Proceedings of the
Linux Symposium

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

Conference Organizers

Review Committee

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

Kernel Scalability—Expanding the Horizon Beyond Fine Grain Locks

Corey Gough
Intel Corp.
corey.d.gough@intel.com

Suresh Siddha
Intel Corp.
suresh.b.siddha@intel.com

Ken Chen
Google, Inc.
kenchen@google.com

Abstract

System time is increasing on enterprise workloads as multi-core and non-uniform memory architecture (NUMA) systems become mainstream. A defining characteristic of the increase in system time is an increase in reference costs due to contention of shared resources—instances of poor memory locality also play a role. An exploration of these issues reveals further opportunities to improve kernel scalability.

This paper examines kernel lock and scalability issues encountered on enterprise workloads. These issues are examined at a software level through structure definition and organization, and at a hardware level through cache line contention characteristics and system performance metrics. Issues and opportunities are illustrated in the process scheduler, I/O paths, timers, slab, and IPC.

1 Introduction

Processor stalls due to memory latency are the most significant contributor to kernel CPI (cycles per instruction) on enterprise workloads. NUMA systems drive kernel CPI higher as cache misses that cost hundreds of clock cycles on traditional symmetric multiprocessing systems (SMP) can extend to over a thousand clock cycles on large NUMA systems. With a fixed amount of kernel instructions taking longer to execute, system time increases, resulting in fewer clock cycles for user applications. Increases in memory latency have a similar effect on the CPI of user applications. As a result, minimizing the effects of memory latency increases on NUMA systems is essential to achieving good scalability.

Cache coherent NUMA systems are designed to overcome limitations of traditional SMP systems, enabling more processors and higher bandwidth. NUMA systems split hardware resources into multiple nodes, with each node consisting of a set of one or more processors and physical memory units. Local node memory references, including references to physical memory on the same node and cache-to-cache transfers between processors on the same node, are less expensive due to lower latency. References that cross nodes, or remote node references, are done at a higher latency due to the additional costs introduced by crossing a node interconnect. As more nodes are added to a system, the cost to reference memory on a far remote node may increase further as multiple interconnects are needed to link the source and destination nodes.

In terms of memory latency, the most expensive kernel memory references can be broadly categorized as *remote memory reads*, or reads from physical memory on a different node, and as *remote cache-to-cache transfers*, or reads from a processor cache on a different node.

Many improvements have been added to the Linux kernel to reduce remote memory reads. Examples include `libnuma` and the kernel `mbind()` interface, NUMA aware slab allocation, and per-cpu scheduler group allocations. These are all used to optimize memory locality. Similarly, many improvements have been added to the kernel to improve lock sequences which decrease latency from cache-to-cache transfers. Use of the RCU (read-copy update) mechanism has enabled several scalability improvements and continues to be utilized in new development. Use of finer grain locks such as array locks for `SYSV` semaphores, conversion of the `page_table_lock` to per pmd locks for fast concurrent page faults, and per device block I/O unplug, all contribute to reduce cache line contention.

Through characterization of kernel memory latency, analysis of high latency code paths, and examination of kernel data structure layout, we illustrate further opportunities to reduce latency.

1.1 Methodology

Kernel memory latency characteristics are studied using an enterprise OLTP (online transaction processing) workload with Oracle Database 10g Release 2 on Dual-Core Intel® Itanium® 2 processors. The configuration includes a large database built on a robust, high performance storage subsystem. The workload is tuned to make best use of the kernel, utilizing fixed process priority and core affinity, optimized binding of interrupts, and use of kernel tunable settings where appropriate.

Itanium processors provide a mechanism to precisely profile cache misses. The Itanium EAR (event address registers) performance monitoring registers latch data reads that miss the L1 cache and collect an extended set of information about targeted misses. This includes memory latency in clock cycles, the address of the load instruction that caused the miss, the processor that retired the load, and the address of the data item that was missed.

Several additional processing steps are taken to supplement the data collected by hardware. Virtual data addresses are converted to a physical address during collection using the *tpa* (translate physical address) instruction and the resulting physical addresses are looked up against `SRAT` (static resource affinity table) data structures to determine node location. Kernel symbols and `gdwarf-2` annotated assembly code are analyzed to translate data addresses to kernel global variable names, structure names, and structure field names. Symbols are enhanced by inlining spinlocks and sinking them into wrapper functions, with the exported function name describing the lock being acquired, where it is acquired, and by whom it is acquired.

2 Cache Coherency

To maintain consistency between internal caches and caches on other processors, systems use a cache coherency protocol, such as the MESI protocol (modified, exclusive, shared, invalid). Each cache line contains status flags that indicate the current cache line state.

