Linux Kernel Debugging on Google-sized clusters

Martin Bligh
Google
mbligh@mbligh.org

Mathieu Desnoyers
École Polytechnique de Montréal
mathieu.desnoyers@polymtl.ca

Rebecca Schultz
Google
rschultz@google.com

Abstract

This paper will discuss the difficulties and methods involved in debugging the Linux kernel on huge clusters. Intermittent errors that occur once every few years are hard to debug and become a real problem when running across thousands of machines simultaneously. The more we scale clusters, the more reliability becomes critical. Many of the normal debugging luxuries like a serial console or physical access are unavailable. Instead, we need a new strategy for addressing thorny intermittent race conditions. This paper presents the case for a new set of tools that are critical to solve these problems and also very useful in a broader context. It then presents the design for one such tool created from a hybrid of a Google internal tool and the open source LTTng project. Real world case studies are included.

1 Introduction

Well established techniques exist for debugging most Linux kernel problems; instrumentation is added, the error is reproduced, and this cycle is repeated until the problem can be identified and fixed. Good access to the machine via tools such as hardware debuggers (ITPs), VGA and serial consoles simplify this process significantly, reducing the number of iterations required. These techniques work well for problems that can be reproduced quickly and produce a clear error such as an oops or kernel panic. However, there are some types of problems that cannot be properly debugged in this fashion as they are:

- Not easily reproducible on demand;
- Only reproducible in a live production environment;
- Occur infrequently, particularly if they occur infrequently on a single machine, but often enough across a thousand-machine cluster to be significant;
- Only reproducible on unique hardware; or
- Performance problems, that don't produce any error condition.

These problems present specific design challenges; they require a method for extracting debugging information from a running system that does not impact performance, and that allows a developer to drill down on the state of the system leading up to an error, without overloading them with inseparable data. Specifically, problems that only appear in a full-scale production environment require a tool that won't affect the performance of systems running a production workload. Also, bugs which occur infrequently may require instrumentation of a significant number of systems in order to catch the bug in a reasonable time-frame. Additionally, for problems that take a long time to reproduce, continuously collecting and parsing debug data to find relevant information may be impossible, so the system must have a way to prune the collected data.

This paper describes a low-overhead, but powerful, kernel tracing system designed to assist in debugging this class of problems. This system is lightweight enough to run on production systems all the time, and allows for an arbitrary event to trigger trace collection when the bug occurs. It is capable of extracting only the information leading up to the bug, provides a good starting point for analysis, and it provides a framework for easily adding more instrumentation as the bug is tracked. Typically the approach is broken down into the following stages:

1. Identify the problem – for an error condition, this is simple; however, characterization may be more difficult for a performance issue.
2. Create a trigger that will fire when the problem occurs – it could be the error condition itself, or a timer that expires.

  - Use the trigger to dump a buffer containing the trace information leading up to the error.
  - Log the trigger event to the trace for use as a starting point for analysis.

3. Dump information about the succession of events leading to the problem.

4. Analyze results.

In addition to the design and implementation of our tracing tool, we will also present several case studies illustrating the types of errors described above in which our tracing system proved an invaluable resource.

After the bug is identified and fixed, tracing is also extremely useful to demonstrate the problem to other people. This is particularly important in an open source environment, where a loosely coupled team of developers must work together without full access to each other's machines.

2 Related Work

Before being used widely in such large-scale contexts, kernel tracers have been the subject of a lot of work in the past. Besides each and every kernel programmer writing his or her own ad-hoc tracer, a number of formalized projects have presented tracing systems that cover some aspect of kernel tracing.

Going through the timeline of such systems, we start with the Linux Trace Toolkit [6] which aimed primarily at offering a kernel tracing infrastructure to trace a static, fixed set of important kernel-user events useful to understand interactions between kernel and user-space. It also provided the ability to trace custom events. User-space tracing was done through device write. Its high-speed kernel-to-user-space buffering system for extraction of the trace data led to the development of RelayFS [3], now known as Relay, and part of the Linux kernel.

The K42 [5] project, at IBM Research, included a kernel and user-space tracer. Both kernel and user-space applications write trace information in a shared memory segment using a lockless scheme. This has been ported to LTT and inspired the buffering mechanism of LTTng [7], which will be described in this paper.

The SystemTAP[4] project has mainly been focused on providing tracing capabilities to enterprise-level users for diagnosing problems on production systems. It uses the kprobes mechanism to provide dynamic connection of probe handlers at particular instrumentation sites by insertion of breakpoints in the running kernel. SystemTAP defines its own probe language that offers the security guarantee that a programmer's probes won't have side-effects on the system.

Ingo Molnar's IRQ latency tracer, Jens Axboe's blktrace, and Rick Lindsley's schedstats are examples of in-kernel single-purpose tracers which have been added to the mainline kernel. They provide useful information about the system's latency, block I/O, and scheduler decisions.

It must be noted that tracers have existed in proprietary real-time operating systems for years—for example, take the WindRiver Tornado (now replaced by LTTng in their Linux products). Irix has had an in-kernel tracer for a long time, and Sun provides Dtrace[1], an open source tracer for Solaris.

