```
gsyprf3:~# pfmon -e uc_loads_retired -k --system-wide \
-- /usr/src/pktgen-testing/pktgen-single-tg3
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:    5809687          0  IO-SAPIC-level  eth1
Result: OK: 5914889(c5843865+d71024) usec, 5000000 (64byte) 845451pps 412Mb/se
c (432870912bps)  errors: 0
 57:    6427969          0  IO-SAPIC-level  eth1
CPU0                   1855253 UC_LOADS_RETIRED
CPU1                       950 UC_LOADS_RETIRED
```

Figure 2: tg3 v3.6 MMIO reads with pktgen/IRQ on diff CPU

```
gsyprf3:~# pfmon -e uc_loads_retired -k --system-wide \
-- /usr/src/pktgen-testing/pktgen-single-e1000
Configuring devices
Running... ctrl^C to stop
 59:     622143          0  IO-SAPIC-level  eth3
Result: OK: 10228738(c9990105+d238633) usec, 5000000 (64byte) 488854pps 238Mb/
sec (250293248bps)  errors: 81669
 59:     703829          0  IO-SAPIC-level  eth3
CPU0                    173315 UC_LOADS_RETIRED
CPU1                      1422 UC_LOADS_RETIRED
```

Figure 3: MMIO reads for e1000 v5.2.52-k4

First, memory ordering is enforced since PCI requires strong ordering of MMIO writes. This means the MMIO write will push all previous regular memory writes ahead. This is not a serious issue but it can make a MMIO write take longer.

MMIO writes are short transactions (i.e., much less than a cache-line). The PCI bus setup time to select the device, send the target address and data, and disconnect measurably reduces PCI bus utilization. It typically results in six or more PCI bus cycles to send four (or eight) bytes of data. On systems which strongly order DMA Read Returns and MMIO Writes, the latter will also interfere with DMA flows by interrupting in-flight, outbound DMA.

If the IO bridge (e.g., PCI Bus controller) nearest the CPU has a full write queue, the CPU will stall. The bridge would normally queue the MMIO write and then tell the CPU it's done. The chip designers normally make the write queue deep enough so the CPU never needs to stall. But drivers that perform many MMIO writes (e.g., use door bells) and burst many of MMIO writes at a time, could run into a worst case.

The last concern, stalling MMIO reads longer than normal, exists because of PCI ordering rules. MMIO reads and MMIO writes are strongly ordered. E.g., if four MMIO writes are queued before a MMIO read, the read will wait until all four MMIO write transactions have completed. So instead of say 1000 CPU cycles, the MMIO read might take more than 2000 CPU cycles on current platforms.

3.2 `pfmon -e uc_stores_retired`

`pfmon` counts MMIO Writes with no surprises.

3.3 tg3 Memory Writes

Figure 4 shows tg3 does about 10M MMIO writes to send 5M packets. However, we can break the MMIO writes down into base level (feed packets onto transmit queue) and `tg3_interrupt` which handles TX (and RX) completions. Knowing which code path the MMIO writes are in helps track down usage in the source code.

Output in Figure 5 is after hacking the `pktgen-single-tg3` script to bind `pktgen` kernel thread to CPU 1 when eth1 is directing interrupts to CPU 0. The distribution between TX queue setup and interrupt handling is obvious now. CPU 0 is handling interrupts and performs $3013580/(5803789 - 5201193) \approx 5$ MMIO writes per interrupt. CPU 1 is handling TX setup and performs $5000376/5000000 \approx 1$ MMIO write per packet.

Again, as noted in section 2.5, binding pktgen thread to one CPU and interrupts to another, changes the performance dramatically.

3.4 e1000 Memory Writes

Figure 6 shows $248891/(991082 - 908366) \approx 3$ MMIO writes per interrupt and $5001303/5000000 \approx 1$ MMIO write per packet. In other words, slightly better than tg3 driver. Nonetheless, the hardware can't push as many packets. One difference is the e1000 driver is pushing data to a NIC behind a PCI-PCI Bridge.

Figure 7 shows a $\approx$40% improvement in throughput[1] for pktgen without a PCI-PCI Bridge in the way. Note the ratios of MMIO writes per interrupt and MMIO writes per

[1]This demonstrates how the distance between the IO device and CPU (and memory) directly translates into latency and performance.

```
gsyprf3:~# pfmon -e uc_stores_retired -k --system-wide -- /usr/src/pktgen-test
ing/pktgen-single-tg3
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:    4284466          0  IO-SAPIC-level  eth1
Result: OK: 7611689(c7610900+d789) usec, 5000000 (64byte) 656943pps 320Mb/sec
(336354816bps)  errors: 0
 57:    5198436          0  IO-SAPIC-level  eth1
CPU0                     9570269 UC_STORES_RETIRED
CPU1                         445 UC_STORES_RETIRED
```

Figure 4: tg3 v3.6 MMIO writes

```
gsyprf3:~# pfmon -e uc_stores_retired -k --system-wide -- /usr/src/
pktgen-testing/pktgen-single-tg3
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:    5201193          0  IO-SAPIC-level  eth1
Result: OK: 5880249(c5811180+d69069) usec, 5000000 (64byte) 850340pps 415Mb
/sec (435374080bps)  errors: 0
 57:    5803789          0  IO-SAPIC-level  eth1
CPU0                     3013580 UC_STORES_RETIRED
CPU1                     5000376 UC_STORES_RETIRED
```

Figure 5: tg3 v3.6 MMIO writes with pktgen/IRQ split across CPUs

```
gsyprf3:~# pfmon -e uc_stores_retired -k --system-wide -- /usr/src/
pktgen-testing/pktgen-single-e1000
Running... ctrl^C to stop
 59:     908366          0  IO-SAPIC-level  eth3
Result: OK: 10340222(c10104719+d235503) usec, 5000000 (64byte) 483558pps 236Mb
/sec (247581696bps)  errors: 82675
 59:     991082          0  IO-SAPIC-level  eth3
CPU0                      248891 UC_STORES_RETIRED
CPU1                     5001303 UC_STORES_RETIRED
```

Figure 6: MMIO writes for e1000 v5.2.52-k4

```
gsyprf3:~# pfmon -e uc_stores_retired -k --system-wide -- /usr/src/pktgen-test
ing/pktgen-single-e1000
Running... ctrl^C to stop
 71:          3          0  IO-SAPIC-level  eth7
Result: OK: 7491358(c7342756+d148602) usec, 5000000 (64byte) 667467pps 325Mb/s
ec (341743104bps)  errors: 59870
 71:      59907          0  IO-SAPIC-level  eth7
CPU0                      180406 UC_STORES_RETIRED
CPU1                     5000939 UC_STORES_RETIRED
```

Figure 7: e1000 v5.2.52-k4 MMIO writes without PCI-PCI Bridge

packet are the same. I doubt the MMIO reads and MMIO writes are the limiting factors. More likely DMA access to memory (and thus TX/RX descriptor rings) limits NIC packet processing.

4 Measuring Cache-line Misses

The Event Address Registers[2] (EAR) can only record one event at a time. What is so interesting about them is that they record precise information about data cache misses. For instance for a data cache miss, you get the:

- address of the instruction, likely a load
- address of the target data
- latency in cycles to resolve the miss

The information pinpoints the source of the miss, not the consequence (i.e., the stall).

The Data EAR (DEAR) can also tell us about MMIO reads via sampling. The DEAR can only record *loads* that miss, not stores. Of course, MMIO reads always miss because they are uncached. This is interesting if we want to track down which MMIO addresses are "hot." It's usually easier to track down usage in source code knowing which MMIO address is referenced.

Collecting with DEAR sampling requires two parameters be tweaked to statistically improve the samples. One is the frequency at which Data Addresses are recorded and the other is the threshold (how many CPU cycles latency).

Because we know the latency to L3 is about 21 cycles, setting the EAR threshold to a value higher (e.g., 64 cycles) ensures only the load misses accessing main memory will be captured. This is how to select which level of cacheline misses one samples.

While high threshholds (e.g., 64 cycles) will show us where the longest delays occur, it will not show us the worst offenders. Doing a second run with a lower threshold (e.g., 4 cycles) shows all L1, L2, and L3 cache misses and provides a much broader picture of cache utilization.

When sampling events with low threshholds, we will get saturated with events and need to reduce the number of events actually sampled to every 5000th. The appropriate value will depend on the workload and how patient one is. The workload needs to be run long enough to be statistically significant and the sampling period needs to be high enough to not significantly perturb the workload.

4.1 tg3 Data Cache misses > 64 cycles

For the output in Figure 8, I've iteratively decreased the smpl-periods until I noticed the total pktgen throughput starting to drop. Figure 8 output only shows the tg3 interrupt code path since `pfmon` is bound to CPU 0. Normally, it would be useful to run this again with `cpu-list=1`. We could then see what the TX code path and pktgen are doing.

Also, the `pin-command` option in this example doesn't do anything since `pktgen-single-tg3` directs a `pktgen` kernel thread bound CPU 1 to do the real work. I've included the option only to make people aware of it.

4.2 tg3 Data Cache misses > 4 cycles

Figure 9 puts the `lat64` output in Figure 8 into better perspective. It shows tg3 is spending more time for L1 and L2 misses than L3 misses

[2]**`pfmon` v3.1 is the first version to support EAR and is expected to be available in August, 2004.**

```
gsyprf3:~# pfmon31 --us-c --cpu-list=0 --pin-command --resolve-addr \
           --smpl-module=dear-hist-itanium2 \
           -e data_ear_cache_lat64 --long-smpl-periods=500 \
           --smpl-periods-random=0xfff:10 --system-wide \
           -k -- /usr/src/pktgen-testing/pktgen-single-tg3
added event set 0
only kernel symbols are resolved in system-wide mode
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:    7209769          0  IO-SAPIC-level  eth1
Result: OK: 5915877(c5845032+d70845) usec, 5000000 (64byte) 845308pps 412Mb/sec
(432797696bps)  errors: 0
 57:    7827812          0  IO-SAPIC-level  eth1
# total_samples 672
# instruction addr view
# sorted by count
# showing per per distinct value
# %L2  : percentage of L1 misses that hit L2
# %L3  : percentage of L1 misses that hit L3
# %RAM : percentage of L1 misses that hit memory
# L2   :  5 cycles load latency
# L3   : 12 cycles load latency
# sampling period: 500
#count %self   %cum    %L2    %L3    %RAM   instruction addr
  38  5.65%   5.65%  0.00%  0.00% 100.00% 0xa000000100009141 ia64_spinlock_contention
                                                              +0x21<kernel>
  36  5.36%  11.01%  0.00%  0.00% 100.00% 0xa00000020003e580 tg3_interrupt[tg3]+0xe0<kernel>
  32  4.76%  15.77%  0.00%  0.00% 100.00% 0xa000000200034770 tg3_write_indirect_reg32[tg3]
                                                              +0x90<kernel>
  32  4.76%  20.54%  0.00%  0.00% 100.00% 0xa00000020003e640 tg3_interrupt[tg3]+0x1a0<kernel>
  30  4.46%  25.00%  0.00%  0.00% 100.00% 0xa000000200034e91 tg3_enable_ints[tg3]+0x91<kernel>
  29  4.32%  29.32%  0.00%  0.00% 100.00% 0xa00000020003e510 tg3_interrupt[tg3]+0x70<kernel>
  28  4.17%  33.48%  0.00%  0.00% 100.00% 0xa00000020003d1a0 tg3_tx[tg3]+0x2e0<kernel>
  27  4.02%  37.50%  0.00%  0.00% 100.00% 0xa00000020003cfa0 tg3_tx[tg3]+0xe0<kernel>
  24  3.57%  41.07%  0.00%  0.00% 100.00% 0xa00000020003cfd1 tg3_tx[tg3]+0x111<kernel>
  21  3.12%  44.20%  0.00%  0.00% 100.00% 0xa000000200034e60 tg3_enable_ints[tg3]+0x60<kernel>
   .
   .
   .
# level 0 : counts=0 avg_cycles=0.0ms   0.00%
# level 1 : counts=0 avg_cycles=0.0ms   0.00%
# level 2 : counts=672 avg_cycles=0.0ms 100.00%
approx cost: 0.0s
```

Figure 8: tg3 v3.6 lat64 output

```
gsyprf3:~# pfmon31 --us-c --cpu-list=0 --resolve-addr --smpl-module=dear-hist-itanium2 \
-e data_ear_cache_lat4 --long-smpl-periods=5000 --smpl-periods-random=0xfff:10 \
--system-wide -k -- /usr/src/pktgen-testing/pktgen-single-tg3
added event set 0
only kernel symbols are resolved in system-wide mode
Adding devices to run.
Configuring devices
Running... ctrl^C to stop
 57:    8484552          0  IO-SAPIC-level  eth1
Result: OK: 5938001(c5866437+d71564) usec, 5000000 (64byte) 842034pps 411Mb/sec
            (431121408bps)  errors: 0
 57:    9093642          0  IO-SAPIC-level  eth1
# total_samples 795
# instruction addr view
# sorted by count
# showing per per distinct value
# %L2  : percentage of L1 misses that hit L2
# %L3  : percentage of L1 misses that hit L3
# %RAM : percentage of L1 misses that hit memory
# L2   :  5 cycles load latency
# L3   : 12 cycles load latency
# sampling period: 5000
# #count %self    %cum     %L2     %L3    %RAM   instruction addr
      95  11.95%  11.95%   0.00%  98.95%   1.05% 0xa00000020003d150 tg3_tx[tg3]+0x290<kernel>
      83  10.44%  22.39%  93.98%   4.82%   1.20% 0xa00000020003d030 tg3_tx[tg3]+0x170<kernel>
      21   2.64%  25.03%   0.00%  95.24%   4.76% 0xa0000001000180f0 ia64_handle_irq+0x170<kernel>
      20   2.52%  27.55%   5.00%  80.00%  15.00% 0xa00000020003d040 tg3_tx[tg3]+0x180<kernel>
      18   2.26%  29.81%  50.00%  11.11%  38.89% 0xa00000020003cfa0 tg3_tx[tg3]+0xe0<kernel>
      17   2.14%  31.95%   0.00%   0.00% 100.00% 0xa00000020003e671 tg3_interrupt[tg3]
                                                                   +0x1d1<kernel>
      17   2.14%  34.09%   0.00% 100.00%   0.00% 0xa00000020003e700 tg3_interrupt[tg3]
                                                                   +0x260<kernel>
      16   2.01%  36.10%  56.25%  43.75%   0.00% 0xa000000100012160 ia64_leave_kernel
                                                                   +0x180<kernel>
      16   2.01%  38.11%  62.50%   0.00%  37.50% 0xa00000020003cf60 tg3_tx[tg3]+0xa0<kernel>
      15   1.89%  40.00%  86.67%   6.67%   6.67% 0xa00000020003cfd0 tg3_tx[tg3]+0x110<kernel>
      15   1.89%  41.89%   0.00%   0.00% 100.00% 0xa000000100016041 do_IRQ+0x1a1<kernel>
      15   1.89%  43.77%   0.00%  53.33%  46.67% 0xa00000020003e370 tg3_poll[tg3]+0x350<kernel>
      .
      .
      .
# level 0 : counts=226 avg_cycles=0.0ms  28.43%
# level 1 : counts=264 avg_cycles=0.0ms  33.21%
# level 2 : counts=305 avg_cycles=0.0ms  38.36%
approx cost: 0.0s
```

Figure 9: tg3 v3.6 lat4 output

and in only two locations. Adding one prefetch to pull data from L3 into L2 would help for the top offender. One needs to figure out which bit of data each recorded access refers to and determine how early one can prefetch that data.