A cache line in *E* (exclusive) state indicates that the cache line does not exist in any other processor's cache. The data is clean; it matches the image in main memory.

A cache line in *S* (shared) state can exist in several caches at once. This is frequently the case for cache lines that contain data that is read, but rarely, if ever, modified.

Cache lines in *M* (modified) state are only present in one processor cache at a time. The data is dirty; it is modified and typically does not match the image in main memory. Cache lines in M state can be directly transferred to another processors cache, with the ability to satisfy another processor's read request detected through snooping. Before the cache line can be transferred, it must be written back to main memory.

Cache lines in *I* (invalid) state have been invalidated, and they cannot be transferred to another processor's cache. Cache lines are typically invalidated when there are multiple copies of the line in S state, and one of the processors needs to invalidate all copies of the line so it can modify the data. Cache lines in I state are evicted from a cache without being written back to main memory.

2.1 Cache Line Contention

Coherency is maintained at a cache line level—the coherency protocol does not distinguish between individual bytes on the same cache line. For kernel structures that fit on a single cache line, modification of a single field in the structure will result in any other copies of the cache line containing the structure to be invalidated.

Cache lines are contended when there are several threads that attempt to write to the same line concurrently or in short succession. To modify a cache line, a processor must hold it in M state. Coherency operations and state transitions are necessary to accomplish this, and these come at a cost. When a cache line containing a kernel structure is modified by many different threads, only a single image of the line will exist across the processor caches, with the cache line transferring from cache to cache as necessary. This effect is typically referred to as *cache line bouncing*.

Cache lines are also contended when global variables or fields that are frequently read are located on the same cache line as data that is frequently modified. Coherency operations and state transitions are required to transition cache lines to S state so multiple processors can hold a copy of the cache line for reading. This effect is typically referred to as *false sharing*.

Cache line contention also occurs when a thread referencing a data structure is migrated to another processor, or when a second thread picks up computation based on a structure where a first thread left off—as is the case in interrupt handling. This behavior mimics contention between two different threads as cache lines need to be transferred from one processor's cache to another to complete the processing.

Issues with cache line contention expand further with several of the critical kernel structures spanning multiple cache lines. Even in cases where a code path is referencing only a few fields in a structure, we frequently have contention across several different cache lines.

Trends in system design further intensify issues with cache line contention. Larger caches increase the chance that a cache miss on a kernel structure will hit in a processor's cache. Doubling the number of cores and threads per processor also increases the number of threads that can concurrently reference a kernel structure. The prevalence of NUMA increases the number of nodes in a system, adding latency to the reference types mentioned in the preceding section. Cache line contention that appears fairly innocuous on small servers today has the potential to transform into significant scalability problems in the future.

3 NUMA Costs

In an experiment to characterize NUMA costs, a system is populated with two processors and 64 GB of memory using two different configurations. In the *single node* configuration, both processors and memory are placed into a single node—while the platform is NUMA, this configuration is representative of traditional SMP. In the split node configuration, processors and memory are divided evenly across two NUMA nodes. This experiment essentially toggles NUMA on and off without changing the physical computing resources, revealing interesting changes in kernel memory latency.

With the OLTP workload, scalability deteriorated when comparing the 2.6.18 kernel to the 2.6.9 kernel. On the 2.6.9 kernel, system time increases from 19% to 23% when comparing single node to split node configurations. With less processor time available for the user workload, we measure a 6% performance degradation. On the 2.6.18 kernel, comparing single node to split node configurations, system time increases from 19% to 25%, resulting in a more significant 10% performance degradation.

Kernel data cache miss latency increased 50% as we moved from single to split nodes, several times greater than the increase in user data cache miss latency. A kernel memory latency increase of this magnitude causes several of the critical kernel paths to take over 30% longer to execute—with kernel CPI increasing from 1.95 to 2.55. The two kernel paths that exhibited the largest increases in CPI were in process scheduling and I/O paths.

A comparison of single to split node configurations also reveals that a surprisingly large amount of kernel data cache misses are satisfied by cache-to-cache transfers. The most expensive cache misses were due to remote cache-to-cache transfers—reads from physical memory on a remote node accounted for a much smaller amount of the overall kernel memory latency.

4 2.6.20 Kernel

4.1 Memory Latency Characterization

Figure 1 illustrates the frequency of kernel misses at a given latency value. This data was collected using a NUMA system with two fully-populated nodes on the 2.6.20 kernel running the OLTP workload. Micro-benchmarks were used to measure latency lower bounds, confirming assumptions regarding latency of different reference types.