3 Why do we need a tracing tool?

Once the cause of a bug has been identified, fixing it is generally trivial. The difficulty lies in making the connection between an error conveyed to the user—an oops, panic, application error—and the source. In a complex, multi-threaded system such as the Linux kernel, which is both reentrant and preemptive, understanding the paths taken through kernel code can be difficult, especially where the problem is intermittent (such as a race condition). These issues sometimes require powerful information gathering and visualization tools to comprehend.

Existing solutions, such as statistical profiling tools like oprofile, can go some way to presenting an overall view of a system's state and are helpful for a wide class of problems. However, they don't work well for all situations. For example, identifying a race condition requires capturing the precise sequence of events that occurred; the tiny details of ordering are what is needed to identify the problem, not a broad overview. In these situations, a tracing tool is critical. For performance issues, tools like OProfile are useful for identifying hot functions, but don't provide much insight into intermittent latency problems, such as some fraction of a query taking 100 times as long to complete for no apparent reason.

Often the most valuable information for identifying these problems is in the state of the system preceding the event. Collecting that information requires continuous logging and necessitates preserving information about the system for at least some previous section of time.

In addition, we need a system that can capture failures at the earliest possible moment; if a problem takes a week to reproduce, and 10 iterations are required to collect enough information to fix it, the debugging process quickly becomes intractable. The ability to instrument a wide spectrum of the system ahead of time, and provide meaningful data the first time the problem appears, is extremely useful. Having a system that can be deployed in a production environment is also invaluable. Some problems only appear when you run your application in a full cluster deployment; re-creating them in a sandbox is impossible.

Most bugs seem obvious in retrospect, after the cause is understood; however, when a problem first appears, getting a general feel for the source of the problem is essential. Looking at the case studies below, the reader may be tempted to say "you could have detected that using existing tool X;" however, that is done with the benefit of hindsight. It is important to recognize that in some cases, the bug behavior provides no information about what subsystem is causing the problem or even what tools would help you narrow it down. Having a single, holistic tracing tool enables us to debug a wide variety of problems quickly. Even if not all necessary sites are instrumented prior to the fact, it quickly identifies the general area the problem lies in, allowing a developer to quickly and simply add instrumentation on top of the existing infrastructure.

If there is no clear failure event in the trace (e.g. an OOM kill condition, or watchdog trigger), but a more general performance issue instead, it is important to be able to visualize the data in some fashion to see how performance changes around the time the problem is observed. By observing the elapsed time for a series of calls (such as a system call), it is often easy to build an expected average time for an event making it possible to identify outliers. Once a problem is narrowed down to a particular region of the trace data, that part of the trace can be more closely dissected and broken down into its constituent parts, revealing which part of the call is slowing it down.

Since the problem does not necessarily present itself at each execution of the system call, logging data (local variables, static variables) when the system call executes can provide more information about the particularities of an unsuccessful or slow system call compared to the normal behavior. Even this may not be sufficient—if the problem arises from the interaction of other CPUs or interrupt handlers with the system call, one has to look at the trace of the complete system. Only then can we have an idea of where to add further instrumentation to identify the code responsible for a race condition.

4 Case Studies

4.1 Occasional poor latency for I/O write requests

Problem Summary: The master node of a large-scale distributed system was reporting occasional timeout errors on writes to disk, causing a cluster fail-over event. No visible errors or detectable hardware problems seemed to be related.

Debugging Approach: By setting our tracing tool to log trace data continuously to a circular buffer in memory, and stopping tracing when the error condition was detected, we were able to capture the events preceding the problem (from a point in time determined by the buffer size, e.g. 1GB of RAM) up until it was reported as a timeout. Looking at the start and end times for write requests matching the process ID reporting the timeout, it was easy to see which request was causing the problem.

By then looking at the submissions and removals from the IO scheduler (all of which are instrumented), it was obvious that there was a huge spike in IO traffic at the same time as the slow write request. Through examining the process ID which was the source of the majority of the IO, we could easily see the cause, or as it turned out in this case, two separate causes:

1. An old legacy process left over from 2.2 kernel era that was doing a full `sync()` call every 30s.
2. The logging process would occasionally decide to rotate its log files, and then call `fsync()` to make sure it was done, flushing several GB of data.

Once the problem was characterized and understood, it was easy to fix.

1. The sync process was removed, as its duties have been taken over in modern kernels by pdflush, etc.

2. The logging process was set to rotate logs more often and in smaller data chunks; we also ensured it ran in a separate thread, so as not to block other parts of the server.

Application developers assumed that since the individual writes to the log files were small, the fsync would be inexpensive; however, in some cases the resulting `fsync` was quite large.

This is a good example of a problem that first appeared to be kernel bug, but was in reality the result of a user-space design issue. The problem occurred infrequently, as it was only triggered by the fsync and sync calls coinciding. Additionally, the visibility that the trace tool provided into system behavior enabled us to make general latency improvements to the system, as well as fixing the specific timeout issue.

