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, bet-

ter 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 inter-communication 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.

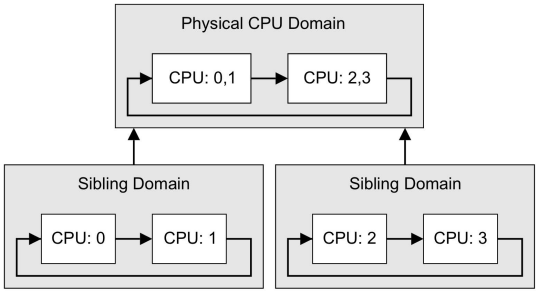

Figure 1: SMT Domains

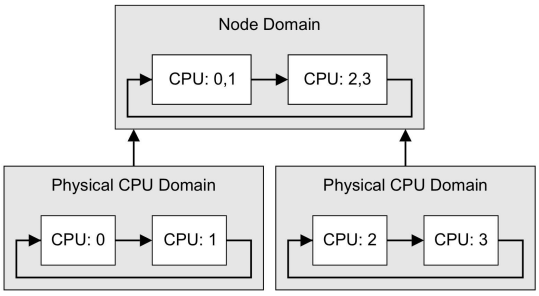

Figure 2: NUMA Domains

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

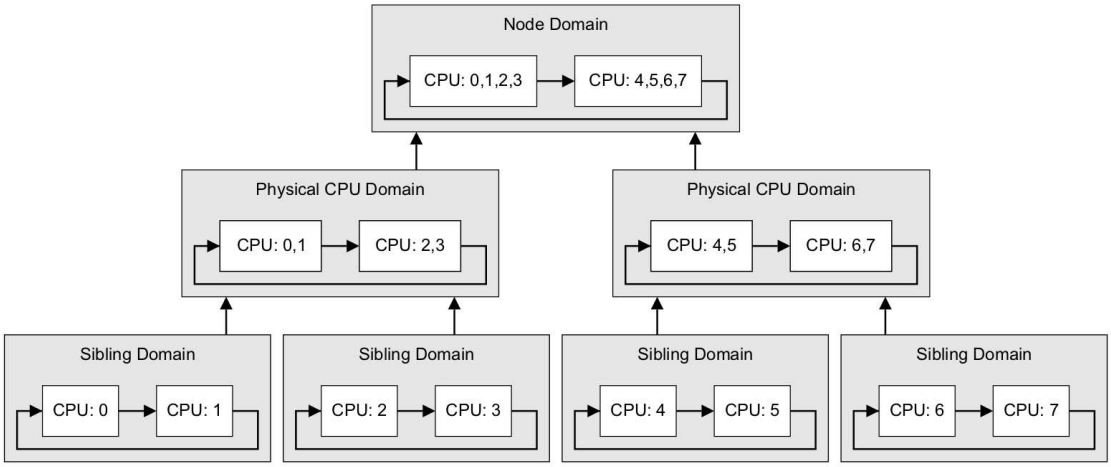

Figure 3: SMT NUMA Domains

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

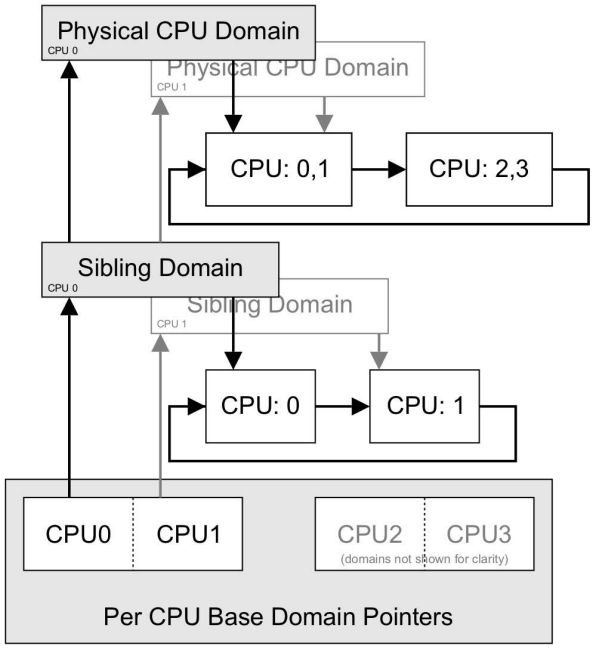

Figure 5: Per CPU Domains

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

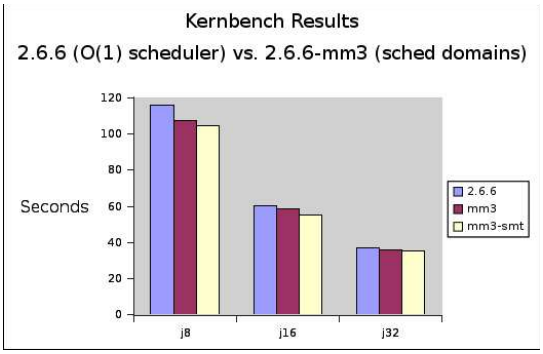

Figure 7: Kernbench Results