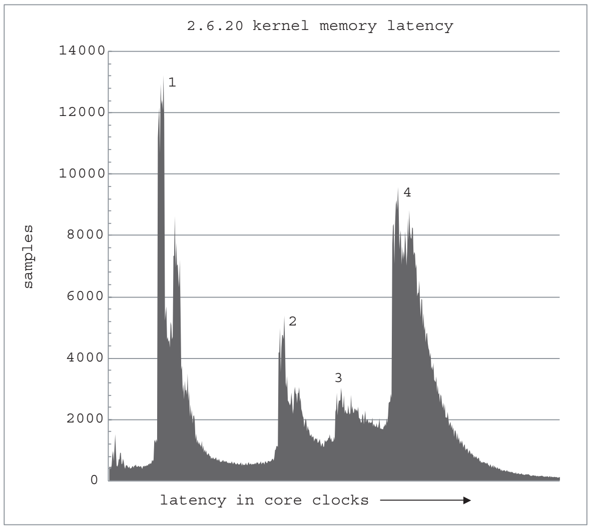

Figure 1: Kernel memory latency histogram

Four significant peaks exist on the kernel memory latency histogram. The first, largest peak corresponds to local node cache-to-cache transfers and the second peak corresponds to local node reads from main memory. The third peak corresponds to a remote node read from main memory, and the fourth peak highlights remote node cache-to-cache transfers. Peaks do not represent absolute boundaries between different types of references as the upper bound is not fixed for any reference type. They do, however, illustrate specific latency ranges where one reference type is much more frequent than another.

46% of the kernel data cache misses are node local, however, these inexpensive misses only account for 27% of the overall kernel latency. The remote node cache misses are considerably more expensive, particularly the remote node cache-to-cache transfers. The height and width of the fourth peak captures the severity of remote node cache-to-cache transfers. The tail of the fourth peak indicates references to a heavily contended cache line, for instance a contended spinlock.

A two node configuration is used in these experiments to illustrate that latency based scalability issues are not exclusive to big iron. As we scale up the number of NUMA nodes, latencies for remote cache-to-cache transfers increase and the chance that cache-to-cache transfers will be local inexpensive references decreases.

Figure 1 also illustrates that the most frequently referenced kernel structures have a tendency to hit in processor caches. Memory reads are, relatively speaking, infrequent. Optimizations targeting the location or NUMA allocation of data do not address cache-to-cache transfers because these are infrequently read from main memory.

In examining memory latency histograms for well tuned user applications, the highest peaks are typically local node cache-to-cache transfers and local node memory reads. The most expensive reference type, the remote node cache-to-cache transfer, is typically the smallest peak. In a well tuned user application, the majority of latency comes from local node references, or is at least split evenly between local and remote node references. In comparison to user applications, the kernel's resource management, communication, and synchronization responsibilities make it much more difficult to localize processing to a single node.

4.2 Cache Line Analysis

With cache-to-cache transfers making up the two tallest peaks in the kernel memory latency histogram, and remote cache-to-cache transfers representing the most expensive overall references, 2.6.20 cache line analysis is focused on only those samples representative of a cache-to-cache transfer.

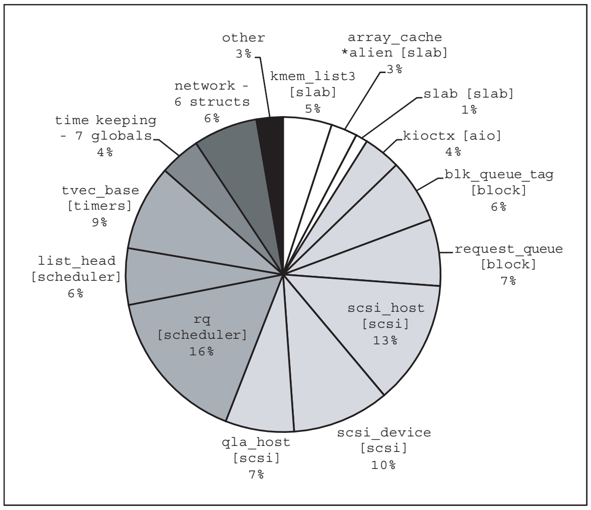

Figure 2: Top 500 kernel cache misses based on total latency

Figure 2 illustrates the top contended structures in the kernel, based on the top 500 cache lines contributing the most to kernel memory latency. While the top 500 cache lines represent only a fraction of 1% of the total kernel cache lines sampled during measurement, they account for 33% of total kernel latency. Among the thousands of structures in the kernel, only twelve structures, spread across scheduler, timers, and I/O make up the majority of the breakdown.