4.2 Race condition in OOM killer

Problem summary: In a set of production clusters, the OOM killer was firing with an unexpectedly high frequency and killing production jobs. Existing monitoring tools indicated that these systems had available memory when the OOM condition was reported. Again this problem didn't correlate with any particular application state, and in this case there was no reliable way to reproduce it using a benchmark or load test in a controlled environment.

While the rate of OOM killer events was statistically significant across the cluster, it was too low to enable tracing on a single machine and hope to catch an event in a reasonable time frame, especially since some amount of iteration would likely be required to fully diagnose the problem. As before, we needed a trace system which could tell us what the state of the system was in the time leading up to a particular event. In this case, however, our trace system also needed to be lightweight and safe enough to deploy on a significant portion of a cluster that was actively running production workloads. The effect of tracing overhead needed to be imperceptible as far as the end user was concerned.

Debugging Approach: The first step in diagnosing this problem was creating a trigger to stop tracing when the OOM killer event occurred. Once this was in place we waited until we had several trace logs to examine. It was apparent that we were failing to scan or successfully reclaim a suitable number of pages, so we instrumented the main reclaim loop. For each pass over the LRU list, we recorded the reclaim priority, the number of pages scanned, the number of pages reclaimed, and kept counters for each of 33 different reasons why a page might fail to be reclaimed.

From examining this data for the PID that triggered the OOM killer, we could see that the memory pressure indicator was increasing consistently, forcing us to scan increasing number of pages to successfully reclaim memory. However, suddenly the indicator would be set back to zero for no apparent reason. By backtracking and examining the events for all processes in the trace, we were able to determine see that a different process had reclaimed a different class of memory, and then set the global memory pressure counter back to zero.

Once again, with the problem fully understood, the bug was easy to fix through the use of a local memory pressure counter. However, to send the patch back upstream into the mainline kernel, we first had to convince the external maintainers of the code that the problem was real. Though they could not see the proprietary application, or access the machines, by showing them a trace of the condition occurring, it was simple to demonstrate what the problem was.

4.3 Timeout problems following transition from local to distributed storage

Problem summary: While adapting Nutch/Lucene to a clustered environment, IBM transitioned the filesystem from local disk to a distributed filesystem, resulting in application timeouts.

The software stack consisted of the Linux kernel, the open source Java application Nutch/Lucene, and a distributed filesystem. With so many pieces of software, the number and complexity of interactions between components was very high, and it was unclear which layer was causing the slowdown. Possibilities ranged from sharing filesystem data that should have been local, to lock contention within the filesystem, with the added possibility of insufficient bandwidth.

Identifying the problem was further complicated by the nature of error handling in the Nutch/Lucene application. It consists of multiple monitor threads running periodically to check that each node is executing properly. This separated the error condition, a timeout, from the root cause. It can be especially challenging to find the source of such problems as they are seen only in relatively long tests, in this case of 15 minutes or more. By the time the error condition was detected, its cause is no longer apparent or even observable: it has passed out of scope. Only by examining the complete execution window of the timeout—a two-minute period, with many threads—can one pinpoint the problem.

Debugging Approach: The cause of this slowdown was identified using the LTTng/LTTV tracing toolkit. First, we repeated the test with tracing enabled on each node, including the user-space application. This showed that the node triggering the error condition varied between runs. Next, we examined the trace from this node at the time the error condition occurred in order to learn what happened in the minutes leading up to the error. Inspecting the source code of the reporting process was not particularly enlightening, as it was simply a monitoring process for the whole node. Instead, we had to look at the general activity on this node; which was the most active thread, and what was it doing?

The results of this analysis showed that the most active process was doing a large number of `read` system calls. Measuring the duration of these system calls, we saw that each was taking around 30ms, appropriate for disk or network access, but far too long for reads from the data cache. It thus became apparent that the application was not properly utilizing its cache; increasing the cache size of the distributed system completely resolved the problem.

This problem was especially well suited to an investigation through tracing. The timeout error condition presented by the program was a result of a general slowdown of the system, and as such would not present with any obvious connection with the source of the problem. The only usable source of information was the two-minute window in which the slowdown occurred. A trace of the interactions between each thread and the kernel during this window revealed the specific execution mode responsible for the slowdown.

4.4 Latency problem in printk on slow serialization

Problem Summary: User-space applications randomly suffer from scheduler delays of about 12ms.

While some problems can be blamed on user-space design issues that interact negatively with the kernel, most user-space developers expect certain behaviors from the kernel and unexpected kernel behaviors can directly and negatively impact user-space applications, even if they aren't actually errors. For instance, [2] describes a problem in which an application sampling video streams at 60Hz was dropping frames. At this rate, the application must process one frame every 16.6ms to remain synchronized with incoming data. When tracing the kernel timer interrupt, it became clear that delays in the scheduler were causing the application to miss samples. Particularly interesting was the jitter in timer interrupt latency as seen in Figure 1.

A normal timer IRQ should show a jitter lower than the actual timer period in order to behave properly. However, tracing showed that under certain conditions, the timing jitter was much higher than the timer interval. This was first observed around tracing start and stop. Some timer ticks, accounting for 12ms, were missing (3 timer ticks on a 250HZ system).