We can also rule out MMIO accesses as the top culprit. `tg3_interrupt+0x1d1` could be an MMIO read but it doesn't show up in Figure 8 like `tg3_write_indirect_reg32` does.

Note `smpl-periods` is 10x higher in Figure 9 than in Figure 8. Collecting 10x more samples with `lat4` definitely disturbs the workload.

5 q-tools

`q-syscollect` and `q-view` are trivial to use. An example and brief explanation for kernel usage follow.

Please remember most applications spend most of the time in user space and not in the kernel. q-tools is especially good in user space.

5.1 q-syscollect

```
q-syscollect -c 5000 -C 5000 -t
20 -k
```

This will collect system wide kernel data during the 20 second period. Twenty to thrity seconds is usually long enough to get sufficient accuracy[3]. However, if the workload generates a very wide call graph with even distribution, one will likely need to sample for longer periods to get accuracy in the ±1% range. When in doubt, try sampling for longer periods to see if the call-counts change significantly.

[3]See Page 7 of the David Mosberger's Gelato talk [4] for a nice graph on accuracy which *only applies to his example.*

The `-c` and `-C` set the call sample rate and code sample rate respectively. The call sample rate is used to collect function call counts. This is one of the key differences compared to traditional profiling tools: q-syscollect obtains call-counts in a statistical fashion, just as has been done traditionally for the execution-time profile. The code sample rate is used to collect a flat profile (`CPU_CYCLES` by default).

The `-e` option allows one to change the event used to sample for the flat profile. The default is to sample CPU_CYCLES event. This provides traditional execution time in the flat profile.

The data is stored in the current directory under `.q/` directory. The next section demonstrates how `q-view` displays the data.

5.2 q-view

I was running the netperf [7] TCP_RR test in the background to another server when I collected the following data. As Figure 10 shows, this particular TCP_RR test isn't costing many cycles in tg3 driver. Or, at least not ones I can measure.

`tg3_interrupt()` shows up in the flat profile with 0.314 seconds time associated with it. The time measurement is only possible because `handle_IRQ_event()` re-enables interrupts if the IRQ handler is not registered with `SA_INTERRUPT` (to indicate latency sensitive IRQ handler). `do_IRQ()` and other functions in that same call graph do NOT have any time measurements because interrupts are disabled. As noted before, the call-graph is sampled using a different part of the PMU than the part which samples the flat profile.

Lastly, I've omitted the trailing output of `q-view` which explains the fields and columns more completely. Read that first be-

```
gsyprf3:~# q-view .q/kernel-cpu0.info | more
Flat profile of CPU_CYCLES in kernel-cpu0.hist#0:
 Each histogram sample counts as 200.510u seconds
% time      self     cumul     calls self/call  tot/call name
 68.88     13.41     13.41      215k     62.5u     62.5u default_idle
  2.90      0.56     13.97      431k     1.31u     1.31u finish_task_switch
  2.50      0.49     14.46      233k     2.09u     4.89u tg3_poll
  1.77      0.35     14.80     1.38M      251n      268n ipt_do_table
  1.61      0.31     15.12      240k     1.31u     1.31u tg3_interrupt
  1.51      0.29     15.41      240k     1.22u     5.95u net_rx_action
  .
  .
  .
Call-graph table:
index %time      self  children         called     name
                                                       <spontaneous>
[176]  69.4     30.5m      13.4         -              cpu_idle
                29.5m     0.285      231k/457k         schedule [164]
                10.0m      0.00      244k/244k         check_pgt_cache [178]
                 13.4      0.00      215k/215k         default_idle [177]
----------------------------------------------------
   .
   .
   .
----------------------------------------------------
                0.293      1.14      240k              __do_softirq [40]
[56]    7.4     0.293      1.14      240k              net_rx_action
                0.487     0.649      233k/233k         tg3_poll [57]
----------------------------------------------------
                0.487     0.649      233k              net_rx_action [56]
[57]    5.9     0.487     0.649      233k              tg3_poll
                    -      0.00      229k/229k         tg3_enable_ints [133]
                97.7m     0.552      225k/225k         tg3_rx [61]
                    -      0.00      227k/227k         tg3_tx [58]
----------------------------------------------------
   .
   .
   .
----------------------------------------------------
                    -      1.88      348k              ia64_leave_kernel [10]
[11]    9.7         -      1.88      348k              ia64_handle_irq
                    -      1.52      239k/240k         do_softirq [39]
                    -     0.367      356k/356k         do_IRQ [12]
----------------------------------------------------
   .
   .
   .
```

Figure 10: `q-view` output for TCP_RR over tg3 v3.6

fore going through the rest of the output.

6 Conclusion

6.1 More pfmon examples

CPU L2 cache misses in one kernel function

```
pfmon --verb -k \
--irange=sba_alloc_range \
-el2_misses --system-wide \
--session-timeout=10
```

Show all L2 cache misses in `sba_alloc_range`. This is interesting since `sba_alloc_range()` walks a bitmap to look for "free" resources. One can instead specify `-el3_misses` since L3 cache misses are much more expensive.

CPU 1 memory loads

```
pfmon --us-c \
--cpu-list=1 \
-e loads_retired \
-k --system-wide \
-- /tmp/pktgen-single
```

Only count memory loads on CPU 1. This is useful for when we can bind the interrupt to CPU 1 and the workload to a different CPU. This lets us separate interrupt path from base level code, i.e., when is the load happening (before or after DMA occurred) and which code path should one be looking more closely at.

List EAR events supported `pfmon -lear`
List all EAR types supported by `pfmon`[4].

More info on Event `pfmon -i DATA_EAR_TLB_ALL`
`pfmon` can provide more info on particular events it supports.

[4]EAR isn't supported until `pfmon v3.1`

6.2 And thanks to...

Special thanks to Stephane Eranian [2] for dedicating so much time to the perfmon kernel driver and associated tools. People might think the PMU does it all—but only with a lot of SW driving it. His review of this paper caught some good bloopers. This talk only happened because I sit across the aisle from him and could pester him regularly.

Thanks to David Mosberger[5] for putting together q-tools and making it so trivial to use.

In addition, in no particular order:
Christophe de Dinechin, Bjorn Helgaas, Matthew Wilcox, Andrew Patterson, Al Stone, Asit Mallick, and James Bottomley for reviewing this document or providing technical guidance.

Thanks also to the OLS staff for making this event happen every year.

My apologies if I omitted other contributors.

References

[1] perfmon homepage, `http://www.hpl.hp.com/research/linux/perfmon/`

[2] Stephane Eranian, `http://www.gelato.org/community/gelato_meeting.php?id=CU2004#talk22`

[3] The IA-32 Intel(R) Architecture Software Developer's Manuals, `http://www.intel.com/design/pentium4/manuals/253668.htm`

[4] q-tools homepage, `http://www.hpl.hp.com/`

research/linux/
q-tools/

[5] David Mosberger,
http://www.gelato.org/
community/gelato_
meeting.php?id=CU2004#
talk19

[6] qprof homepage,
http://www.hpl.hp.com/
research/linux/qprof/

[7] netperf homepage, http:
//www.netperf.org/

Carrier Grade Server Features in the Linux Kernel

Towards Linux-based Telecom Plarforms

Ibrahim Haddad
Ericsson Research
ibrahim.haddad@ericsson.com

Abstract

Traditionally, communications and data service networks were built on proprietary platforms that had to meet very specific availability, reliability, performance, and service response time requirements. Today, communication service providers are challenged to meet their needs cost-effectively for new architectures, new services, and increased bandwidth, with highly available, scalable, secure, and reliable systems that have predictable performance and that are easy to maintain and upgrade. This paper presents the technological trend of migrating from proprietary to open platforms based on software and hardware building blocks. It also focuses on the ongoing work by the Carrier Grade Linux working group at the Open Source Development Labs, examines the CGL architecture, the requirements from the latest specification release, and presents some of the needed kernel features that are not currently supported by Linux such as a Linux cluster communication mechanism, a low-level kernel mechanism for improved reliability and soft-realtime performance, support for multi-FIB, and support for additional security mechanisms.

1 Open platforms

The demand for rich media and enhanced communication services is rapidly leading to significant changes in the communication industry, such as the convergence of data and voice technologies. The transition to packet-based, converged, multi-service IP networks require a carrier grade infrastructure based on interoperable hardware and software building blocks, management middleware, and applications, implemented with standard interfaces. The communication industry is witnessing a technology trend moving away from proprietary systems toward open and standardized systems that are built using modular and flexible hardware and software (operating system and middleware) common off the shelf components. The trend is to proceed forward delivering next generation and multimedia communication services, using open standard carrier grade platforms. This trend is motivated by the expectations that open platforms are going to reduce the cost and risk of developing and delivering rich communications services. Also, they will enable faster time to market and ensure portability and interoperability between various components from different providers. One frequently asked question is: ’How can we meet tomorrow’s requirements using existing infrastructures and technologies?’. Proprietary platforms are closed systems, expensive to develop, and often lack support of the current and upcoming standards. Using such closed platforms to meet tomorrow’s requirements for new architectures and services is almost impossible. A uniform open software environment with the characteristics demanded by telecom applications, combined with commercial off-the-shelf software and hardware components is a necessary part of these new architectures. The following key industry consortia are defining hardware and software high availability specifications that are directly related to telecom platforms:

1. The PCI Industrial Computer Manufacturers Group [1] (PICMG) defines standards for high availability (HA) hardware.
2. The Open Source Development Labs [2] (OSDL) Carrier Grade Linux [3] (CGL) working group was established in January 2002 with the goal of enhancing the Linux operating system, to achieve an Open Source platform that is highly available, secure, scalable and easily maintained, suitable for carrier grade systems.
3. The Service Availability Forum [4] (SA Forum) defines the interfaces of HA middleware and focusing on APIs for hardware platform management and for application failover in the application API. SA compliant middleware will provide services to an application that needs to be HA in a portable way.

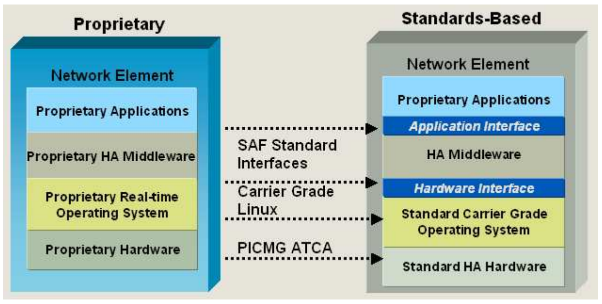


Figure 1: From Proprietary to Open Solutions

The operating system is a core component in such architectures. In the remaining of this paper, we will be focusing on CGL, its architecture and specifications.

2 The term Carrier Grade

In this paper, we refer to the term Carrier Grade on many occasions. Carrier grade is a term for public network telecommunications products that require a reliability percentage up to 5 or 6 nines of uptime.

- 5 nines refers to 99.999% of uptime per year (i.e., 5 minutes of downtime per year). This level of availability is usually associated with Carrier Grade servers.
- 6 nines refers to 99.9999% of uptime per year (i.e., 30 seconds of downtime per year). This level of availability is usually associated with Carrier Grade switches.

3 Linux versus proprietary operating systems

This section describes briefly the motivating reasons in favor of using Linux on Carrier Grade systems, versus continuing with proprietary operating systems. These motivations include:

- Cost: Linux is available free of charge in the form of a downloadable package from the Internet.
- Source code availability: With Linux, you gain full access to the source code allowing you to tailor the kernel to your needs.
- Open development process (Figure 2): The development process of the kernel is open to anyone to participate and contribute. The process is based on the concept of "release early, release often."
- Peer review and testing resources: With access to the source code, people using a

wide variety of platform, operating systems, and compiler combinations; can compile, link, and run the code on their systems to test for portability, compatibility and bugs.

- Vendor independent: With Linux, you no longer have to be locked into a specific vendor. Linux is supported on multiple platforms.
- High innovation rate: New features are usually implemented on Linux before they are available on commercial or proprietary systems.

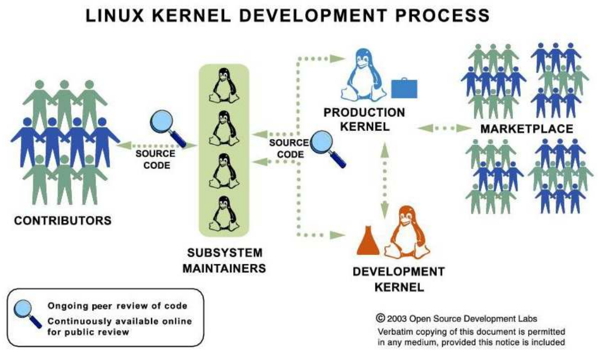

LINUX KERNEL DEVELOPMENT PROCESS
CONTRIBUTORS
SOURCE CODE
SUBSYSTEM MAINTAINERS
SOURCE CODE
PRODUCTION KERNEL
MARKETPLACE
DEVELOPMENT KERNEL
Ongoing peer review of code
Continuously available online for public review
© 2003 Open Source Development Labs
Verbatim copying of this document is permitted in any medium, provided this notice is included

Figure 2: Open development process of the Linux kernel

Other contributing factors include Linux' support for a broad range of processors and peripherals, commercial support availability, high performance networking, and the proven record of being a stable, and reliable server platform.

4 Carrier Grade Linux

The Linux kernel is missing several features that are needed in a telecom environment. It is not adapted to meet telecom requirements in various areas such as reliability, security, and scalability. To help the advancement of Linux in the telecom space, OSDL established the CGL working group. The group specifies and helps implement an Open Source platform targeted for the communication industry that is highly available, secure, scalable and easily maintained. The CGL working group is composed of several members from network equipment providers, system integrators, platform providers, and Linux distributors. They all contribute to the requirement definition of Carrier Grade Linux, help Open Source projects to meet these requirements, and in some cases start new Open Source projects. Many of the CGL members companies have contributed pieces of technologies to Open Source in order to make the Linux Kernel a more viable option for telecom platforms. For instance, the Open Systems Lab [5] from Ericsson Research has contributed three key technologies: the Transparent IPC [6], the Asynchronous Event Mechanism [7], and the Distributed Security Infrastructure [8]. There are already Linux distributions, MontaVista [9] for instance, that are providing CGL distribution based on the CGL requirement definition. Many companies are also either deploying CGL, or at least evaluating and experimenting with it.