The chart is organized to co-locate nearby or related kernel paths. I/O includes structures from AIO, through block, through SCSI, down to the storage driver. These represent approximately half of the breakdown. Structures for the slab allocation are also included on this side of the chart as the I/O paths and related structures are the primary users of slab allocation. Scheduler paths, including the run queue, and list structures contained in the priority arrays, account for 21% of the breakdown—several percent more if we include IPC as part of this path. Dynamic timers account for 9% of the breakdown.

4.3 Cache Line Visualization

Figure 3 is an example of a concise data format used to illustrate issues with cache line contention. Every path

```
samples   remote  module   inst. address       inst.         function                           data address        data

    338    1.18%  vmlinux  0xA000000100646036  cmpxchg4.acq  [spinlock] @ schedule              0xE000001000054B50  rq_lock
    633   59.40%  vmlinux  0xA000000100069086  cmpxchg4.acq  [spinlock] @ try_to_wake_up        0xE000001000054B50  rq_lock
     67   22.39%  vmlinux  0xA000000100009106  ld4           ia64_spinlock_contention           0xE000001000054B50  rq_lock
      9   77.78%  vmlinux  0xA000000100009126  cmpxchg4.acq  ia64_spinlock_contention           0xE000001000054B50  rq_lock
     72    2.78%  vmlinux  0xA00000010006B386  cmpxchg4.acq  [spinlock] @ task_running_tick     0xE000001000054B50  rq_lock
      1    0.00%  vmlinux  0xA00000010064B840  cmpxchg4.acq  [spinlock] @ wake_sleeping_dep...  0xE000001000054B50  rq_lock
      3   66.67%  vmlinux  0xA00000010006BC76  ld4.acq       resched_task                       0xE000001000054B50  rq_lock
     18   44.44%  vmlinux  0xA000000100069EA6  cmpxchg4.acq  [spinlock] @ try_to_wake_up        0xE000001000054B50  rq_lock

      5    0.00%  vmlinux  0xA000000100646130  ld8           schedule                           0xE000001000054B58  nr_running
      2    0.00%  vmlinux  0xA000000100646726  ld8           schedule                           0xE000001000054B58  nr_running
      2    0.00%  vmlinux  0xA000000100070806  ld8           try_to_wake_up                     0xE000001000054B58  nr_running
      1    0.00%  vmlinux  0xA00000010006CAF0  ld8           move_tasks                         0xE000001000054B58  nr_running
      3    0.00%  vmlinux  0xA00000010006AE60  ld8           deactivate_task                    0xE000001000054B58  nr_running
      1    0.00%  vmlinux  0xA0000001000753C6  ld8           scheduler_tick                     0xE000001000054B58  nr_running
     32   81.25%  vmlinux  0xA00000010006BBD0  ld8           __activate_task                    0xE000001000054B58  nr_running

     30    0.00%  vmlinux  0xA000000100068010  ld8           find_busiest_group                 0xE000001000054B60  raw_weighted_load
      3    0.00%  vmlinux  0xA000000100070720  ld8           try_to_wake_up                     0xE000001000054B60  raw_weighted_load
      2    0.00%  vmlinux  0xA000000100070740  ld8           try_to_wake_up                     0xE000001000054B60  raw_weighted_load
     10    0.00%  vmlinux  0xA000000100070780  ld8           try_to_wake_up                     0xE000001000054B60  raw_weighted_load
      1    0.00%  vmlinux  0xA00000010006CAA6  ld8           move_tasks                         0xE000001000054B60  raw_weighted_load
      1    0.00%  vmlinux  0xA00000010006CAF6  ld8           move_tasks                         0xE000001000054B60  raw_weighted_load
      6    0.00%  vmlinux  0xA000000100075486  ld8           scheduler_tick                     0xE000001000054B60  raw_weighted_load
     21   61.90%  vmlinux  0xA00000010006BBD6  ld8           __activate_task                    0xE000001000054B60  raw_weighted_load

     14    0.00%  vmlinux  0xA000000100068030  ld8           find_busiest_group                 0xE000001000054B68  cpu_load[0]
      2    0.00%  vmlinux  0xA000000100070786  ld8           try_to_wake_up                     0xE000001000054B68  cpu_load[0]
      5    0.00%  vmlinux  0xA000000100070790  ld8           try_to_wake_up                     0xE000001000054B68  cpu_load[0]
      2    0.00%  vmlinux  0xA000000100067F70  ld8           find_busiest_group                 0xE000001000054B68  cpu_load[0]

      1    0.00%  vmlinux  0xA000000100068030  ld8           find_busiest_group                 0xE000001000054B70  cpu_load[1]
      1    0.00%  vmlinux  0xA000000100075490  ld8           scheduler_tick                     0xE000001000054B70  cpu_load[1]
```

Figure 3: First cache line of rq struct

that references a structure field satisfied by a cache-to-cache transfer is listed. Samples that are local cache hits or cache misses that reference main memory are not included, as a result, we do not see each and every field referenced in a structure. Several columns of data are included to indicate contention hotspots and to assist in locating code paths for analysis.