Debugging Approach: Instrumenting each `local_irq_{save,restore,disable,enable}` macro provided the information needed to find the problem, and extracting the instruction pointer at each call to these macros revealed exactly which address disabled the interrupts for too long around the problematic behavior.

Inspecting the trace involved first finding occurrences of the problematic out-of-range intervals of the interrupt timer and using this timestamp to search backward for the last `irq_save` or `irq_disable` event. Surprisingly, this was `release_console_sem` from `printk`. Disabling the serial console output made the problem disappear, as evidenced by Figure 2. Disabling interrupts while waiting for the serial port to flush the buffers was responsible for this latency, which not only affects the scheduler, but also general timekeeping in the Linux kernel.

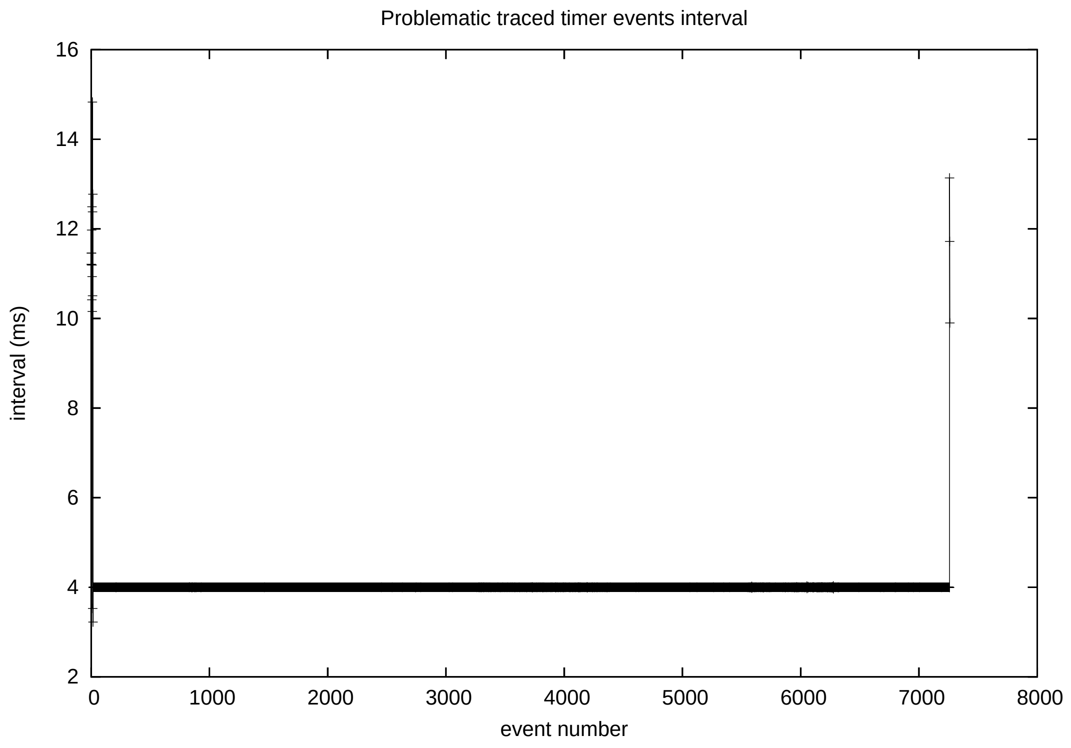

Figure 1: Problematic traced timer events interval

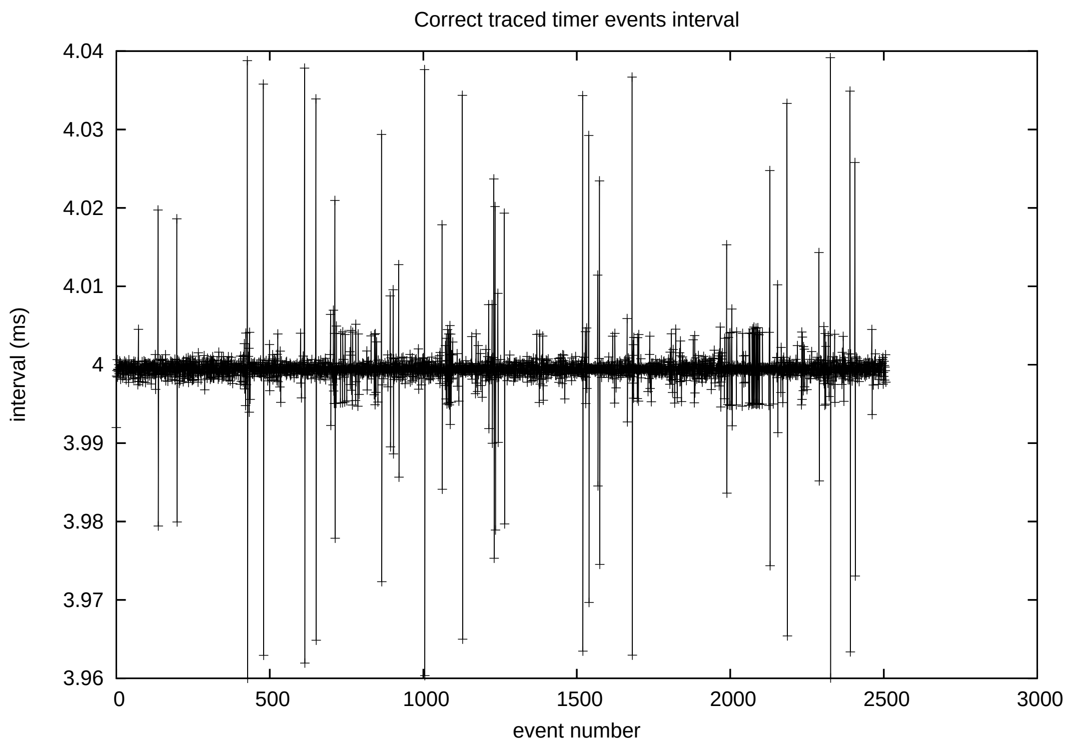

Figure 2: Correct traced timer events interval

4.5 Hardware problems causing a system delay

Problem Summary: The video/audio acquisition software running under Linux at Autodesk, while in development, was affected by delays induced by the PCI-Express version of a particular card. However, the manufacturer denied that their firmware was the cause of the problem, and insisted that the problem was certainly driver or kernel-related.

Debugging Approach: Using LTTng/LTTV to trace and analyze the kernel behavior around the experienced delay led to the discovery that this specific card's interrupt handler was running for too long. Further instrumentation within the handler permitted us to pinpoint the problem more exactly—a register read was taking significantly longer than expected, causing the deadlines to be missed for video and audio sampling. Only when confronted with this precise information did the hardware vendor acknowledge the issue, which was then fixed within a few days.

5 Design and Implementation

We created a hybrid combination of two tracing tools—Google's Ktrace tool and the open source LTTng tool, taking the most essential features from each, while trying to keep the tool as simple as possible. The following set of requirements for tracing was collected from users and from experience through implementation and use:

- When not running, must have zero effective impact.
- When running, should have low enough impact so as not to disturb the problem, or impede production traffic.
- Spooling data off the system should not completely saturate the network.
- Compact data format—must be able to store large amounts of data using as little storage as possible.
- Applicability to a wide range of kernel points, i.e., able to profile in interrupt context, and preferably in NMI context.
- User tools should be able to read multiple different kernel versions, deal with custom debug points, etc.
- One cohesive mechanism (and time ordered stream), not separate tools for scheduler, block tracing, VM tracing, etc.

The resulting design has four main parts described in detail in the sections that follow:

1. a logging system to collect and store trace data and make it available in user-space;
2. a triggering system to identify when an error has occurred and potentially stop tracing;
3. an instrumentation system that meets the performance requirements and also is easily extensible; and
4. an analysis tool for viewing and analyzing the resulting logs.

5.1 Collection and Logging

The system must provide buffers to collect trace data whenever a trace point is encountered in the kernel and have a low-overhead mechanism for making that data available in user-space. To do this we use preallocated, per-CPU buffers as underlying data storage and fast data copy to user-space performed via Relay. When a "trigger" event occurs, assuming the machine is still in a functional state, passing data to user-space is done via simple tools reading the Relay interfaces. If the system has panicked, we may need to spool the data out over the network to another machine (or to local disk), as in the the netdump or crashdump mechanisms.

The in-kernel buffers can be configured to operate in three modes:

- Non-overwrite – when the buffer is full, drop events and increment an event lost counter.
- Overwrite – use the buffer as a circular log buffer, overwriting the oldest data.
- Hybrid – a combination of the two where high rate data is overwritten, but low rate state information is treated as non-overwrite.

Each trace buffer actually consists of a group of per-cpu buffers, each assigned to high, medium, and low rate data. High-rate data accounts for the most common event types described in detail below—system call entry and exits, interrupts, etc. Low-rate data is generally static throughout the trace run and consists in part of the information required to decode the resulting trace, system data type sizes, alignment, etc. Medium-rate channels record meta-information about the system, such as the mapping of interrupt handlers to devices (which might change due to Hotplug), process names, their memory maps, and opened file descriptors. Loaded modules and network interfaces are also treated as medium-rate events. By iterating on kernel data structures we can record a listing of the resources present at trace start time, and update it whenever it changes, thus building a complete picture of the system state.

Separating high-rate events (prone to fill the buffers quickly) from lower rate events allows us to use the maximum space for high-rate data without losing the valuable information provided by the low- and medium-rate channel. Also, it makes it easy to create a hybrid mode system where the last few minutes of interrupt or system call information can be viewed, and we can also get the mapping of process IDs to names even if they were not created within that time window.

Multiple channels can also be used to perform fast user-space tracing, where each process is responsible for writing the trace to disk by itself without going through a system call and Xen hypervisor tracing. The trace merging is performed by the analysis tool in the same manner in which the multiple CPU buffers are handled, permitting merging the information sources at post-processing time.