Consequently, CGL activities are giving much momentum for Linux in the telecom space allowing it to be a viable option to proprietary operating system. Member companies of CGL are releasing code to Open Source and are making some of their proprietary technologies open, which leads to going forward from closed platforms to open platforms that use CGL Linux.

5 Target CGL applications

The CGL Working Group has identified three main categories of application areas into which they expect the majority of applications implemented on CGL platforms to fall. These application areas are gateways, signaling, and management servers.

- Gateways are bridges between two different technologies or administration domains. For example, a media gateway performs the critical function of converting voice messages from a native telecommunications time-division-multiplexed network, to an Internet protocol packet-switched network. A gateway processes a large number of small messages received and transmitted over a large number of physical interfaces. Gateways perform in a timely manner very close to hard real-time. They are implemented on dedicated platforms with replicated (rather than clustered) systems used for redundancy.

- Signaling servers handle call control, session control, and radio recourse control. A signaling server handles the routing and maintains the status of calls over the network. It takes the request of user agents who want to connect to other user agents and routes it to the appropriate signaling. Signaling servers require soft real time response capabilities less than 80 milliseconds, and may manage tens of thousands of simultaneous connections. A signaling server application is context switch and memory intensive due to requirements for quick switching and a capacity to manage large numbers of connections.

- Management servers handle traditional network management operations, as well as service and customer management. These servers provide services such as: a Home Location Register and Visitor Location Register (for wireless networks) or customer information (such as personal preferences including features the customer is authorized to use). Typically, management applications are data and communication intensive. Their response time requirements are less stringent by several orders of magnitude, compared to those of signaling and gateway applications.

6 Overview of the CGL working group

The CGL working group has the vision that next-generation and multimedia communication services can be delivered using Linux based open standards platforms for carrier grade infrastructure equipment. To achieve this vision, the working group has setup a strategy to define the requirements and architecture for the Carrier Grade Linux platform, develop a roadmap for the platform, and promote the development of a stable platform upon which commercial components and services can be deployed.

In the course of achieving this strategy, the OSDL CGL working group, is creating the requirement definitions, and identifying existing Open Source projects that support the roadmap to implement the required components and interfaces of the platform. When an Open Source project does not exist to support a certain requirement, OSDL CGL is launching (or support the launch of) new Open Source projects to implement missing components and interfaces of the platform.

The CGL working group consists of three distinct sub-groups that work together. These sub-groups are: specification, proof-of-concept, and validation. Responsibilities of each sub-group are as follows:

1. Specifications: The specifications sub-group is responsible for defining a set of

requirements that lead to enhancements in the Linux kernel, that are useful for carrier grade implementations and applications. The group collects, categorizes, and prioritizes the requirements from participants to allow reasonable work to proceed on implementations. The group also interacts with other standard defining bodies, open source communities, developers and distributions to ensure that the requirements identify useful enhancements in such a way, that they can be adopted into the base Linux kernel.

2. Proof-of-Concept: This sub-group generates documents covering the design, features, and technology relevant to CGL. It drives the implementation and integration of core Carrier Grade enhancements to Linux as identified and prioritized by the requirement document. The group is also responsible for ensuring the integrated enhancements pass, the CGL validation test suite and for establishing and leading an open source umbrella project to coordinate implementation and integration activities for CGL enhancements.

3. Validation: This sub-group defines standard test environments for developing validation suites. It is responsible for coordinating the development of validation suites, to ensure that all of the CGL requirements are covered. This group is also responsible for the development of an Open Source project CGL validation suite.

7 CGL architecture

Figure 3 presents the scope of the CGL Working Group, which covers two areas:

- Carrier Grade Linux: Various requirements such as availability and scalability are related to the CGL enhancements to the operating system. Enhancements may also be made to hardware interfaces, interfaces to the user level or application code and interfaces to development and debugging tools. In some cases, to access the kernel services, user level library changes will be needed.

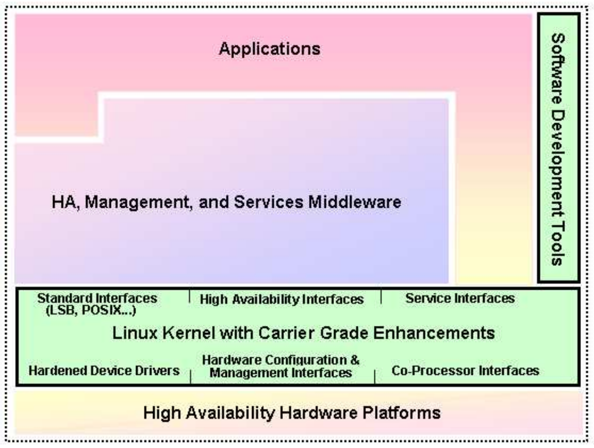

Applications
HA, Management, and Services Middleware
Software Development Tools
Standard Interfaces (LSB, POSIX...)
High Availability Interfaces
Service Interfaces
Linux Kernel with Carrier Grade Enhancements
Hardened Device Drivers
Hardware Configuration & Management Interfaces
Co-Processor Interfaces
High Availability Hardware Platforms

Figure 3: CGL architecture and scope

- Software Development Tools: These tools will include debuggers and analyzers.

  On October 9, 2003, OSDL announced the availability of the OSDL Carrier Grade Linux Requirements Definition, Version 2.0 (CGL 2.0). This latest requirement definition for next-generation carrier grade Linux offers major advances in security, high availability, and clustering.

8 CGL requirements

The requirement definition document of CGL version 2.0 introduced new and enhanced features to support Linux as a carrier grade platform. The CGL requirement definition divides the requirements in main categories described briefly below:

8.1 Clustering

These requirements support the use of multiple carrier server systems to provide higher levels of service availability through redundant resources and recovery capabilities, and to provide a horizontally scaled environment supporting increased throughput.

8.2 Security

The security requirements are aimed at maintaining a certain level of security while not endangering the goals of high availability, performance, and scalability. The requirements support the use of additional security mechanisms to protect the systems against attacks from both the Internet and intranets, and provide special mechanisms at kernel level to be used by telecom applications.

8.3 Standards

CGL specifies standards that are required for compliance for carrier grade server systems. Examples of these standards include:

- Linux Standard Base
- POSIX Timer Interface
- POSIX Signal Interface
- POSIX Message Queue Interface
- POSIX Semaphore Interface
- IPv6 RFCs compliance
- IPsecv6 RFCs compliance
- MIPv6 RFCs compliance
- SNMP support
- POSIX threads

8.4 Platform

OSDL CGL specifies requirements that support interactions with the hardware platforms making up carrier server systems. Platform capabilities are not tied to a particular vendor's implementation. Examples of the platform requirements include:

- Hot insert: supports hot-swap insertion of hardware components
- Hot remove: supports hot-swap removal of hardware components
- Remote boot support: supports remote booting functionality
- Boot cycle detection: supports detecting reboot cycles due to recurring failures. If the system experiences a problem that causes it to reboot repeatedly, the system will go offline. This is to prevent additional difficulties from occurring as a result of the repeated reboots
- Diskless systems: Provide support for diskless systems loading their kernel/application over the network
- Support remote booting across common LAN and WAN communication media

8.5 Availability

The availability requirements support heightened availability of carrier server systems, such as improving the robustness of software components or by supporting recovery from failure of hardware or software. Examples of these requirements include:

- RAID 1: support for RAID 1 offers mirroring to provide duplicate sets of all data on separate hard disks

- Watchdog timer interface: support for watchdog timers to perform certain specified operations when timeouts occur
- Support for Disk and volume management: to allow grouping of disks into volumes
- Ethernet link aggregation and link failover: support bonding of multiple NIC for bandwidth aggregation and provide automatic failover of IP addresses from one interface to another
- Support for application heartbeat monitor: monitor applications availability and functionality.

8.6 Serviceability

The serviceability requirements support servicing and managing hardware and software on carrier server systems. These are wide-ranging set requirements, put together, help support the availability of applications and the operating system. Examples of these requirements include:

- Support for producing and storing kernel dumps
- Support for dynamic debug to allow dynamically the insertion of software instrumentation into a running system in the kernel or applications
- Support for platform signal handler enabling infrastructures to allow interrupts generated by hardware errors to be logged using the event logging mechanism
- Support for remote access to event log information

8.7 Performance

OSDL CGL specifies the requirements that support performance levels necessary for the environments expected to be encountered by carrier server systems. Examples of these requirements include:

- Support for application (pre) loading.
- Support for soft real time performance through configuring the scheduler to provide soft real time support with latency of 10 ms.
- Support Kernel preemption.
- Raid 0 support: RAID Level 0 provides "disk striping" support to enhance performance for request-rate-intensive or transfer-rate-intensive environments

8.8 Scalability

These requirements support vertical and horizontal scaling of carrier server systems such as the addition of hardware resources to result in acceptable increases in capacity.

8.9 Tools

The tools requirements provide capabilities to facilitate diagnosis. Examples of these requirements include:

- Support the usage of a kernel debugger.
- Support for Kernel dump analysis.
- Support for debugging multi-threaded programs

9 CGL 3.0

The work on the next version of the OSDL CGL requirements, version 3.0, started in January 2004 with focus on advanced requirement areas such as manageability, serviceability, tools, security, standards, performance, hardware, clustering and availability. With the success of CGL's first two requirement documents, OSDL CGL working group anticipates that their third version will be quite beneficial to the Carrier Grade ecosystem. Official release of the CGL requirement document Version 3.0 is expected in October 2004.

10 CGL implementations

There are several enhancements to the Linux Kernel that are required by the communication industry, to help adopt Linux on their carrier grade platforms, and support telecom applications. These enhancements (Figure 4) fall into the following categories availability, security, serviceability, performance, scalability, reliability, standards, and clustering.

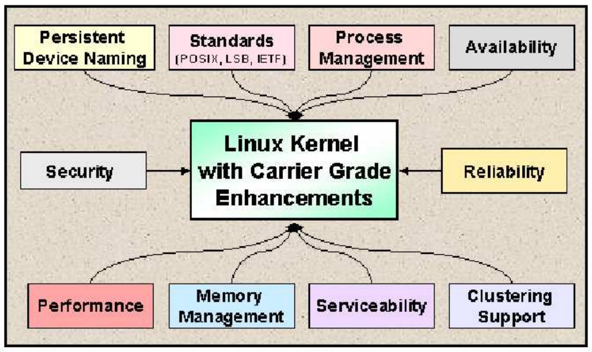


Figure 4: CGL enhancements areas

The implementations providing theses enhancements are Open Source projects and planned for integration with the Linux kernel when the implementations are mature, and ready for merging with the kernel code. In some cases, bringing some projects into maturity levels takes a considerable amount of time before being able to request its integration into the Linux kernel. Nevertheless, some of the enhancements are targeted for inclusion in kernel version 2.7. Other enhancements will follow in later kernel releases. Meanwhile, all enhancements, in the form of packages, kernel modules and patches, are available from their respective project web sites. The CGL 2.0 requirements are in-line with the Linux development community. The purpose of this project is to form a catalyst to capture common requirements from end-users for a CGL distribution. With a common set of requirements from the major Network Equipment Providers, developers can be much more productive and efficient within development projects. Many individuals within the CGL initiative are also active participants and contributors in the Open Source development community.

11 Examples of needed features in the Linux Kernel

In this section, we provide some examples of missing features and mechanisms from the Linux kernel that are necessary in a telecom environment.

11.1 Transparent Inter-Process and Inter-Processor Communication Protocol for Linux Clusters

Today's telecommunication environments are increasingly adopting clustered servers to gain benefits in performance, availability, and scalability. The resulting benefits of a cluster are greater or more cost-efficient than what a single server can provide. Furthermore, the telecommunications industry interest in clustering originates from the fact that clusters address carrier grade characteristics such as guaranteed service availability, reliability and scaled performance, using cost-effective hardware and software. Without being absolute about these requirements, they can be divided in these three categories: short failure detection and failure recovery, guaranteed availability of service, and short response times. The most widely adopted clustering technique is use of multiple interconnected loosely coupled nodes to create a single highly available system.

One missing feature from the Linux kernel in this area is a reliable, efficient, and transparent inter-process and inter-processor communication protocol. Transparent Inter Process Communication (TIPC) [6] is a suitable Open Source implementation that fills this gap and provides an efficient cluster communication protocol. This leverages the particular conditions present within loosely coupled clusters. It runs on Linux and is provided as a portable source code package implementing a loadable kernel module.

TIPC is unique because there seems to be no other protocol providing a comparable combination of versatility and performance. It includes some original innovations such as the functional addressing, the topology subscription services, and the reactive connection concept. Other important TIPC features include full location transparency, support for lightweight connections, reliable multicast, signaling link protocol, topology subscription services and more.

TIPC should be regarded as a useful toolbox for anyone wanting to develop or use Carrier Grade or Highly Available Linux clusters. It provides the necessary infrastructure for cluster, network and software management functionality, as well as a good support for designing site-independent, scalable, distributed, high-availability and high-performance applications.

It is also worthwhile to mention that the ForCES (Forwarding and Control Element WG) [11] working group within IETF has agreed that their router internal protocol (the ForCES protocol) must be possible to carry over different types of transport protocols. There is consensus on that TCP is the protocol to be used when ForCES messages are transported over the Internet, while TIPC is the protocol to be used in closed environments (LANs), where special characteristics such as high performance and multicast support is desirable. Other protocols may also be added as options.

TIPC is a contribution from Ericsson [5] to the Open Source community. TIPC was announced on LKML on June 28, 2004; it is licensed under a dual GPL and BSD license.

11.2 IPv4, IPv6, MIPv6 forwarding tables fast access and compact memory with multiple FIB support

Routers are core elements of modern telecom networks. They propagate and direct billion of data packets from their source to their destination using air transport devices or through high-speed links. They must operate as fast as the medium in order to deliver the best quality of service and have a negligible effect on communications. To give some figures, it is common for routers to manage between 10.000 to 500.000 routes. In these situations, good performance is achievable by handling around 2000 routes/sec. The actual implementation of the IP stack in Linux works fine for home or small business routers. However, with the high expectation of telecom operators and the new capabilities of telecom hardware, it appears as barely possible to use Linux as an efficient forwarding and routing element of a high-end router for large network (core/border/access router) or a high-end server with routing capabilities.

One problem with the networking stack in Linux is the lack of support for multiple forward-ing information bases (multi-FIB) wit h overlapping interface's IP address, and the lack of appropriate interfaces for addressing FIB. Another problem with the curren t implementation is the limited scalability of the routing table.