- samples – The number of sampled cache misses at a given reference point
- remote – The ratio of cache misses that reference a remote node compared to local node references
- module – The kernel module that caused the cache miss
- instruction address – The virtual address for the instruction that caused the miss
- instruction – Indicates the instruction. This illustrates the size of the field, for example ld4 is a load of a four byte field. For locks, the instruction indicates whether a field was referenced atomically, as is the case for a compare and exchange or an atomic increment / decrement
- function – The kernel function that caused the miss. In the case of spinlocks, we indicate the spinlock call as well as the caller function
- data address – The virtual address of the data item missed. This helps to illustrate spatial characteristics of contended structure fields
- data – Structure field name or symbolic information for the data item

4.4 Process Scheduler

The process scheduler run queue structure spans four 128 byte cache lines, with the majority of latency coming from the try_to_wake_up() path. Cache lines from the rq struct are heavily contended due to remote wakeups and local process scheduling referencing the same fields. In the first cache line of rq, the most expensive references to the rq lock and remote references from __activate_task come from the wakeup path.

Remote wakeups are due to two frequent paths. First, database transactions are completed when a commit occurs—this involves a write to an online log file. Hundreds of foreground processes go to sleep in the final

samples	remote	module	inst. address	inst.	function	data address	data
2	0.00%	vmlinux	0xA000000100647420	ld8	schedule	0xE000001000014B80	nr_switches
1	0.00%	vmlinux	0xA000000100646100	ld8	schedule	0xE000001000014B88	nr_uninterruptible
2	0.00%	vmlinux	0xA00000010006CB60	ld8	move_tasks	0xE000001000014B98	most_recent_timestamp
752	**63.70%**	**vmlinux**	**0xA000000100066000**	**ld8**	**idle_cpu**	**0xE000001000014BA0**	**curr (task_struct*)**
5	0.00%	vmlinux	0xA00000010006C4F0	ld8	move_tasks	0xE000001000014BA0	curr (task_struct*)
6	0.00%	vmlinux	0xA0000001006468B0	ld8	schedule	0xE000001000014BA0	curr (task_struct*)
1	0.00%	vmlinux	0xA000000100070F10	ld8	try_to_wake_up	0xE000001000014BA0	curr (task_struct*)
799	**65.96%**	**vmlinux**	**0xA000000100066006**	**ld8**	**idle_cpu**	**0xE000001000014BA8**	**idle (task_struct*)**
1	0.00%	vmlinux	0xA000000100646770	ld8	schedule	0xE000001000014BA8	idle (task_struct*)
10	0.00%	vmlinux	0xA0000001006468B6	ld8	schedule	0xE000001000014BA8	idle (task_struct*)
4	0.00%	vmlinux	0xA000000100647056	ld8	schedule	0xE000001000014BA8	idle (task_struct*)
1	0.00%	vmlinux	0xA0000001006472E0	ld8	schedule	0xE000001000014BA8	idle (task_struct*)
31	0.00%	vmlinux	0xA0000001000753A6	ld8	scheduler_tick	0xE000001000014BA8	idle (task_struct*)
4	0.00%	vmlinux	0xA000000100065C90	ld8	account_system_time	0xE000001000014BA8	idle (task_struct*)
176	**1.14%**	**vmlinux**	**0xA000000100645E46**	**ld8**	**schedule**	**0xE000001000014BA8**	**idle (task_struct*)**
2	0.00%	vmlinux	0xA000000100075526	ld8	scheduler_tick	0xE000001000014BB0	next_balance
3	0.00%	vmlinux	0xA000000100647DE6	ld8	schedule	0xE000001000014BB8	prev_mm (mm_struct*)
6	0.00%	vmlinux	0xA000000100646950	ld8	schedule	0xE000001000014BC0	active (prio_array*)
1	0.00%	vmlinux	0xA00000010006B306	ld8	task_running_tick	0xE000001000014BC0	active (prio_array*)
17	64.71%	vmlinux	0xA00000010006BB86	ld8	__activate_task	0xE000001000014BC0	active (prio_array*)
5	0.00%	vmlinux	0xA00000010006C4F6	ld8	move_tasks	0xE000001000014BC8	expired (prio_array*)
13	0.00%	vmlinux	0xA00000010006AD86	ld4	dequeue_task	0xE000001000014BD0	arrays[0].nr_active
2	50.00%	vmlinux	0xA00000010006B2B6	ld4	enqueue_task	0xE000001000014BD0	arrays[0].nr_active
1	0.00%	vmlinux	0xA00000010006AE10	ld4.acq	dequeue_task	0xE000001000014BE8	arrays[0].bitmap

Figure 4: Second cache line of `rq` struct

stage of completing a transaction and need to be woken up when their transaction is complete. Second, database processes are submitting I/O on one processor and the interrupt is arriving on another processor, causing processes waiting for I/O to be woken up on a remote processor.