It may also be useful to integrate other forms of information into the trace, in order to get one merged stream of data—i.e., we could record readprofile-style data (where the instruction pointer was at a given point in time) either in the timer tick event, or as a periodic dump of the collated hash table data. Also functions to record meminfo, slabinfo, ps data, user-space and kernel stacks for the running threads might be useful, though these would have to be enabled on a custom basis. Having all the data in one place makes it significantly easier to write analysis and visualization tools.

5.2 Triggering

Often we want to capture the state of the system in a short period of time preceding a critical error or event. In order to avoid generating massive amounts of data and the performance impact of disk or network writes to the system, we leave the system logging into a circular buffer, then stop tracing when the critical event occurs.

To do this, we need to create a trigger. If this event can easily be recognized by a user-space daemon, we can simply call the usual tracing interface with an instruction to stop tracing. For some situations, a small in-kernel trigger is more appropriate. Typical trigger events we have used include:

- OOM kill;
- Oops / panic;
- User-space locks up (processes are not getting scheduled);
- User application indicates poor response from system; or
- Manual intervention from user.

5.3 Instrumentation

When an instrumentation point is encountered, the tracer takes a timestamp and the associated event data and logs it to our buffers. Each encountered instrumentation point must have minimum overhead, while providing the most information.

Section 5.3.1 explains how our system minimizes the impact of instrumentation and compares and contrasts static and dynamic instrumentation schemes.

We will discuss the details of our event formats in Section 5.3.2 and our approach to timestamping in Section 5.3.3.

To eliminate cache-line bouncing and potential race conditions, each CPU logs data to its own buffer, and system-wide event ordering is done via timestamps. Because we would like to be able to instrument reentrant contexts, we must provide a locking mechanism to avoid potential race conditions. We have investigated two options described in Section 5.3.4.

5.3.1 Static vs. Dynamic Instrumentation Points

There are two ways we can insert trace points—at static markers that are pre-defined in the source code, or dynamically insert them while the system is running. For standard events that we can anticipate the need for in advance, the static mechanism has several advantages. For events that are not anticipated in advance, we can either insert new static points in the source code, compile a new kernel and reboot, or insert dynamic probes via a mechanism such as kprobes. Static vs dynamic markers are compared below:

- Trace points from static markers are significantly faster in use. Kprobes uses a slow int3 mechanism; development efforts have been made to create faster dynamic mechanisms, but they are not finished, very complex, cannot instrument fully preemptible kernels, and they are still significantly slower than static tracing.
- Static trace points can be inserted anywhere in the code base; dynamic probes are limited in scope.
- Dynamic trace points cannot easily access local variables or registers at arbitrary points within a function.
- Static trace points are maintained within the kernel source tree and can follow its evolution; dynamic probes require constant maintenance outside of the tree, and new releases if the traced code changes. This is more of a problem for kernel developers, who mostly work with mainline kernels that are constantly changing.
- Static markers have a potential performance impact when not being used—with care, they can be designed so that this is practically non-existent, and this can be confirmed with performance benchmarks.

We use a marker infrastructure which is a hook-callback mechanism. Hooks are our markers placed in the kernel at the instrumentation site. When tracing is enabled, these are connected to the callback probes—the code executed to perform the tracing. The system is designed to have an impact as low as possible on the system performance, so markers can be compiled into a production kernel without appreciable performance impact. The probe callback connection to its markers is done dynamically. A predicted branch is used to skip the hook stack setup and function call when the marker is "disabled" (no probe is connected). Further optimizations can be implemented for each architecture to make this branch faster.

The other key facet of our instrumentation system is the ability to allow the user to extend it. It would be impossible to determine in advance the complete set of information that would be useful for a particular problem, and recording every thing occurring on a system would be clearly be impractical if not infeasible. Instead, we have designed a system for adding instrumentation iteratively from a coarse-grained level including major events like system calls, scheduling, interrupts, faults, etc. to a finer grained level including kernel synchronization primitives and important user-space functions. Our tool is capable of dealing with an extensible set of user-definable events, including merged information coming from both kernel and user-space execution contexts, synchronized in time.

Events can also be filtered; the user can request which event types should be logged, and which should not. By filtering only by event type, we get an effective, if not particularly fine-grained filter, and avoid the concerns over inserting buggy new code into the kernel, or the whole new languages that tools like Dtrace and Systemtap invent in order to fix this problem. In essence, we have chosen to do coarse filtering in the kernel, and push the rest of the task to user-space. This design is backed up by our efficient probes and logging, compact logging format, and efficient data relay mechanism to user-space (Relay).