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 across 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 NUMAness 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_structs` 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()`.

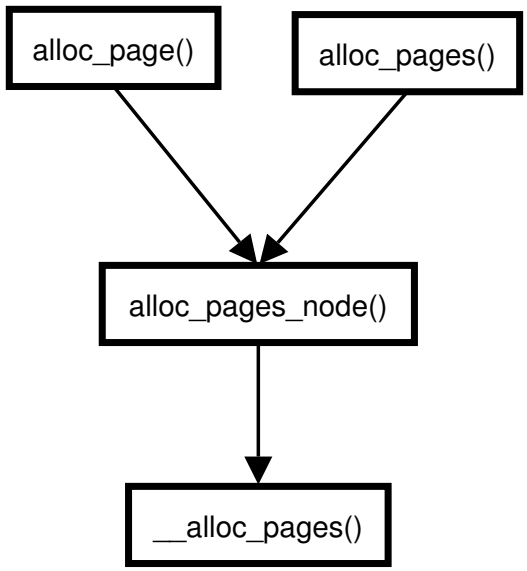

Figure 8: old `alloc_pages`

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

References

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

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

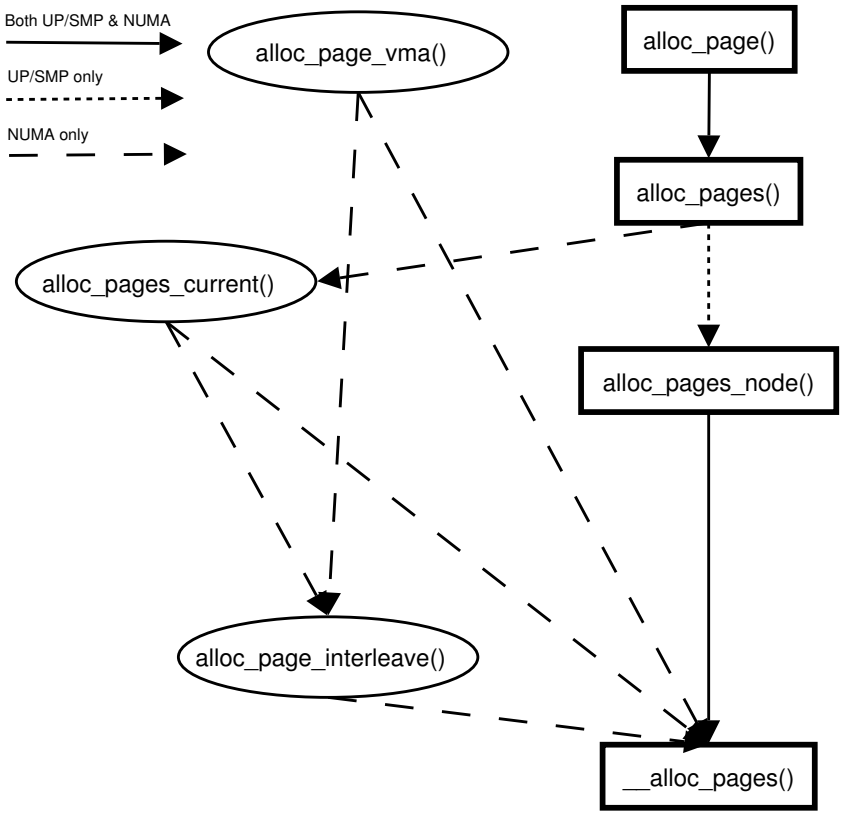

Figure 9: new `alloc_pages`

Proceedings of the Linux Symposium

Volume One

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

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*