A new feature being discussed in the Linux kernel community, called syslets, could be used to address issues with remote wakeups. Syslets are simple lightweight programs consisting of system calls, or *atoms*, that the kernel can execute autonomously and asynchronously. The kernel utilizes a different thread to execute the atoms asynchronously, even if a user application making the call is single threaded. Using this mechanism, user applications can wakeup foreground processes on local nodes in parallel by splitting the work between a number of syslets.

A near term approach, and one complementary to node local wakeups using syslets, would be to minimize the number of remote cache line references in the `try_to_wake_up()` path. Figure 3 confirms that the run queue is not cache line aligned. By aligning the run queue, the `rq` structure uses one less cache line. This results in a measurable performance increase as the scheduler stats `ttwu` field on the fourth cache line is brought into the third cache line alongside other data used in the wakeup.

The second cache line of the `rq` struct in figure 4 shows a high level of contention between the `idle` and `curr task_struct` pointers in `idle_task`. This issue originates from the decision to schedule a task on an idle sibling if the processor targeted for a wakeup is busy. In this path, we reference the sibling's runqueue to check if it is busy or not. When a wakeup occurs remotely, the sibling's runqueue status is also checked, resulting in additional remote cache-to-cache transfers. Load balancing at the SMT `sched_domain` happens more frequently, influencing the equal distribution of the load between siblings. Instead of checking the sibling, `idle_cpu()` can simply return the target cpu if there is more than one task running, because it's likely that siblings will also have tasks running. This change reduces contention on the second cache line, and also results in a measurable performance increase.

The third line contains one field that contributes to cache line contention, the `sd sched_domain` pointer used in `try_to_wake_up()`. The fourth `rq` cache line contains scheduler stats fields, with the most expensive remote references coming from the try to wake up count field mentioned earlier. The `list_head` structures in the top 500 are also contended in the wakeup path.

When a process is woken up, it needs to be added to the prio_array queue.

4.5 Timers

Processes use the per-processor tvec_base that corresponds to the processor they are running on when adding timers. This ensures timers are always added locally. The majority of scalability issues with timers are introduced during timer deletion. With I/O submitted on a different processor than it is completed on, it is necessary to acquire a remote tvec_base lock in the interrupt handler to detach a timer from that base.

Figure 5 illustrates cache line contention for the tvec_base lock. The mod_timer() path represents deletes of the block queue plug timers, which are unnecessary given we are using direct I/O in this workload. Cache line contention for the tvec_base lock in del_timer() represents deletes from the scsi_delete_timer() path—the result of deleting a SCSI command eh_timeout timer upon I/O completion.

Getting the base_lock for every I/O completion results in tens-of-thousands of expensive remote cache-to-cache transfers per second. The tvec_base locks are among the most contended locks in this workload. An alternative approach would be to batch timers for deletion, keeping track of a timer's state. A timer's state could be changed locally during an I/O completion since the timer is part of the scsi_cmnd struct that is read into the cache earlier in the path. Timer state could indicate both when the timer can be detached, and when the kernel can free the memory.

Where we have a large number of timers to delete across a number of tvec_base structures, we can prefetch locks, hiding latency for the majority of the lock references in the shadow of the first tvec_base lock referenced.

4.6 Slab

Submitting I/O on one processor and handling the interrupt on another processor also affects scalability of slab allocation. Comparisons between the 2.6.20 and the 2.6.9 kernel, which did not have NUMA aware slab allocation, indicate the slab allocation is hurting more than it is helping on the OLTP workload. Figures 6, 7, and 8 illustrate scalability issues introduced by freeing slabs from a remote node. The alien pointer in the kmem_list3 structure and the nodeid in the slab structure are mostly read, so we may be able to reduce false sharing of these fields by moving them to another cache line. If cache lines with these fields exist in S state, several copies of the data can be held simultaneously by multiple processors, and we can eliminate some of the remote cache-to-cache transfers.