5.3.2 Event Formats

It would be beneficial to log as much data about the system state as possible, but instrumenting every interrupt or system call clearly will rapidly generate large volumes of data. To maximize the usefulness of our tool, we must store our event data in the most efficient way possible. In Google's ktrace tool, for the sake of compactness and alignment we chose to make our most common set of events take up 8 bytes. The best compromise between data compactness and information completeness within these bytes was to use the first 4 bytes for type and timestamp information, and the second 4 for an

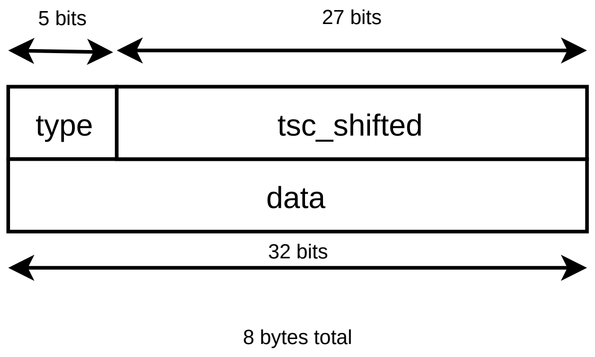

Figure 3: Common event format

event-specific data payload. The format of our events is shown in Figure 3.

Commonly logged events include:

- System call entry / exit (including system call number, lower bytes of first argument)
- Interrupt entry / exit
- Schedule a new task
- Fork / exec of a task, new task seen
- Network traffic
- Disk traffic
- VM reclaim events

In addition to the basic compact format, we required a mechanism for expanding the event space and logging data payloads larger than 4 bytes. We created an expanded event format, shown in Figure 4, that can be used to store larger events needing more data payload space (up to 64K). The normal 32-bit data field is broken into a major and minor expanded event types (256 of each) and a 16-bit length field specifying the length of the data payload that follows.

LTTng's approach is similar to Ktrace; we use 4-byte event headers, followed by a variable size payload. The compact format is also available; it records the timestamp, the event ID, and the payload in 4 bytes. It dynamically calculates the minimum number of bits required to represent the TSC and still detect overflows. It uses the timer frequency and CPU frequency to determine this value.

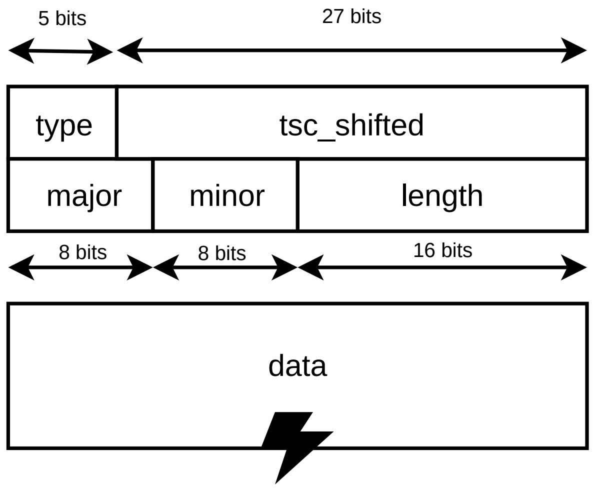

Figure 4: Expanded event format

5.3.3 Timestamps

Our instrumentation system must provide an accurate, low-overhead timestamp to associate with each logged event. The ideal timestamp would be a high-resolution fixed frequency counter, that has very low cost to retrieve, is always monotonic, and is synchronized across all CPUs and readable from both kernel and user-space. However, due to the constraints of current hardware, we are forced to an uncomfortable compromise.

If we look at a common x86-style architecture (32- or 64-bit), choices of time source include PIT, TSC, and HPET. The only time source with acceptable overhead is TSC; however, it is not constant frequency, or well synchronized across platforms. It is also too high-frequency to be compactly logged. The chosen compromise has been to log the TSC at every event, truncated (both on the left and right sides)—effectively, in Ktrace:

$$tsc_{timestamp} = (tsc >> 10)\&(2^{27})$$

On a 2GHz processor, this gives an effective resolution of 0.5us, and takes 27 bits of space to log. LTTng calculates the shifting required dynamically.

However, this counter will roll over every 128 seconds. To ensure we can both unroll this information properly and match it up to the wall time (e.g. to match user-space events) later, we periodically log a timestamp event:

A new timestamp event must be logged:

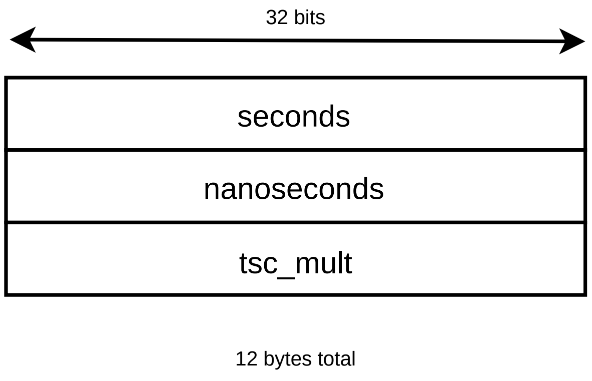

Figure 5: Timestamp format

1. More frequently than the logged timestamp derived from the TSC rolls over.
2. Whenever TSC frequency changes.
3. Whenever TSCs are resynchronized between CPUs.