The solution to these problems is to provide support for multi-FIB with overlapping IP address. As such, we can have on differe nt VLAN or different physical interfaces, independent network in the same Linux box. For example, we can have two HTTP servers serving two different networks with potentially the same IP address. One HTTP server will serve the network/FIB 10, and the othe r HTTP server will serves the network/FIB 20. The advantage gained is to have one Linux box serving two different customers usi ng the same IP address. ISPs adopt this approach by providing services for multiple customers sharing the same server (server pa rtitioning), instead of using a server per customer.

The way to achieve this is to have an ID (an identifier that identifies the customer or user of the service) to completely separ ate the routing table in memory. Two approaches exist: the first is to have a separate routing tables, each routing table is looked up by their ID and within tha t table the lookup is done one the prefix. The second approach is to have one table, and the lookup is done on the combined key = prefix + ID.

A different kind of problem arises when we are not able to predict access time, with the chaining in the hash table of the routi ng cache (and FIB). This problem is of particular inter-est in an environment that requires predictable performance.

Another aspect of the problem is that the route cache and the routing table are not kept synchronized most of the time (path MTU, just to name one). The route cache flush is executed regularly; therefore, any updates on the cache are lost. For example, if you have a routing cache flush, you have to rebuild every route that you are currently talking to, by going for every route in the hash/try table and rebuilding the information. First, you have to lookup in the routing cache, and if you have a miss, then you need to go in the hash/try table. This process is very slow and not predictable since the hash/try table is implemented wi th linked list and there is high potential for collisions when a large number of routes are present. This design is suitable fo r a home PC with a few routes, but it is not scalable for a large server.

To support the various routing requirements of server nodes operating in high performance and mission critical envrionments, Linux should support the following:

- Implementation of multi-FIB using tree (radix, patricia, etc.): It is very important to have predictable performance in insert/delete/lookup from 10.000 to 500.000 routes. In addition, it is favourable to have the same data structure for both IPv4 and IPv6.
- Socket and ioctl interfaces for addressing multi-FIB.
- Multi-FIB support for neighbors (arp).

Providing these implementations in Linux will affect a large part of net/core, net/ipv4 and net/ipv6; these subsystems (mostly network layer) will need to be re-written. Other areas will have minimal impact at the source code level, mostly at the transport layer (socket, TCP, UDP, RAW, NAT, IPIP, IGMP, etc.).

As for the availability of an Open Source project that can provide these functionalities, there exists a project called "Linux Virtual Routing and Forwarding" [12]. This project aims to implement a flexible and scalable mechanism for providing multiple routing instances within the Linux kernel. The project has some potential in providing the needed functionalities, however no progress has been made since 2002 and the project seems to be inactive.

11.3 Run-time Authenticity Verification for Binaries

Linux has generally been considered immune to the spread of viruses, backdoors and Trojan programs on the Internet. However, with the increasing popularity of Linux as a desktop platform, the risk of seeing viruses or Trojans developed for this platform are rapidly growing. To alleviate this problem, the system should prevent on run time the execution of un-trusted software. One solution is to digitally sign the trusted binaries and have the system check the digital signature of binaries before running them. Therefore, untrusted (not signed) binaries are denied the execution. This can improve the security of the system by avoiding a wide range of malicious binaries like viruses, worms, Trojan programs and backdoors from running on the system.

DigSig [13] is a Linux kernel module that checks the signature of a binary before running it. It inserts digital signatures inside the ELF binary and verifies this signature before loading the binary. It is based on the Linux Security Module hooks (LSM has been integrated with the Linux kernel since 2.5.X and higher).

Typically, in this approach, vendors do not sign binaries; the control of the system remains with the local administrator. The responsible administrator is to sign all binaries they trust with their private key. Therefore, DigSig guarantees two things: (1) if you signed a binary, nobody else other than yourself can modify that binary without being detected. (2) Nobody can run a binary which is not signed or badly signed.

There has already been several initiatives in this domain, such as Tripwire [14], BSign [15], Cryptomark [16], but we believe the DigSig project is the first to be both easily accessible to all (available on SourceForge, under the GPL license) and to operate at kernel level on run time. The run time is very important for Carrier Grade Linux as this takes into account the high availability aspects of the system.

The DigSig approach has been using existing solutions like GnuPG [17] and BSign (a Debian package) rather than reinventing the wheel. However, in order to reduce the overhead in the kernel, the DigSig project only took the minimum code necessary from GnuPG. This helped much to reduce the amount of code imported to the kernel in source code of the original (only 1/10 of the original GnuPG 1.2.2 source code has been imported to the kernel module).

DigSig is a contribution from Ericsson [5] to the Open Source community. It was released under the GPL license and it is available from [8].

DigSig has been announced on LKML [18] but it not yet integrated in the Linux Kernel.

11.4 Efficient Low-Level Asynchronous Event Mechanism

Carrier grade systems must provide a 5-nines availability, a maximum of five minutes per year of downtime, which includes hardware, operating system, software upgrade and maintenance. Operating systems for such systems must ensure that they can deliver a high response rate with minimum downtime. In addition, carrier-grade systems must take into account such characteristics such as scalability, high availability and performance. In carrier grade systems, thousands of requests must be handled concurrently without affecting the overall system's performance, even under extremely high loads. Subscribers can expect some latency time when issuing a request, but they are not willing to accept an unbounded response time. Such transactions are not handled instantaneously for many reasons, and it can take some milliseconds or seconds to reply. Waiting for an answer reduces applications abilities to handle other transactions.

Many different solutions have been envisaged to improve Linux's capabilities in this area using different types of software organization, such as multithreaded architectures, implementing efficient POSIX interfaces, or improving the scalability of existing kernel routines.

One possible solution that is adequate for carrier grade servers is the Asynchronous Event Mechanism (AEM), which provides asynchronous execution of processes in the Linux kernel. AEM implements a native support for asynchronous events in the Linux kernel and aims to bring carrier-grade characteristics to Linux in areas of scalability and soft real-time responsiveness. In addition, AEM offers event-based development framework, scalability, flexibility, and extensibility.

Ericsson [5] released AEM to Open Source in February 2003 under the GPL license. AEM was announced on the Linux Kernel Mailing List (LKML) [20], and received feedback that resulted in some changes to the design and implementation. AEM is not yet integrated with the Linux kernel.

12 Conclusion

There are many challenges accompanying the migration from proprietary to open platforms. The main challenge remains to be the availability of the various kernel features and mechanisms needed for telecom platforms and integrating these features in the Linux kernel.

References

[1] PCI Industrial Computer Manufacturers Group,
`http://www.picmg.org`

[2] Open Source Development Labs,
`http://www.osdl.org`

[3] Carrier Grade Linux,
`http://osdl.org/lab_activities`

[4] Service Availability Forum,
`http://www.saforum.org`

[5] Open System Lab,
`http://www.linux.ericsson.ca`

[6] Transparent IPC,
`http://tipc.sf.net`

[7] Asynchronous Event Mechanism,
`http://aem.sf.net`

[8] Distributed Security Infrastructure,
`http://disec.sf.net`

[9] MontaVista Carrier Grade Edition,
`http://www.mvista.com/cge`

[10] Make Clustering Easy with TIPC, LinuxWorld Magazine, April 2004

[11] IETF ForCES working group,
`http://www.sstanamera.com/~forces`

[12] Linux Virtual Routing and Forwarding project,
`http://linux-vrf.sf.net`

[13] Stop Malicious Code Execution at Kernel Level, LinuxWorld Magazine, January 2004

[14] Tripwire,
`http://www.tripwire.com`

[15] Bsign,
`http://packages.debian.org/bsign`

[16] Cryptomark,
`http://immunix.org/cryptomark.html`

[17] GnuPG,
`http://www.gnupg.org`

[18] DigSig announcement on LKML,
`http://lwn.net/Articles/51007`

[19] An Event Mechanism for Linux, Linux Journal, July 2003

[20] AEM announcement on LKML,
`http://lwn.net/Articles/45633`

Acknowledgments

Thank you to Ludovic Beliveau, Mathieu Giguere, Magnus Karlson, Jon Maloy, Mats Naslund, Makan Pourzandi, and Frederic Rossi, for their valuable contributions and reviews.

Demands, Solutions, and Improvements for Linux Filesystem Security

Michael Austin Halcrow
International Business Machines, Inc.
`mike@halcrow.us`

Abstract

Securing file resources under Linux is a team effort. No one library, application, or kernel feature can stand alone in providing robust security. Current Linux access control mechanisms work in concert to provide a certain level of security, but they depend upon the integrity of the machine itself to protect that data. Once the data leaves that machine, or if the machine itself is physically compromised, those access control mechanisms can no longer protect the data in the filesystem. At that point, data privacy must be enforced via encryption.

As Linux makes inroads in the desktop market, the need for transparent and effective data encryption increases. To be practically deployable, the encryption/decryption process must be secure, unobtrusive, consistent, flexible, reliable, and efficient. Most encryption mechanisms that run under Linux today fail in one or more of these categories. In this paper, we discuss solutions to many of these issues via the integration of encryption into the Linux filesystem. This will provide access control enforcement on data that is not necessarily under the control of the operating environment. We also explore how stackable filesystems, Extended Attributes, PAM, GnuPG web-of-trust, supporting libraries, and applications (such as GNOME/KDE) can all be orchestrated to provide robust encryption-based access control over filesystem content.

1 Development Efforts

This paper is motivated by an effort on the part of the IBM Linux Technology Center to enhance Linux filesystem security through better integration of encryption technology. The author of this paper is working together with the external community and several members of the LTC in the design and development of a transparent cryptographic filesystem layer in the Linux kernel. The “we” in this paper refers to immediate members of the author’s development team who are working together on this project, although many others outside that development team have thus far had a significant part in this development effort.

2 The Filesystem Security

2.1 Threat Model

Computer users tend to be overly concerned about protecting their credit card numbers from being sniffed as they are transmitted over the Internet. At the same time, many do not think twice when sending equally sensitive information in the clear via an email message. A thief who steals a removable device, laptop, or server can also read the confidential files on those devices if they are left unprotected. Nevertheless, far too many users neglect to take the necessary steps to protect their files from such an event. Your liability limit for unauthorized charges to your credit card is $50 (and most credit card companies waive that liability for victims of fraud); on the other hand, confidentiality cannot be restored once lost.

Today, we see countless examples of neglect to use encryption to protect the integrity and the confidentiality of sensitive data. Those who are trusted with sensitive information routinely send that information as unencrypted email attachments. They also store that information in clear text on disks, USB keychain drives, backup tapes, and other removable media. GnuPG[7] and OpenSSL[8] provide all the encryption tools necessary to protect this information, but these tools are not used nearly as often as they ought to be.

If required to go through tedious encryption or decryption steps every time they need to work with a file or share it, people will select insecure passwords, transmit passwords in an insecure manner, fail to consider or use public key encryption options, or simply stop encrypting their files altogether. If security is overly obstructive, people will remove it, work around it, or misuse it (thus rendering it less effective). As Linux gains adoption in the desktop market, we need integrated file integrity and confidentiality that is seamless, transparent, easy to use, and effective.

2.2 Integration of File Encryption into the Filesystem

Several solutions exist that solve separate pieces of the problem. In one example highlighting transparency, employees within an organization that uses IBM™ Lotus Notes™ [9] for its email will not even notice the complex PKI or the encryption process that is integrated into the product. Encryption and decryption of sensitive email messages is seamless to the end user; it involves checking an "Encrypt" box, specifying a recipient, and sending the message. This effectively addresses a significant file in-transit confidentiality problem. If the local replicated mailbox database is also encrypted, then it also addresses confidentiality on the local storage device, but the protection is lost once the data leaves the domain of Notes (for example, if an attached file is saved to disk). The process must be seamlessly integrated into *all* relevant aspects of the user's operating environment.

In Section 4, we discuss filesystem security in general under Linux, with an emphasis on confidentiality and integrity enforcement via cryptographic technologies. In Section 6, we propose a mechanism to integrate encryption of files at the filesystem level, including integration of GnuPG[7] web-of-trust, PAM[10], a stackable filesystem model[2], Extended Attributes[6], and libraries and applications, in order to make the entire process as transparent as possible to the end user.

3 A Team Effort

Filesystem security encompasses more than just the filesystem itself. It is a team effort, involving the kernel, the shells, the login processes, the filesystems, the applications, the administrators, and the users. When we speak of "filesystem security," we refer to the security of the files in a filesystem, no matter what ends up providing that security.

For any filesystem security problem that exists, there are usually several different ways of solving it. Solutions that involve modifications in the kernel tend to introduce less overhead. This is due to the fact that context switches and copying of data between kernel and user memory is reduced. However, changes in the kernel may reduce the efficiency of the kernel's VFS while making it both harder to maintain and more bug-prone. As notable exceptions, Erez Zadok's stackable filesystem framework, FiST[3], and Loop-aes, require no change to the current Linux kernel VFS. Solutions that exist entirely in userspace do not complicate the kernel, but they tend to have more overhead and may be limited in the functionality they are able to provide, as they are limited by the interface to the kernel from userspace. Since they are in userspace, they are also more prone to attack.

4 Aspects of Filesystem Security

Computer security can be decomposed into several areas:

- Identifying who you are and having the machine recognize that identification (*authentication*).
- Determining whether or not you should be granted access to a resource such as a sensitive file (*authorization*). This is often based on the permissions associated with the resource by its owner or an administrator (*access control*).
- Transforming your data into an encrypted format in order to make it prohibitively costly for unauthorized users to decrypt and view (*confidentiality*).
- Performing checksums, keyed hashes, and/or signing of your data to make unauthorized modifications of your data detectable (*integrity*).

4.1 Filesystem Integrity

When people consider filesystem security, they traditionally think about access control (file permissions) and confidentiality (encryption). File integrity, however, can be just as important as confidentiality, if not more so. If a script that performs an administrative task is altered in an unauthorized fashion, the script may perform actions that violate the system's security policies. For example, many rootkits modify system startup and shutdown scripts to facilitate the attacker's attempts to record the user's keystrokes, sniff network traffic, or otherwise infiltrate the system.

More often than not, the value of the data stored in files is greater than that of the machine that hosts the files. For example, if an attacker manages to insert false data into a financial report, the alteration to the report may go unnoticed until substantial damage has been done; jobs could be at stake and in more extreme cases even criminal charges against the user could result . If trojan code sneaks into the source repository for a major project, the public release of that project may contain a backdoor.[1]

Many security professionals foresee a nightmare scenario wherein a widely propagated Internet worm quietly alters the contents of word processing and spreadsheet documents. Without any sort of integrity mechanism in place in the vast majority of the desktop machines in the world, nobody would know if any data that traversed vulnerable machines could be trusted. This threat could be very effectively addressed with a combination of a kernel-level mandatory access control (MAC)[11] protection profile and a filesystem that provides integrity and auditing capabilities. Such a combination would be resistant to damage done by a root compromise, especially if aided by a Trusted Platform Module (TPM)[13] using attestation.