Further opportunity may exist in using kmem_cache_alloc calls that result in refills to free local slabs currently batched to be freed on remote processors. This has the potential to reduce creation of new slabs, localize memory references for slabs that have been evicted from the cache, and provide more efficient lock references.

A number of additional factors limit scalability. Alien array sizes are limited to 12 and are not resized based on slab tunables. In addition, the total size of alien arrays increases squarely as the number of nodes increases, putting additional memory pressure on the system.

A similar analysis for the new slub (unqueued slab) is in progress on this configuration to determine if similar issues exist.

4.7 I/O: AIO

As illustrated throughout this paper, cache line contention of I/O structures is primarily due to submitting I/O on one processor and handling the interrupt on another processor. A superior solution to this scalability issue would be to utilize hardware and device drivers that support extended message-signaled interrupts (MSI-X) and support multiple per-cpu or per-node queues. Using extended features of MSI, a device can send *messages* to a specific set of processors in the system. When an interrupt occurs, the device can decide a target processor for the interrupt. In the case of the OLTP workload, these features could be used to enable I/O submission and completion on the same processor. This approach also ensures that slab objects and timers get allocated, referenced, and freed within a local node.

Another approach, explored in earlier Linux kernels, is to delay the I/O submission so it can be executed on the processor or node which will receive the interrupt during I/O completion. Performance results with this approach

```
samples   remote  module   inst. address       inst.        function                           data address        data

     39   17.95%  vmlinux  0xA000000100009106  ld4          ia64_spinlock_contention           0xE00000100658C000  lock
      9   11.11%  vmlinux  0xA000000100009126  cmpxchg4.acq ia64_spinlock_contention           0xE00000100658C000  lock
    307    2.93%  vmlinux  0xA00000010009B4C6  cmpxchg4.acq [spinlock] @ __mod_timer           0xE00000100658C000  lock
    861   58.07%  vmlinux  0xA0000001000995A6  cmpxchg4.acq [spinlock] @ del_timer             0xE00000100658C000  lock
     10    0.00%  vmlinux  0xA00000010009B666  cmpxchg4.acq [spinlock] @ run_timer_softirq     0xE00000100658C000  lock
    982   40.22%  vmlinux  0xA0000001000996C6  cmpxchg4.acq [spinlock] @ mod_timer             0xE00000100658C000  lock
     24    0.00%  vmlinux  0xA0000001000997E6  cmpxchg4.acq [spinlock] @ try_to_del_timer_sync 0xE00000100658C000  lock
      1    0.00%  vmlinux  0xA00000010009B966  cmpxchg4.acq [spinlock] @ run_timer_softirq     0xE00000100658C000  lock

      7   14.29%  vmlinux  0xA00000010009B466  ld8          __mod_timer                        0xE00000100658C008  running_timer (timer_list*)

      3    0.00%  vmlinux  0xA00000010009A906  ld8          internal_add_timer                 0xE00000100658C010  timer_jiffies
     78    1.28%  vmlinux  0xA00000010009B606  ld8          run_timer_softirq                  0xE00000100658C010  timer_jiffies
```

Figure 5: Cache line with tvec_base struct

```
samples   remote  module   inst. address       inst.        function                          data address        data

      9  100.00%  vmlinux  0xA0000001001580F0  ld8          ____cache_alloc_node              0xE00000207945BD00  slabs_partial.next (list_head*)
      5  100.00%  vmlinux  0xA0000001002C3556  ld8          list_del                          0xE00000207945BD00  slabs_partial.next (list_head*)
      1  100.00%  vmlinux  0xA0000001002C3726  ld8          __list_add                        0xE00000207945BD00  slabs_partial.next (list_head*)
      2  100.00%  vmlinux  0xA0000001002C3816  ld8          list_add                          0xE00000207945BD00  slabs_partial.next (list_head*)

      2  100.00%  vmlinux  0xA0000001002C35C0  ld8          list_del                          0xE00000207945BD08  slabs_partial.prev (list_head*)
     22   59.09%  vmlinux  0xA0000001002C36B6  ld8          __list_add                        0xE00000207945BD08  slabs_partial.prev (list_head*)
     17   70.59%  vmlinux  0xA000000100159EE0  ld8          free_block                        0xE00000207945BD08  slabs_partial.prev (list_head*)

      3   66.67%  vmlinux  0xA0000001002C3556  ld8          list_del                          0xE00000207945BD10  slabs_full.next (list_head*)
      1  100.00%  vmlinux  0xA0000001002C3726  ld8          __list_add                        0xE00000207945BD10  slabs_full.next (list_head*)

      2  100.00%  vmlinux  0xA000000100158200  ld8          ____cache_alloc_node              0xE00000207945BD30  free_objects
     24   54.17%  vmlinux  0xA000000100159E10  ld8          free_block                        0xE00000207945BD30  free_objects

    408  100.00%  vmlinux  0xA0000001001580D6  cmpxchg4.acq [spinlock] @ ____cache_alloc_node 0xE00000207945BD40  list_lock
     17    0.00%  vmlinux  0xA00000010015A206  cmpxchg4.acq [spinlock] @ cache_flusharray     0xE00000207945BD40  list_lock
    114  100.00%  vmlinux  0xA00000010015A3B6  cmpxchg4.acq [spinlock] @ __drain_alien_cache  0xE00000207945BD40  list_lock
     18    0.00%  vmlinux  0xA000000100158656  cmpxchg4.acq [spinlock] @ cache_alloc_refill   0xE00000207945BD40  list_lock
     57   84.21%  vmlinux  0xA000000100009106  ld4          ia64_spinlock_contention          0xE00000207945BD40  list_lock
      3  100.00%  vmlinux  0xA000000100009126  cmpxchg4.acq ia64_spinlock_contention          0xE00000207945BD40  list_lock

    727    0.00%  vmlinux  0xA0000001001596B0  ld8          kmem_cache_free                   0xE00000207945BD50  alien (array_cache**)
```