The effective time of an event is derived by comparing the event TSC to the TSC recorded in the last timestamp and multiplying by a constant representing the current processor frequency.

$$\delta_{walltime} = (event_{tsc} - timestamp_{tsc}) * k_{tsc\_freq}$$

$$event_{walltime} = \delta_{walltime} + timestamp_{walltime}$$

5.3.4 Locking

One key design choice for the instrumentation system for this tool was how to handle potential race conditions from reentrant contexts. The original Google tool, Ktrace, protected against re-entrant execution contexts by disabling interrupts at the instrumentation site, while LTTng uses a lock-less algorithm based on atomic operations local to one CPU (`asm/local.h`) to take timestamps and reserve space in the buffer. The atomic method is more complex, but has significant advantages—it is faster, and it permits tracing of code paths reentering even when IRQs are disabled (lockdep lock dependency checker instrumentation and NMI instrumentation are two examples where is has shown to be useful). The performance improvement of using atomic operations (local compare-and-exchange: 9.0ns) instead of disabling interrupts (save/restore: 210.6ns) on a 3GHz Pentium 4 removes 201.6ns from each probe's execution time. Since the average probe duration of LTTng is about 270ns in total, this is a significant performance improvement.

The main drawback of the lock-less scheme is the added code complexity in the buffer-space reservation function. LTTng's reserve function is based on work previously done on the K42 research kernel at IBM Research, where the timestamp counter read is done within a compare-and-exchange loop to insure that the timestamps will increment monotonically in the buffers. LTTng made some improvements in how it deals with buffer boundaries; instead of doing a separate timestamp read, which can cause timestamps of buffer boundaries to go backward compared to the last/first events, it computes the offsets of the buffer switch within the compare-and-exchange loop and effectively does it when the compare-and-exchange succeeds. The rest of the callbacks called at buffer switch are then called out-of-order. Our merged design considered the benefit of such a scheme to outweigh the complexity.

5.4 Analysis

There are two main usage modes for the tracing tools:

- Given an event (e.g. user-space lockup, OOM kill, user-space noticed event, etc.), we want to examine data leading up to it.
- Record data during an entire test run, sift through it off-line.

Whenever an error condition is not fatal or recurring, taking only one sample of this condition may not give a full insight into what is really happening on the system. One has to verify whether the error is a single case or periodic, and see if the system always triggers this error or if it sometimes shows a correct behavior. In these situations, recording the full trace of the systems is useful because it gives a better overview of what is going on globally on the system.

However, this approach may involve dealing with huge amounts of data, in the order of tens of gigabytes per node. The Linux Trace Toolkit Viewer (LTTV) is designed to do precisely this. It gives both a global graphical overview of the trace, so patterns can be easily identified, and permits the user to zoom into the trace to get the highest level of detail.

Multiple different user-space visualization tools have been written (in different languages) to display or process the tracing data, and it's helpful for them to share this pre-processing phase. These tools fall into two categories:

1. Text printer – one event per line, formatted in a way to make it easy to parse with simple scripts, and fairly readable by a kernel developer with some experience and context.
2. Graphical – easy visualization of large amounts of data. More usable by non-kernel-developers.

6 Future Work

The primary focus of this work has been on creating a single-node trace tool that can be used in a clustered environment, but it is still based on generating a view of the state of a single node in response to a particular trigger on that node. This system lacks the ability to track dependent events between nodes in a cluster or to follow dependencies between nodes. The current configuration functions well when the problem can be tracked to a single node, but doesn't allow the user to investigate a case where events on another system caused or contributed to an error. To build a cluster-wide view, additional design features would be needed in the triggering, collection, and analysis aspects of the trace tool.

- Ability to start and stop tracing on across an entire cluster when a trigger event occurs on one node.
- Low-overhead method for aggregating data over the network for analysis.
- Sufficient information to analyze communication between nodes.
- A unified time base from which to do such analysis.
- An analysis tool capable of illustrating the relationships between systems and displaying multiple parallel traces.

Relying on NTP to provide said synchronization appears to be too imprecise. Some work has been started in this area, primarily aiming at using TCP exchanges between nodes to synchronize the traces. However, it is restrained to a limited subset of network communication: it does not deal with UDP and ICMP packets.

References

[1] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX '04*, 2004.

[2] Mathieu Desnoyers and Michel Dagenais. Low disturbance embedded system tracing with linux trace toolkit next generation. In *ELC (Embedded Linux Conference) 2006*, 2006.

[3] Mathieu Desnoyers and Michel Dagenais. The lttng tracer : A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium) 2006*, pages 209–224, 2006.

[4] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *OLS (Ottawa Linux Symposium) 2005*, 2005.

[5] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing, 2003 ACM/IEEE Conference*, 2003.

[6] Karim Yaghmour and Michel R. Dagenais. The linux trace toolkit. *Linux Journal*, May 2000.

[7] Tom Zanussi, Karim Yaghmour Robert Wisniewski, Richard Moore, and Michel Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *OLS (Ottawa Linux Symposium) 2003*, pages 519–531, 2003.

Proceedings of the Linux Symposium

Volume One

June 27th–30th, 2007
Ottawa, Ontario
Canada