[1] A high-profile example of an attempt to do this occurred with the Linux kernel last year. Fortunately, the source code management process used by the kernel developers allowed them to catch the attempted insertion of the trojan code before it made it into the actual kernel.

One can approach filesystem integrity from two angles. The first is to have strong authentication and authorization mechanisms in place that employ sufficiently flexible policy languages. The second is to have an auditing mechanism, to detect unauthorized attempts at modifying the contents of a filesystem.

4.1.1 Authentication and Authorization

The filesystem must contain support for the kernel's security structure, which requires stateful security attributes on each file. Most GNU/Linux applications today use PAM[10] (see Section 4.1.2 below) for authentication and process credentials to represent their authorization; policy language is limited to what can be expressed using the file owner and group, along with the owner/group/world read/write/execute attributes of the file. The administrator and the current owner have the authority to set the owner of the file or the read/write/execute policies for that file. In many filesystems, files may also contain additional security flags, such as an immutable or append-only flag.

Posix Access Control Lists (ACL's)[6] provide for more stringent delegations of access authority on a per-file basis. In an ACL, individual read/write/execute permissions can be assigned to the owner, the owning group, individual users, or groups. Masks can also be applied that indicate the maximum effective permissions for a class.

For those who require even more flexible access control, SE Linux[15] uses a powerful policy language that can express a wide variety of access control policies for files and filesystem operations. In fact, Linux Security Module (LSM)[14] hooks (see Section 4.1.3 below) exist for most of the security-relevant filesystem operations, which makes it easier to implement custom filesystem-agnostic security models. Authentication and authorization are pretty well covered with a combination of existing filesystem, kernel, and user-space solutions that are part of most GNU/Linux distributions. Many distributions could, however, do a better job of aiding both the administrator and the user in understanding and using all the tools that they have available to them.

Policies that safeguard sensitive data should include timeouts, whereby the user must periodically re-authenticate in order to continue to access the data. In the event that the authorized users neglect to lock down the machine before leaving work for the day, timeouts help to keep the custodial staff from accessing the data when they come in at night to clean the office. As usual, this must be implemented in such a way as to be unobtrusive to the user. If a user finds a security mechanism overly imposing or inconvenient, he will usually disable or circumvent it.

4.1.2 PAM

Pluggable Authentication Modules (PAM)[10] implement authentication-related security policies. PAM offers discretionary access control (DAC)[12]; applications must defer to PAM in order to authenticate a user. If the authenticating PAM function that is called returns an affirmative answer, then the application can use that response to authorize the action, and vice versa. The exact mechanism that the PAM function uses to evaluate the authentication is dependent on the module called.[2]

In the case of filesystem security and encryption, PAM can be employed to obtain and forward keys to a filesystem encryption layer in kernel space. This would allow seamless inte-

[2]This is parameterizable in the configuration files found under */etc/pam.d/*

gration with any key retrieval mechanism that can be coded as a Pluggable Authentication Module.

4.1.3 LSM

Linux Security Modules (LSM) can provide customized security models. One possible use of LSM is to allow decryption of certain files only when a physical device is connected to the machine. This could be, for example, a USB keychain device, a Smartcard, or an RFID device. Some devices of these classes can also be used to house the encryption keys (retrievable via PAM, as previously discussed).

4.1.4 Auditing

The second angle to filesystem integrity is auditing. Auditing should only fill in where authentication and authorization mechanisms fall short. In a utopian world, where security systems are perfect and trusted people always act trustworthily, auditing does not have much of a use. In reality, code that implements security has defects and vulnerabilities. Passwords can be compromised, and authorized people can act in an untrustworthy manner. Auditing can involve keeping a log of all changes made to the attributes of the file or to the file data itself. It can also involve taking snapshots of the attributes and/or contents of the file and comparing the current state of the file with what was recorded in a prior snapshot.

Intrusion detection systems (IDS), such as Tripwire[16], AIDE[17], or Samhain[18], perform auditing functions. As an example, Tripwire periodically scans the contents of the filesystem, checking file attributes, such as the size, the modification time, and the cryptographic hash of each file. If any attributes for the files being checked are found to be altered, Tripwire will report it. This approach can work fairly well in cases where the files are not expected to change very often, as is the case with most system scripts, shared libraries, executables, or configuration files. However, care must be taken to assure that the attacker cannot also modify Tripwire's database when he modifies a system file; the integrity of the IDS system itself must also be assured.

In cases where a file changes often, such as a database file or a spreadsheet file in an active project, we see a need for a more dynamic auditing solution - one which is perhaps more closely integrated with the filesystem itself. In many cases, the simple fact that the file has changed does not imply a security violation. We must also know who made the change. More robust security requirements also demand that we know what parts of the file were changed and when the changes were made. One could even imagine scenarios where the context of the change must also be taken into consideration (i.e., who was logged in, which processes were running, or what network activity was taking place at the time the change was made).

File integrity, particularly in the area of auditing, is perhaps the security aspect of Linux filesystems that could use the most improvement. Most efforts in secure filesystem development have focused on confidentiality more so than integrity, and integrity has been regulated to the domain of userland utilities that must periodically scan the entire filesystem. Sometimes, just knowing that a file has been changed is insufficient. Administrators would like to know exactly how the attacker made the changes and under what circumstances they were made.

Cryptographic hashes are often used. These can detect unauthorized circumvention of the filesystem itself, as long as the attacker forgets (or is unable) to update the hashes when making unauthorized changes to the files. Some auditing solutions, such as the Linux Auditing System (LAuS)[3] that is part of SuSE Linux Enterprise Server, can track system calls that affect the filesystem. Another recent addition to the 2.6 Linux kernel is the Light-weight Auditing Framework written by Rik Faith[28]. These are implemented independently of the filesystem itself, and the level of detail in the records is largely limited to the system call parameters and return codes. It is advisable that you keep your log files on a separate machine than the one being audited, since the attacker could modify the audit logs themselves once he has compromised the machine's security.

4.1.5 Improvements on Integrity

Extended Attributes provide for a convenient way to attach metadata relating to a file to the file itself. On the premise that possession of a secret equates to authentication, every time an authenticated subject makes an authorized write to a file, a hash over the concatenation of that secret to the file contents (keyed hashing; HMAC is one popular standard) can be written as an Extended Attribute on that file. Since this action would be performed on the filesystem level, the user would not have to conscientiously re-run userspace tools to perform such an operation every time he wants to generate an integrity verifier on the file.

This is an expensive operation to perform over large files, and so it would be a good idea to define extent sizes over which keyed hashes are formed, with the Extended Attributes including extent descriptors along with the keyed hashes. That way, a small change in the middle of a large file would only require the keyed hash to be re-generated over the extent in which the change occurs. A keyed hash over the sequential set of the extent hashes would also keep an attacker from swapping around extents undetected.

[3]Note that LAuS is being covered in more detail in the 2004 Ottawa Linux Symposium by Doc Shankar, Emily Ratliff, and Olaf Kirch as part of their presentation regarding CAPP/EAL3+ Certification.

4.2 File Confidentiality

Confidentiality means that only authorized users can read the contents of a file. Sometimes the names of the files themselves or a directory structure can be sensitive. In other cases, the sizes of the files or the modification times can betray more information than one might want to be known. Even the security policies protecting the files can reveal sensitive information. For example, "Only employees of Novell and SuSE can read this file" would imply that Novell and SuSE are collaborating on something, and neither of them may want this fact to be public knowledge as of yet. Many interesting protocols have been developed that can address these sorts of issues; some of them are easier to implement than others.

When approaching the question of confidentiality, we assume that the block device that contains the file is vulnerable to physical compromise. For example, a laptop that contains sensitive material might be lost, or a database server might be stolen in a burglary. In either event, the data on the hard drive must not be readable by an unauthorized individual. If any individual must be authenticated before he is able to access to the data, then the data is protected against unauthorized access.

Surprisingly, many users surrender their own data's confidentiality (and more often than not they do so unwittingly). It has been my personal observation that most people do not fully understand the lack of confidentiality afforded their data when they send it over the Internet. To compound this problem, comprehending and even using most encryption tools takes considerable time and effort on the part of most users. If sensitive files could be *encrypted by default*, only to be decrypted by those authorized at the time of access, then the user would not have to expend so much effort toward protecting the data's confidentiality.

By putting the encryption at the filesystem layer, this model becomes possible without any modifications to the applications or libraries. A policy at that layer can dictate that certain processes, such as the mail client, are to receive the encrypted version any files that are read from disk.

4.2.1 Encryption

File confidentiality is most commonly accomplished through encryption. For performance reasons, secure filesystems use symmetric key cryptography, like AES or Triple-DES, although an asymmetric public/private keypair may be used to encrypt the symmetric key in some key management schemes. This hybrid approach is in common use through SSL and PGP encryption protocols.

One of our proposals to extend Cryptfs is to mirror the techniques used in GnuPG encryption. If the symmetric key that protects the contents of a file is encrypted with the public key of the intended recipient of the file and stored as an Extended Attribute of the file, then that file can be transmitted in multiple ways (e.g., physical device such as removable storage); as long as the Extended Attributes of the file are preserved across filesystem transfers, then the recipient with the corresponding private key has all the information that his Cryptfs layer needs to transparently decrypt the contents of the file.

4.2.2 Key Management

Key management will make or break a cryptographic filesystem.[5] If the key can be easily compromised, then even the strongest cipher will provide weak protection. If your key is accessible in an unencrypted file or in an unprotected region of memory, or if it is ever transmitted over the network in the clear, a rogue user can capture that key and use it later. Most passwords have poor entropy, which means that an attacker can have pretty good success with a brute force attack against the password. Thus the weakest link in the chain for password-based encryption is usually the password itself. The Cryptographic Filesystem (CFS)[22] mandates that the user choose a password with a length of at least 16 characters.[4]

Ideally, the key would be kept in password-encrypted form on a removable device (like a USB keychain drive) that is stored separately from the files that the key is used to encrypt. That way, an attacker would have to both compromise the password and gain physical access to the removable device before he could decrypt your files.

Filesystem encryption is one of the most exciting applications for the Trusted Computing Platform. Given that the attacker has physical access to a machine with a Trusted Platform Module, it is significantly more difficult to compromise the key. By using secret sharing (otherwise known as *key splitting*)[4], the actual key used to decrypt a file on the filesystem can be contained as both the user's key and the machine's key (as contained in the TPM). In order to decrypt the files, an attacker must not

[4]The subject of secure password selection, although an important one, is beyond the scope of this article. Recommended reading on this subject is at `http://www.alw.nih.gov/Security/Docs/passwd.html`.

only compromise the user key, but he must also have access to the machine on which the TPM chip is installed. This "binds" the encrypted files to the machine. This is especially useful for protecting files on removable backup media.

4.2.3 Cryptanalysis

All block ciphers and most stream ciphers are, to various degrees, vulnerable to successful cryptanalysis. If a cipher is used improperly, then it may become even easier to discover the plaintext and/or the key. For example, with certain ciphers operating in certain modes, an attacker could discover information that aids in cryptanalysis by getting the filesystem to re-encrypt an already encrypted block of data. Other times, a cryptanalyst can deduce information about the type of data in the encrypted file when that data has predictable segments of data, like a common header or footer (thus allowing for a known-plaintext attack).

4.2.4 Cipher Modes

A block encryption mode that is resistant to cryptanalysis can involve dependencies among chains of bytes or blocks of data. Cipher-block-chaining (CBC) mode, for example, provides adequate encryption in many circumstances. In CBC mode, a change to one block of data will require that all subsequent blocks of data be re-encrypted. One can see how this would impact performance for large files, as a modification to data near the beginning of the file would require that all subsequent blocks be read, decrypted, re-encrypted, and written out again.

This particular inefficiency can be effectively addressed by defining chaining extents. By limiting regions of the file that encompass chained blocks, it is feasible to decrypt and re-encrypt the smaller segments. For example, if the block size for a cipher is 64 bits (8 bytes) and the block size, which is (we assume) the minimum unit of data that the block device driver can transfer at a time (512 bytes) then one could limit the number of blocks in any extent to 64 blocks. Depending on the plaintext (and other factors), this may be too few to effectively counter cryptanalysis, and so the extent size could be set to a small multiple of the page size without severely impacting overall performance. The optimal extent size largely depends on the access patterns and data patterns for the file in question; we plan on benchmarking against varying extent lengths under varying access patterns.

4.2.5 Key Escrow

The proverbial question, "What if the sysadmin gets hit by a bus?" is one that no organization should ever stop asking. In fact, sometimes no one person should alone have independent access to the sensitive data; multiple passwords may be required before the data is decrypted. Shareholders should demand that no single person in the company have full access to certain valuable data, in order to mitigate the damage to the company that could be done by a single corrupt administrator or executive. Methods for secret sharing can be employed to assure that multiple keys be required for file access, and (m,n)-threshold schemes [4] can ensure that the data is retrievable, even if a certain number of the keys are lost. Secret sharing would be easily implementable as part of any of the existing cryptographic filesystems.

4.3 File Resilience

The loss of a file can be just as devastating as the compromise of a file. There are many well-established solutions to performing backups of your filesystem, but some cryptographic filesystems preclude the ability to efficiently and/or securely use them. Backup tapes tend to be easier to steal than secure computer systems are, and if unencrypted versions of secure files exist on the tapes, that constitutes an often-overlooked vulnerability.

The Linux 2.6 kernel cryptoloop device[5] filesystem is an all-or-nothing approach. Most backup utilities must be given free reign on the unencrypted directory listings in order to perform incremental backups. Most other encrypted filesystems keep sets of encrypted files in directories in the underlying filesystem, which makes incremental backups possible without giving the backup tools access to the unencrypted content of the files.

The backup utilities must, however, maintain backups of the metadata in the directories containing the encrypted files in addition to the files themselves. On the other hand, when the filesystem takes the approach of storing the cryptographic metadata as Extended Attributes for each file, then backup utilities need only worry about copying just the file in question to the backup medium (preserving the Extended Attributes, of course).

4.4 Advantages of FS-Level, EA-Guided Encryption

Most encrypted filesystem solutions either operate on the entire block device or operate on entire directories. There are several advantages to implementing filesystem encryption at the filesystem level and storing encryption metadata in the Extended Attributes of each file:

- Granularity: Keys can be mapped to individual files, rather than entire block devices or entire directories.
- Backup Utilities: Incremental backup tools can correctly operate without having to have access to the decrypted content of the files it is backing up.
- Performance: In most cases, only certain files need to be encrypted. System libraries and executables, in general, do not need to be encrypted. By limiting the actual encryption and decryption to only those files that really need it, system resources will not be taxed as much.
- Transparent Operation: Individual encrypted files can be easily transfered off of the block device without any extra transformation, and others with authorization will be able to decrypt those files. The userspace applications and libraries do not need to be modified and recompiled to support this transparency.

[5] Note that this is deprecated and is in the process of being replaced with the Device Mapper crypto target.

Since all the information necessary to decrypt a file is contained in the Extended Attributes of the file, it is possible for a user on a machine that is not running Cryptfs to use userland utilities to access the contents of the file. This also applies to other security-related operations, like verifying keyed hashes. This addresses compatibility issues with machines that are not running the encrypted filesystem layer.

5 Survey of Linux Encrypted Filesystems

5.1 Encrypted Loopback Filesystems

5.1.1 Loop-aes

The most well-known method of encrypting a filesystem is to use a loopback encrypted filesystem.[6] Loop-aes[20] is part of the 2.6 Linux kernel (`CONFIG_BLK_DEV_CRYPTOLOOP`). It performs encryption at the block device level. With Loop-aes, the administrator can choose whatever cipher he wishes to use with the filesystem. The *mount* package on most popular GNU/Linux distributions contains the *losetup* utility, which can be used to set up the encrypted loopback mount (you can choose whatever cipher that the kernel supports; we use blowfish in this example):

```
root# modprobe cryptoloop
root# modprobe blowfish
root# dd if=/dev/urandom of=encrypted.img \
      bs=4k count=1000
root# losetup -e blowfish /dev/loop0 \
      encrypted.img
root# mkfs.ext3 /dev/loop0
root# mkdir /mnt/unencrypted-view
root# mount /dev/loop0 /mnt/unencrypted-view
```

The loopback encrypted filesystem falls short in the fact that it is an all-or-nothing solution. It is impossible for most standard backup utilities to perform incremental backups on sets of encrypted files without being given access to the unencrypted files. In addition, remote users will need to use IPSec or some other network encryption layer when accessing the files, which must be exported from the unencrypted mount point on the server. Loop-aes is, however, the best performing encrypted filesystem that is freely available and integrated with most GNU/Linux distributions. It is an adequate solution for many who require little more than basic encryption of their entire filesystems.

5.1.2 BestCrypt

BestCrypt[23] is a non-free product that uses a loopback approach, similar to Loop-aes.

[6]Note that Loop-aes is being deprecated, in favor of Device Mapping (DM) Crypt, which also does encryption at the block device layer.

5.1.3 PPDD

PPDD[21] is a block device driver that encrypts and decrypts data as it goes to and comes from another block device. It works very much like Loop-aes; in fact, in the 2.4 kernel, it uses the loopback device, as Loop-aes does. PPDD has not been ported to the 2.6 kernel. Loop-aes takes the same approach, and Loop-aes ships with the 2.6 kernel itself.

5.2 CFS

The Cryptographic Filesystem (CFS)[22] by Matt Blaze is a well established transparent encrypted filesystem, originally written for BSD platforms. CFS is implemented entirely in userspace and operates similarly to NFS. A userspace daemon, cfsd, acts as a pseudo-NFS server, and the kernel makes RPC calls to the daemon. The CFS daemon performs transparent encryption and decryption when writing and reading data. Just as NFS can export a directory from any exportable filesystem, CFS can do the same, while managing the encryption on top of that filesystem.

In the background, CFS stores the metadata necessary to encrypt and decrypt files with the files being encrypted or decrypted on the filesystem. If you were to look at those directories directly, you would see a set of files with encrypted values for filenames, and there would be a handful of metadata files mixed in. When accessed through CFS, those metadata files are hidden, and the files are transparently encrypted and decrypted for the user applications (with the proper credentials) to freely work with the data.

While CFS is capable of acting as a remote NFS server, this is not recommended for many reasons, some of which include performance and security issues with plaintext passwords and unencrypted data being transmitted over the network. You would be better off, from a security perspective (and perhaps also performance, depending on the number of clients), to use a regular NFS server to handle remote mounts of the encrypted directories, with local CFS mounts off of the NFS mounts.

Perhaps the most attractive attribute of CFS is the fact that it does not require any modifications to the standard Linux kernel. The source code for CFS is freely obtainable. It is packaged in the Debian repositories and is also available in RPM form. Using apt, CFS is perhaps the easiest encrypted filesystem for a user to set up and start using:

```
root# apt-get install cfs
user# cmkdir encrypted-data
user# cattach encrypted-data unencrypted-view
```

The user will be prompted for his password at the requisite stages. At this point, anything the user writes to or reads from */crypt/unencrypted-view* will be transparently encrypted to and decrypted from files in *encrypted-data*. Note that any user on the system can make a new encrypted directory and attach it. It is not necessary to initialize and mount an entire block device, as is the case with Loop-aes.

5.3 TCFS

TCFS[24] is a variation on CFS that includes secure integrated remote access and file integrity features. TCFS assumes the client's workstation is trusted, and the server cannot necessarily be trusted. Everything sent to and from the server is encrypted. Encryption and decryption take place on the client side.

Note that this behavior can be mimicked with a CFS mount on top of an NFS mount. However, because TCFS works within the kernel (thus requiring a patch) and does not necessitate two levels of mounting, it is faster than an NFS+CFS combination.

TCFS is no longer an actively maintained project. The last release was made three years ago for the 2.0 kernel.

5.4 Cryptfs

As a proof-of-concept for the FiST stackable filesystem framework, Erez Zadok, et. al. developed Cryptfs[1]. Under Cryptfs, symmetric keys are associated with groups of files within a single directory. The key is generated with a password that is entered at the time that the filesystem is mounted. The Cryptfs mount point provides an unencrypted view of the directory that contains the encrypted files.

The authors of this paper are currently working on extending Cryptfs to provide seamless integration into the user's desktop environment (see Section 6).

5.5 Userspace Encrypted Filesystems

EncFS[25] utilizes the Filesystem in Userspace (FUSE) library and kernel module to implement an encrypted filesystem in userspace. Like CFS, EncFS encrypts on a per-file basis.

CryptoFS[26] is similar to EncFS, except it uses the Linux Userland Filesystem (LUFS) library instead of FUSE.

SSHFS[27], like CryptoFS, uses the LUFS kernel module and userspace daemon. It limits itself to encrypting the files via SFTP as they are transfered over a network; the files stored on disk are unencrypted. From the user perspective, all file accesses take place as though they were being performed on any regular filesystem (opens, read, writes, etc.). SSHFS transfers the files back and forth via SFTP with the file server as these operations occur.

5.6 Reiser4

ReiserFS version 4 (Reiser4)[29], while still in the development stage, features pluggable security modules. There are currently proposed modules for Reiser4 that will perform encryption and auditing.

5.7 Network Filesystem Security

Much research has taken place in the domain of networking filesystem security. CIFS, NFSv4, and other networking filesystems face special challenges in relation to user identification, access control, and data secrecy. The NFSv4 protocol definition in RFC 3010 contains descriptions of security mechanisms in section 3[30].

6 Proposed Extensions to Cryptfs

Our proposal is to place file encryption metadata into the Extended Attributes (EA's) of the file itself. Extended Attributes are a generic interface for attaching metadata to files. The Cryptfs layer will be extended to extract that information and to use the information to direct the encrypting and decrypting of the contents of the file. In the event that the filesystem does not support Extended Attributes, another filesystem layer can provide that functionality. The stackable framework effectively allows Cryptfs to operate on top of *any* filesystem.

The encryption process is very similar to that of GnuPG and other public key cryptography programs that use a hybrid approach to encrypting data. By integrating the process into the filesystem, we can achieve a greater degree of transparency, without requiring any changes to userspace applications or libraries.

Under our proposed design, when a new file is created as an encrypted file, the Cryptfs layer generates a new symmetric key K_s for the encryption of the data that will be written. File creation policy enacted by Cryptfs can be dictated by directory attributes or globally defined behavior. The owner of the file is automatically authorized to access the file, and so the symmetric key is encrypted with the public key of the owner of the file K_u, which was passed into the Cryptfs layer at the time that the user logged in by a Pluggable Authentication Module linked against libcryptfs. The encrypted symmetric key is then added to the Extended Attribute set of the file:

$\{K_s\}K_u$

Suppose that the user at this point wants to grant Alice access to the file. Alice's public key, K_a, is in the user's GnuPG keyring. He can run a utility that selects Alice's key, extracts it from the GnuPG keyring, and passes it to the Cryptfs layer, with instructions to add Alice as an authorized user for the file. The new key list in the Extended Attribute set for the file then contains two copies of the symmetric key, encrypted with different public keys:

$\{K_s\}K_u$
$\{K_s\}K_a$

Note that this is not an access control directive; it is rather a confidentiality enforcement mechanism that extends beyond the local machine's access control. Without either the user's or Alice's private key, no entity will be able to access the decrypted contents of the file. The machine that harbors such keys will enact its own access control over the decrypted file, based on standard UNIX file permissions and/or ACL's.

When that file is copied to a removable media or attached to an email, as long as the Extended Attributes are preserved, Alice will have all the information that she needs in order to retrieve the symmetric key for the file and de-

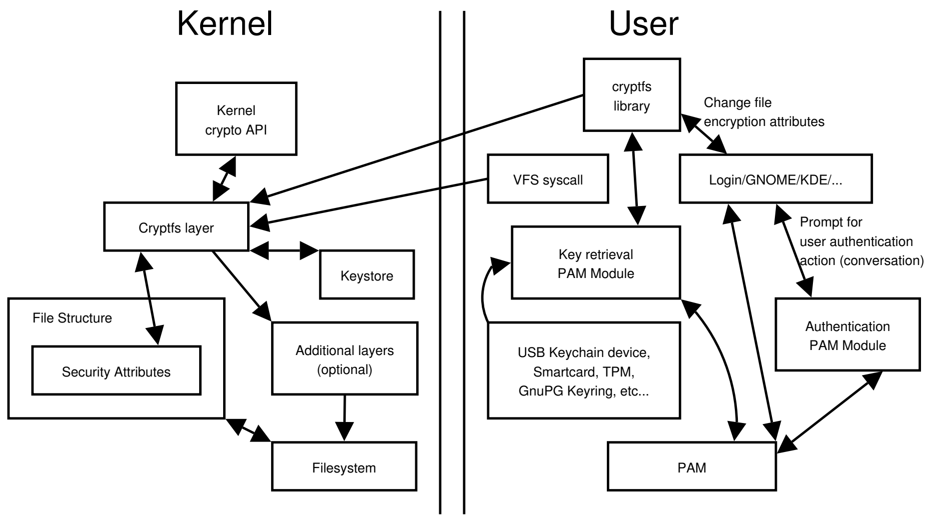

Kernel
User
Kernel crypto API
cryptfs library
Change file encryption attributes
VFS syscall
Login/GNOME/KDE/...
Cryptfs layer
Keystore
Key retrieval PAM Module
Prompt for user authentication action (conversation)
File Structure
Security Attributes
Additional layers (optional)
USB Keychain device, Smartcard, TPM, GnuPG Keyring, etc...
Authentication PAM Module
Filesystem
PAM

Figure 1: Overview of proposed extended Cryptfs architecture

crypt it. If Alice is also running Cryptfs, when she launches an application that accesses the file, the decryption process is entirely transparent to her, since her Cryptfs layer received her private key from PAM at the time that she logged in.

If the user requires the ability to encrypt a file for access by a group of users, then the user can associate sets of public keys with groups and refer to the groups when granting access. The userspace application that links against libcryptfs can then pass in the public keys to Cryptfs for each member of the group and instruct Cryptfs to add the associated key record to the Extended Attributes. Thus no special support for groups is needed within the Cryptfs layer itself.

6.1 Kernel-level Changes

No modifications to the 2.6 kernel itself are necessary to support the stackable Cryptfs layer. The Cryptfs module's logical divisions include a sysfs interface, a keystore, and the VFS operation routines that perform the encryption and the decryption on reads and writes.

By working with a userspace daemon, it would be possible for Cryptfs to export public key cryptographic operations to userspace. In order to avoid the need for such a daemon while using public key cryptography, the kernel cryptographic API must be extended to support it.

6.2 PAM

At login, the user's public and private keys need to find their way into the kernel Cryptfs layer. This can be accomplished by writing a Pluggable Authentication Module, pam_cryptfs.so. This module will link against libcryptfs and will extract keys from the user's GnuPG keystore. The libcryptfs library will use the sysfs interface to pass the user's keys into the Cryptfs layer.

6.3 libcryptfs

The libcryptfs library works with the Cryptfs's sysfs interface. Userspace utilities, such as pam_cryptfs.so, GNOME/KDE, or stand-alone utilities, will link against this library and use it

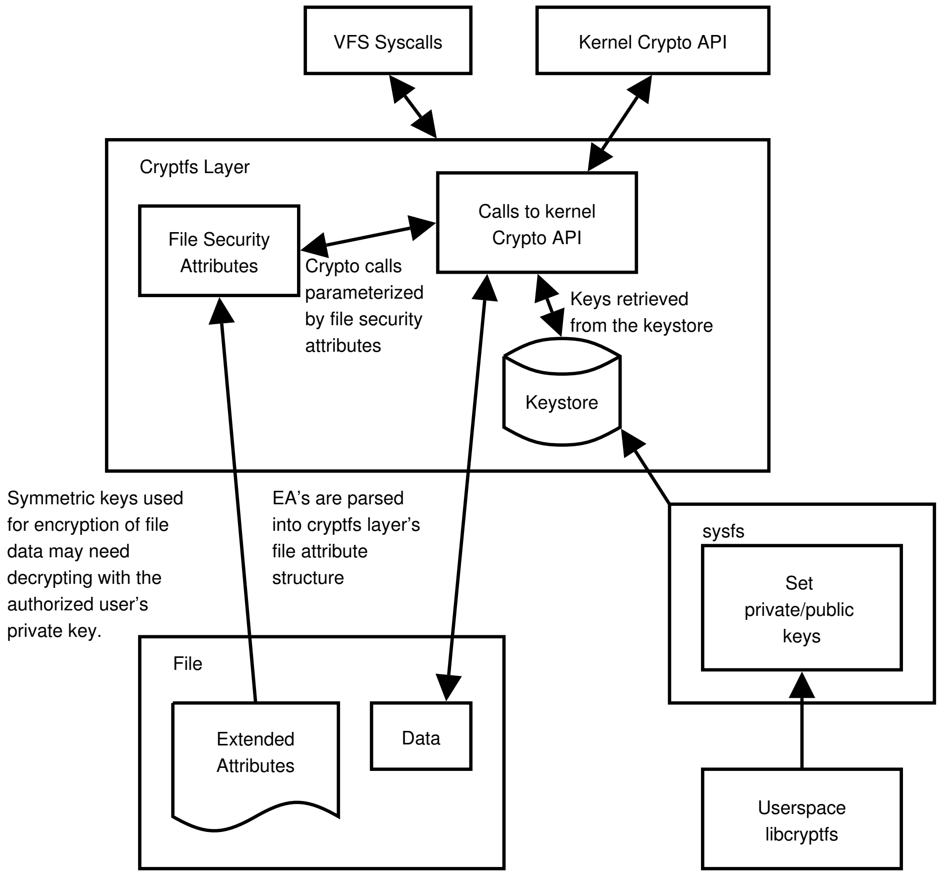

VFS Syscalls
Kernel Crypto API
Cryptfs Layer
File Security Attributes
Calls to kernel Crypto API
Crypto calls parameterized by file security attributes
Keys retrieved from the keystore
Keystore
Symmetric keys used for encryption of file data may need decrypting with the authorized user's private key.
EA's are parsed into cryptfs layer's file attribute structure
sysfs
Set private/public keys
File
Extended Attributes
Data
Userspace libcryptfs

Figure 2: Structure of Cryptfs layer in kernel

to communicate with the kernel Cryptfs layer.

6.4 User Interface

Desktop environments such as GNOME or KDE can link against libcryptfs to provide users with a convenient interface through which to work with the files. For example, by right-clicking on an icon representing the file and selecting "Security", the user will be presented with a window that can be used to control the encryption status of the file. Such options will include whether or not the file is encrypted, which users should be able to encrypt and decrypt the file (identified by their public keys from the user's GnuPG keyring), what cipher is used, what keylength is used, an optional password that encrypts the symmetric key, whether or not to use keyed hashing over extents of the file for integrity, the hash algorithm to use, whether accesses to the file when no key is available should result in an error or in the encrypted blocks being returned (perhaps associated with UID's - good for backup utilities), and other properties that are controlled by the Cryptfs layer.

6.5 Example Walkthrough

When a file's encryption attribute is set, the first thing that the Cryptfs layer will do will be to generate a new symmetric key, which will be used for all encryption and decryption of the file in question. Any data in that file is then immediately encrypted with that key. When using public key-enforced access control, that key will be encrypted with the process owner's private key and stored as an EA of the file. When the process owner wishes to allow others to access the file, he encrypts the symmetric key with the their public keys. From the user's perspective, this can be done by right-clicking on an icon representing the file, selecting "Security→Add Authorized User Key", and having the user specify the authorized user while using PAM to retrieve the public key for that user.

When using password-enforced access control, the symmetric key is instead encrypted using a key generated from a password. The user can then share that password with everyone who he authorized to access the file. In either case (public key-enforced or password-enforced access control), revocation of access to future versions of the file will necessitate regeneration and re-encryption of the symmetric key.

Suppose the encrypted file is then copied to a removable device and delivered to an authorized user. When that user logged into his machine, his private key was retrieved by the key retrieval Pluggable Authentication Module and sent to the Cryptfs keystore. When that user launches any arbitrary application and attempts to access the encrypted file from the removable media, Cryptfs retrieves the encrypted symmetric key correlating with that user's public key, uses the authenticated user's private key to decrypt the symmetric key, associates that symmetric key with the file, and then proceeds to use that symmetric key for reading and writing the file. This is done in an entirely transparent manner from the perspective of the user, and the file maintains its encrypted status on the removable media throughout the entire process. No modification to the application or applications accessing the file are necessary to implement such functionality.

In the case where a file's symmetric key is encrypted with a password, it will be necessary for the user to launch a daemon that listens for password queries from the kernel cryptfs layer. Without such a daemon, the user's initial attempt to access the file will be denied, and the user will have to use a password set utility to send the password to the cryptfs layer in the kernel.

6.6 Other Considerations

Sparse files present a challenge to encrypted filesystems. Under traditional UNIX semantics, when a user seeks more than a block beyond the end of a file to write, then that space is not stored on the block device at all. These missing blocks are known as "holes."

When holes are later read, the kernel simply fills in zeros into the memory without actually reading the zeros from disk (recall that they do not exist on the disk at all; the filesystem "fakes it"). From the point of view of whatever is asking for the data from the filesystem, the section of the file being read appears to be all zeros. This presents a problem when the file is supposed to be encrypted. Without taking sparse files into consideration, the encryption layer will naïvely assume that the zeros being passed to it from the underlying filesystem are actually encrypted data, and it will attempt to decrypt the zeros. Obviously, this will result in something other that zeros being presented above the encryption layer, thus violating UNIX sparse file semantics.

One solution to this problem is to abandon the concept of "holes" altogether at the Cryptfs layer. Whenever we seek past the end of the file and write, we can actually encrypt blocks of zeros and write them out to the underlying filesystem. While this allows Cryptfs to adhere to UNIX semantics, it is much less efficient. One possible solution might be to store a "hole bitmap" as an Extended Attribute of the file. Each bit would correspond with a block of the file; a “1” might indicate that the block is a “hole” and should be zero’d out rather than decrypted, and a “0” might indicate that the block should be normally decrypted.

Our proposed extensions to Cryptfs in the near future do not currently address the issues of directory structure and file size secrecy. We recognize that this type of confidentiality is important to many, and we plan to explore ways to integrate such features into Cryptfs, possibly by employing extra filesystem layers to aid in the process.

Extended Attribute content can also be sensitive. Technically, only enough information to retrieve the symmetric decryption key need be accessible by authorized individuals; all other attributes can be encrypted with that key, just as the contents of the file are encrypted.

Processes that are not authorized to access the decrypted content will either be denied access to the file or will receive the encrypted content, depending on how the Cryptfs layer is parameterized. This behavior permits incremental backup utilities to function properly, without requiring access to the unencrypted content of the files they are backing up.

At some point, we would like to include file integrity information in the Extended Attributes. As previously mentioned, this can be accomplished via sets of keyed hashes over extents within the file:

$$\begin{aligned}
&H_0 = H\{O_0, D_0, K_s\} \\
&H_1 = H\{O_1, D_1, K_s\} \\
&\ldots \\
&H_n = H\{O_n, D_n, K_s\} \\
&H_f = H\{H_0, H_1, \ldots, H_n, n, s, K_s\}
\end{aligned}$$

where n is the number of extents in the file, s is the extent size (also contained as another EA), O_i is the offset number i within the file, D_i is the data from offset O_i to $O_i + s$, K_s is the key that one must possess in order to make authorized changes to the file, and H_f is the hash of the hashes, the number of extents, the extent size, and the secret key, to help detect when an attacker swaps around extents or alters the extent size.

Keyed hashes prove that whoever modified the data had access to the shared secret, which is, in this case, the symmetric key. Digital signatures can also be incorporated into Cryptfs. Executables downloaded over the Internet can often be of questionable origin or integrity. If you trust the person who signed the executable, then you can have a higher degree of certainty that the executable is safe to run if the digital signature is verifiable. The verification of the digital signature can be dynamically performed at the time of execution.

As previously mentioned, in addition to the extensions to the Cryptfs stackable layer, this effort is requiring the development of a cryptfs library, a set of PAM modules, hooks into GNOME and KDE, and some utilities for managing file encryption. Applications that copy files with Extended Attributes must take steps to make sure that they preserve the Extended Attributes.[7]

7 Conclusion

Linux currently has a comprehensive framework for managing filesystem security. Standard file security attributes, process credentials, ACL, PAM, LSM, Device Mapping (DM) Crypt, and other features together provide good security in a contained environment. To extend access control enforcement over individual files beyond the local environment, you must use encryption in a way that can be easily

[7]See `http://www.suse.de/~agruen/ea-acl-copy/`

applied to individual files. The currently employed processes of encrypting and decrypting files, however, is inconvenient and often obstructive.

By integrating the encryption and the decryption of the individual files into the filesystem itself, associating encryption metadata with the individual files, we can extend Linux security to provide seamless encryption-enforced access control and integrity auditing.

8 Recognitions

We would like to express our appreciation for the contributions and input on the part of all those who have laid the groundwork for an effort toward transparent filesystem encryption. This includes contributors to FiST and Cryptfs, GnuPG, PAM, and many others from which we are basing our development efforts, as well as several members of the kernel development community.

9 Legal Statement

References

[1] E. Zadok, L. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. *Technical Report CUCS-021-98, Computer Science Department*, Columbia University, 1998.

[2] J.S. Heidemann and G.J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[3] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. *Proceedings of the Annual USENIX Technical Conference*, pp. 55–70, San Diego, June 2000.

[4] S.C. Kothari, Generalized Linear Threshold Scheme, *Advances in Cryptology: Proceedings of CRYPTO 84*, Springer-Verlag, 1985, pp. 231–241.

[5] Matt Blaze. "Key Management in an Encrypting File System," Proc. *Summer '94 USENIX Tech. Conference*, Boston, MA, June 1994.

[6] For more information on Extended Attributes (EA's) and Access Control Lists (ACL's), see `http://acl.bestbits.at/` or `http://www.suse.de/~agruen/acl/chapter/fs_acl-en.pdf`

[7] For more information on GnuPG, see `http://www.gnupg.org/`

[8] For more information on OpenSSL, see `http://www.openssl.org/`

[9] For more information on IBM Lotus Notes, see `http://www-306.ibm.com/software/lotus/`. Information on Notes security can be obtained from `http://www-10.lotus.com/ldd/today.nsf/f01245ebfc115aaf8525661a006b86b9/232e604b847d2cad88256ab90074e298?OpenDocument`

[10] For more information on Pluggable Authentication Modules (PAM), see `http://www.kernel.org/pub/linux/libs/pam/`

[11] For more information on Mandatory Access Control (MAC), see `http://csrc.nist.gov/publications/nistpubs/800-7/node35.html`

[12] For more information on Discretionary Access Control (DAC), see `http://csrc.nist.gov/publications/nistpubs/800-7/node25.html`

[13] For more information on the Trusted Computing Platform Alliance (TCPA), see `http://www.trustedcomputing.org/home`

[14] For more information on Linux Security Modules (LSM's), see `http://lsm.immunix.org/`

[15] For more information on Security-Enhanced Linux (SE Linux), see `http://www.nsa.gov/selinux/index.cfm`

[16] For more information on Tripwire, see `http://www.tripwire.org/`

[17] For more information on AIDE, see `http://www.cs.tut.fi/~rammer/aide.html`

[18] For more information on Samhain, see `http://la-samhna.de/samhain/`

[19] For more information on Logcrypt, see `http://www.lunkwill.org/logcrypt/`

[20] For more information on Loop-aes, see `http://sourceforge.net/projects/loop-aes/`

[21] For more information on PPDD, see `http://linux01.gwdg.de/~alatham/ppdd.html`

[22] For more information on CFS, see `http://sourceforge.net/projects/cfsnfs/`

[23] For more information on BestCrypt, see `http://www.jetico.com/index.htm#/products.htm`

[24] For more information on TCFS, see `http://www.tcfs.it/`

[25] For more information on EncFS, see `http://arg0.net/users/vgough/encfs.html`

[26] For more information on CryptoFS, see `http://reboot.animeirc.de/cryptofs/`

[27] For more information on SSHFS, see `http://lufs.sourceforge.net/lufs/fs.html`

[28] For more information on the Light-weight Auditing Framework, see `http://lwn.net/Articles/79326/`

[29] For more information on Reiser4, see `http://www.namesys.com/v4/v4.html`

[30] NFSv4 RFC 3010 can be obtained from `http://www.ietf.org/rfc/rfc3010.txt`

Hotplug Memory and the Linux VM

Dave Hansen, Mike Kravetz, with Brad Christiansen
IBM Linux Technology Center
haveblue@us.ibm.com, kravetz@us.ibm.com, bradc1@us.ibm.com

Matt Tolentino
Intel
matthew.e.tolentino@intel.com

Abstract

This paper will describe the changes needed to the Linux memory management system to cope with adding or removing RAM from a running system. In addition to support for physically adding or removing DIMMs, there is an ever-increasing number of virtualized environments such as UML or the IBM pSeries™ Hypervisor which can transition RAM between virtual system images, based on need. This paper will describe techniques common to all supported platforms, as well as challenges for specific architectures.

1 Introduction

As Free Software Operating Systems continue to expand their scope of use, so do the demands placed upon them. One area of continuing growth for Linux is the adaptation to incessantly changing hardware configurations at runtime. While initially confined to commonly removed devices such as keyboards, digital cameras or hard disks, Linux has recently begun to grow to include the capability to hot-plug integral system components. This paper describes the changes necessary to enable Linux to adapt to dynamic changes in one of the most critical system resource—system RAM.

2 Motivation

The underlying reason for wanting to change the amount of RAM is very simple: availability. The systems that support memory hot-plug operations are designed to fulfill mission critical roles; significant enough that the cost of a reboot cycle for the sole purpose of adding or replacing system RAM is simply too expensive. For example, some large ppc64 machines have been reported to take well over thirty minutes for a simple reboot. Therefore, the downtime necessary for an upgrade may compromise the five nine uptime requirement critical to high-end system customers [1].

However, memory hotplug is not just important for big-iron. The availability of high speed, commodity hardware has prompted a resurgence of research into virtual machine monitors—layers of software such as Xen [2], VMWare [3], and conceptually even User Mode Linux that allow for multiple operating system instances to be run in isolated, virtual domains. As computing hardware density has increased, so has the possibility of splitting up that computing power into more manageable pieces. The capability for an operating system to expand or contract the range of physical memory resources available presents the possibility for virtual machine implementations to balance memory requirements and improve the management of memory availability between domains[1]. This author currently leases a small User Mode Linux partition for small Internet tasks such as DNS and low-traffic web serving. Similar configurations with an approximately 100 MHz processor and 64 MB of RAM are not uncommon. Imagine, in the case of an accidental Slashdotting, how useful radically growing such a machine could be.

3 Linux's Hotplug Shortcomings

Before being able to handle the full wrath of Slashdot. we have to consider Linux's current design. Linux only has two data structures that absolutely limit the amount of RAM that Linux can handle: the page allocator bitmaps, and `mem_map[]` (on contiguous memory systems). The page allocator bitmaps are very simple in concept, have a bit set one way when a page is available, and the opposite when it has been allocated. Since there needs to be one bit available for each page, it obviously has to scale with the size of the system's total RAM. The bitmap memory consumption is approximately 1 bit of memory for each page of system RAM.

4 Resizing `mem_map[]`

The `mem_map[]` structure is a bit more complicated. Conceptually, it is an array, with one `struct page` for each physical page which the system contains. These structures contain bookkeeping information such as flags indicating page usage and locking structures. The complexity with the `struct page`s is associated when their size. They have a size of 40 bytes each on i386 (in the 2.6.5 kernel). On a system with 4096 byte hardware pages, this implies that about 1% of the total system memory will be consumed by `struct page`s alone. This use of 1% of the system memory is not a problem in and of itself. But, it does other problems.

The Linux page allocator has a limitation on the maximum amounts of memory that it can allocate to a single request. On i386, this is 4MB, while on ppc64, it is 16MB. It is easy to calculate that anything larger than a 4GB i386 system will be unable to allocate its `mem_map[]` with the normal page allocator. Normally, this problem with `mem_map` is avoided by using a boot-time allocator which does not have the same restrictions as the allocator used at runtime. However, memory hotplug requires the ability to grow the amount of `mem_map[]` used at runtime. It is not feasible to use the same approach as the page allocator bitmaps because, in contrast, they are kept to small-enough sizes to not impinge on the maximum size allocation limits.

4.1 `mem_map[]` preallocation

A very simple way around the runtime allocator limitations might be to allocate sufficient memory form `mem_map[]` at boot-time to account for any amount of RAM that could possibly be added to the system. But, this approach quickly breaks down in at least one important case. The `mem_map[]` must be allocated in low memory, an area on i386 which is approximately 896MB in total size. This is very important memory which is commonly exhausted [4],[5],[6]. Consider an 8GB system which could be expanded to 64GB in the future. Its normal `mem_map[]` use would be around 84MB, an acceptable 10% use of low memory. However, had `mem_map[]` been preallocated to handle a total capacity of 64GB of system memory, it would use an astound-

[1]err, I could write a lot about this, so I won't go any further

ing 71% of low memory, giving any 8GB system all of the low memory problems associated with much larger systems.

Preallocation also has the disadvantage of imposing limitations possibly making the user decide how large they expect the system to be, either when the kernel is compiled, or when it is booted. Perhaps the administrator of the above 8GB machine knows that it will never get any larger than 16GB. Does that make the low memory usage more acceptable? It would likely solve the immediate problem, however, such limitations and user intervention are becoming increasingly unacceptable to Linux vendors, as they drastically increase possible user configurations, and support costs along with it.

4.2 Breaking `mem_map[]` up

Instead of preallocation, another solution is to break up `mem_map[]`. Instead of needing massive amounts of memory, smaller ones could be used to piece together `mem_map[]` from more manageable allocations Interestingly, there is already precedent in the Linux kernel for such an approach. The discontiguous memory support code tries to solve a different problem (large holes in the physical address space), but a similar solution was needed. In fact, there has been code released to use the current discontigmem support in Linux to implement memory hotplug. But, this has several disadvantages. Most importantly, it requires hijacking the NUMA code for use with memory hotplug. This would exclude the use of NUMA and memory hotplug on the same system, which is likely an unacceptable compromise due to the vast performance benefits demonstrated from using the Linux NUMA code for its intended use [6].

Using the NUMA code for memory hotplug is a very tempting proposition because in addition to splitting up `mem_map[]` the NUMA support also handles discontiguous memory. Discontiguous memory simply means that the system does not lay out all of its physical memory in a single block, rather there are holes. Handling these holes with memory hotplug is very important, otherwise the only memory that could be added or removed would be on the end.

Although an approch similar to this “node hotplug” approach will be needed when adding or removing entire NUMA nodes, using it on a regular SMP hotplug system could be disastrous. Each discontiguous area is represented by several data structures but each has at least one `structzone`. This structure is the basic unit which Linux uses to pool memory. When the amounts of memory reach certain low levels, Linux will respond by trying to free or swap memory. Artificially creating too many zones causes these events to be triggered much too early, degrading system performance and under-utilizing available RAM.

5 CONFIG_NONLINEAR

The solution to both the `mem_map[]` and discontiguous memory problems comes in a single package: nonlinear memory. First implemented by Daniel Phillips in April of 2002 as an alternative to discontiguous memory, nonlinear solves a similar set of problems.

Laying out `mem_map[]` as an array has several advantages. One of the most important is the ability to quickly determine the physical address of any arbitrary `struct page`. Since `mem_map[N]` represents the Nth page of physical memory, the physical address of the memory represented by that `struct page` can be determined by simple pointer arithmetic:

Once `mem_map[]` is broken up these simple

```
physical_address = (&mem_map[N] - &mem_map[0]) * sizeof(struct page)

struct page N = mem_map[(physical_address / sizeof(struct page)]
```

Figure 1: Physical Address Calculations

calculations are no longer possible, thus another approach is required. The nonlinear approach is to use a set of two lookup tables, each one complementing the above operations: one for converting `struct page` to physical addresses, the other for doing the opposite. While it would be possible to have a table with an entry for every single page, that approach wastes far too much memory. As a result, nonlinear handles pages in uniformly sized sections, each of which has its own `mem_map[]` and an associated physical address range. Linux has some interesting conventions about how addresses are represented, and this has serious implications for how the nonlinear code functions.

5.1 Physical Address Representations

There are, in fact, at least three different ways to represent a physical address in Linux: a physical address, a `struct page`, and a page frame number (pfn). A pfn is traditionally just the physical address divided by the size of a physical page (the *N* in the above in Figure 1). Many parts of the kernel prefer to use a pfn as opposed to a `struct page` pointer to keep track of pages because pfn's are easier to work with, being conceptually just array indexes. The page allocator bitmaps discussed above are just such a part of the kernel. To allocate or free a page, the page allocator toggles a bit at an index in one of the bitmaps. That index is based on a pfn, not a `struct page` or a physical address.

Being so easily transposed, that decision does not seem horribly important. But it does cause a serious problem for memory hotplug. Consider a system with 100 1GB DIMM slots that support hotplug. When the system is first booted, only one of these DIMM slots is populated. Later on, the owner decides to hotplug another DIMM, but puts it in slot 100 instead of slot 2. Now, nonlinear has a bit of a problem: the new DIMM happens to appear at a physical address 100 times higher address than the first DIMM. The `mem_map[]` for the new DIMM is split up properly, but the allocator bitmap's length is directly tied to the pfn, and thus the physical address of the memory.

Having already stated that the allocator bitmap stays at manageable sizes, this still does not seem like much of an issue. However, the physical address of that new memory *could* have an even greater range than 100 GB; it has the capability to have many, many terabytes of range, based on the hardware. Keeping allocator bitmaps for terabytes of memory could conceivably consume all system memory on a small machine, which is quite unacceptable. Nonlinear offers a solution to this by introducing a new way to represent a physical address: a fourth addressing scheme. With three addressing schemes already existing, a fourth seems almost comical, until its small scope is considered. The new scheme is isolated to use inside of a small set of core allocator functions a single place in the memory hotplug code itself. A simple lookup table converts these new "linear" pfns into the more familiar physical pfns.

5.2 Issues with `CONFIG_NONLINEAR`

Although it greatly simplifies several issues, nonlinear is not without its problems. Firstly, it does require the consultation of a small number of lookup tables during critical sections of code. Random access of these tables is likely to cause cache overhead. The more finely grained the units of hotplug, the larger these tables will grow, and the worse the cache effects.

Another concern arises with the size of the nonlinear tables themselves. While they allow pfns and `mem_map[]` to have nonlinear relationships, the nonlinear structures themselves remain normal, everyday, linear arrays. If hardware is encountered with sufficiently small hotplug units, and sufficiently large ranges of physical addresses, an alternate scheme to the arrays may be required. However, it is the authors' desire to keep the implementation simple, until such a need is actually demonstrated.

6 Memory Removal

While memory addition is a relatively black-and-white problem, memory removal has many more shades of gray. There are many different ways to use memory, and each of them has specific challenges for *un*using it. We will first discuss the kinds of memory that Linux has which are relevant to memory removal, along with strategies to go about unusing them.

6.1 "Easy" User Memory

Unusing memory is a matter of either moving data or simply throwing it away. The easiest, most straightforward kind of memory to remove is that whose contents can just be discarded. The two most common manifestations of this are clean page cache pages and swapped pages. Page cache pages are either dirty (containing information which has not been written to disk) or clean pages, which are simply a copy of something that *is* present on the disk. Memory removal logic that encounters a clean page cache page is free to have it discarded, just as the low memory reclaim code does today. The same is true of swapped pages; a page of RAM which has been written to disk is safe to discard. (Note: there is usually a brief period between when a page is written to disk, and when it is actually removed from memory.) Any page that *can* be swapped is also an easy candidate for memory removal, because it can easily be turned into a swapped page with existing code.

6.2 Swappable User Memory

Another type of memory which is very similar to the two types above is something which is only used by user programs, but is for some reason not a candidate for swapping. This at least includes pages which have been `mlock()`'d (which is a system call to prevent swapping). Instead of discarding these pages out of RAM, they must instead be moved. The algorithm to accomplish this should be very similar to the algorithm for a complete page swapping: freeze writes to the page, move the page's contents to another place in memory, change all references to the page, and re-enable writing. Notice that this is the same process as a complete swap cycle except that the writes to the disk are removed.

6.3 Kernel Memory

Now comes the hard part. Up until now, we have discussed memory which is being used by user programs. There is also memory that Linux sets aside for its own use and this comes in many more varieties than that used by user programs. The techniques for dealing with this memory are largely still theoretical, and do not have existing implementations.

Remember how the Linux page allocator can only keep track of pages in powers of two? The Linux slab cache was designed to make up for that [6], [7]. It has the ability to take those powers of two pages, and chop them up into smaller pieces. There are some fixed-size groups for common allocations like 1024, 1532, or 8192 bytes, but there are also caches for certain kinds of data structures. Some of these caches have the ability to attempt to shrink themselves when the system needs some memory back, but even that is relatively worthless for memory hotplug.

6.4 Removing Slab Cache Pages

The problem is that the slab cache's shrinking mechanism does not concentrate on shrinking any particular memory, it just concentrates on shrinking, period. Plus, there's currently no mechanism to tell *which* slab a particular page belongs to. It could just as easily be a simply discarded dcache entry as it could be a completely immovable entry like a `pte_chain`. Linux will need mechanisms to allow the slab cache shrinking to be much more surgical.

However, there will always be slab cache memory which is not covered by any of the shrinking code, like for generic `kmalloc()` allocations. The slab cache could also make efforts to keep these "mystery" allocations away from those for which it knows how to handle.

While the record-keeping for some slab-cache pages is sparse, there is memory with even more mysterious origins. Some is allocated early in the boot process, while other uses pull pages directly out of the allocator never to be seen again. If hot-removal of these areas is required, then a different approach must be employed: direct replacement. Instead of simply reducing the usage of an area of memory until it is unused, a one-to-one replacement of this memory is required. With the judicious use of page tables, the best that can be done is to preserve the virtual address of these areas. While this is acceptable for most use, it is not without its pitfalls.

6.5 Removing DMA Memory

One unacceptable place to change the physical address of some data is for a device's DMA buffer. Modern disk controllers and network devices can transfer their data directly into the system's memory without the CPU's direct involvement. However, since the CPU is not involved, the devices lack access to the CPU's virtual memory architecture. For this reason, all DMA-capable devices' transfers are based on the physical address of the memory to which they are transferring. Every user of DMA in Linux will either need to be guaranteed to not be affected by memory replacement, or to be notified of such a replacement so that it can take corrective action. It should be noted, however, that the virtualization layer on ppc64 can properly handle this remapping in its IOMMU. Other architectures with IOMMUs should be able to employ similar techniques.

6.6 Removal and the Page Allocator

The Linux page allocator works by keeping lists of groups of pages in sizes that are powers of two times the size of a page. It keeps a list of groups that are available for each power of two. However, when a request for a page is made, the only real information provided is for the *size* required, there is no component for specifically specifying which particular memory is required.

The first thing to consider before removing memory is to make sure that no other part of the system is using that piece of memory. Thankfully, that's exactly what a normal allocation does: make sure that it is alone in its use of the page. So, making the page allocator support memory removal will simply involve walking the same lists that store the page groups. But, instead of simply taking the first available pages, it will be more finicky, only "allocating" pages that are among those about to be removed. In addition, the allocator should have checks in the `free_pages()` path to look for pages which were selected for removal.

1. Inform allocator to catch any pages in the area being removed.
2. Go into allocator, and remove any pages in that area.
3. Trigger page reclaim mechanisms to trigger `free()`s, and hopefully unuse all target pages.
4. If not complete, goto 3.

6.7 Page Groupings

As described above, the page allocator is the basis for all memory allocations. However, when it comes time to remove memory a fixed size block of memory is what is removed. These blocks correspond to the sections defined in the the implementation of nonlinear memory. When removing a section of memory, the code performing the remove operation will first try to essentially allocate all the pages in the section. To remove the section, all pages within the section must be made free of use by some mechanism as described above. However, it should be noted that some pages will not be able to be made available for removal. For example, pages in use for kernel allocations, DMA or via the slab-cache. Since the page allocator makes no attempt to group pages based on usage, it is possible in a worst case situation that every section contains one in-use page that can not be removed. Ideally, we would like to group pages based on their usage to allow the maximum number of sections to be removed.

Currently, the definition of zones provides some level of grouping on specific architectures. For example, on i386, three zones are defined: DMA, NORMAL and HIGHMEM. With such definitions, one would expect most non-removable pages to be allocated out of the DMA and NORMAL zones. In addition, one would expect most HIGHMEM allocations to be associated with userspace pages and thus removable. Of course, when the page allocator is under memory pressure it is possible that zone preferences will be ignored and allocations may come from an alternate zone. It should also be noted that on some architectures, such as ppc64, only one zone (DMA) is defined. Hence, zones can not provide grouping of pages on every architecture. It appears that zones do provide some level of page grouping, but possibly not sufficient for memory hotplug.

Ideally, we would like to experiment with teaching the page allocator about the use of pages it is handing out. A simple thought would be to introduce the concept of sections to the allocator. Allocations of a specific type are directed to a section that is primarily used for allocations of that same type. For example, when allocations for use within the kernel are needed the allocator will attempt to allocate the page from a section that contains other internal kernel allocations. If no such pages can be found, then a new section is marked for internal kernel allocations. In this way pages which can not be easily freed are grouped together rather than spread throughout the system. In this way the page allocator's use of sections would be analogous to the slab caches use of pages.

7 Conclusion

The prevalence of hotplug-capable Linux systems is only expanding. Support for these systems will make Linux more flexible and will make additional capabilities available to other parts of the system.

Legal Statement

References

[1] *Five Nine at the IP Edge* `http://www.iec.org/online/tutorials/five-nines`

[2] Barham, Paul, et al. *Xen and the Art of Virtualization* Proceedings of the ACM Symposium on Operating System Principles (SOSP), October 2003.

[3] Waldspurger, Carl *Memory Resource Management in VMware ESX Server* Proceedings of the USENIX Association Symposium on Operating System Design and Implementation, 2002. pp 181–194.

[4] Dobson, Matthew and Gaughen, Patricia and Hohnbaum, Michael. *Linux Support for NUMA Hardware* Proceedings of the Ottawa Linux Symposium. July 2003. pp 181–196.

[5] Gorman, Mel *Understanding the Linux Virtual Memory Manager* Prentice Hall, NJ. 2004.

[6] Martin Bligh and Dave Hansen *Linux Memory Management on Larger Machines* Proceeedings of the Ottawa Linux Symposium 2003. pp 53–88.

[7] Bonwick, Jeff *The Slab Allocator: An Object-Caching Kernel Memory Allocator* Proceedings of USENIX Summer 1994 Technical Conference `http://www.usenix.org/publications/library/proceedings/bos94/bonwick.html`