Figure 6: Cache line with kmem_list3 struct

were mixed, with a range of both good and bad results across different workloads. This model was also burdened with the complexity of handling many different corner cases.

Figure 9 illustrates the primary contention point for the kioctx structure, the ctx_lock that protects a per-process AIO context. This structure is dynamically allocated as one per process and lives throughout the lifetime of an AIO context. The first cache line is contended due to lock references in both the I/O submission and completion paths.

One approach to improve the kioctx structure would be to reorder structure fields to create a cache line with fields most frequently referenced in the submit path, and another cache line with fields most frequently referenced in the complete path. Further analysis is required to determine the feasibility of this concept. Contention in the I/O submission path occurs with __aio_get_req which needs the lock to put a iocb on a linked list and with aio_run_iocb which needs the lock to mark the current request as running. Contention in the I/O completion path occurs with aio_complete which needs the lock to put a completion event into event queue, unlink a iocb from a linked list, and perform process wakeup if there are waiters.

Raman, Hundt, and Mannarswamy introduced a technique for structure layout in multi threaded applications that optimizes for both improved spatial locality and reduces false sharing. Reordering fields in a few optimized structures in the HP-UX operating system kernel improved performance up to 3.2% on enterprise workloads. Structures across the I/O layers appear to be good candidates for such optimization.

The aio_complete function is a heavyweight function with very long lock hold times. Having a lock hold time dominated by one processor contributes to longer contention wait time with other shorter paths occurring frequently on all the other processors. In some cases, increasing the number of AIO contexts may help address this.

Figure 10 illustrates the AIO context internal structure used to track the kernel mapping of AIO ring_info

```
samples   remote  module   inst. address       inst.         function                          data address        data

      9    0.00%  vmlinux  0xA000000100159740  ld4           kmem_cache_free                   0xE000000128DA5580  avail
      1    0.00%  vmlinux  0xA000000100159770  ld4           kmem_cache_free                   0xE000000128DA5580  avail

      5    0.00%  vmlinux  0xA00000010015A3F0  ld4           __drain_alien_cache               0xE000000128DA5584  limit
      8    0.00%  vmlinux  0xA000000100159746  ld4           kmem_cache_free                   0xE000000128DA5584  limit

     30    0.00%  vmlinux  0xA000000100009106  ld4           ia64_spinlock_contention          0xE000000128DA5590  lock
      2    0.00%  vmlinux  0xA000000100009126  cmpxchg4.acq  ia64_spinlock_contention          0xE000000128DA5590  lock
   1793    0.00%  vmlinux  0xA000000100159716  cmpxchg4.acq  [spinlock] @ cache_free_alien     0xE000000128DA5590  lock

      2    0.00%  vmlinux  0xA0000001002C07A0  ld8           __copy_user                       0xE000000128DA5598  objpp (entry[0]*)

      2    0.00%  vmlinux  0xA0000001002C07A6  ld8           __copy_user                       0xE000000128DA55A0  objpp[i]
      3    0.00%  vmlinux  0xA000000100159C20  ld8           free_block                        0xE000000128DA55A0  objpp[i]

      4    0.00%  vmlinux  0xA000000100159C20  ld8           free_block                        0xE000000128DA55B0  objpp[i]

      1    0.00%  vmlinux  0xA000000100159C20  ld8           free_block                        0xE000000128DA55C0  objpp[i]

      1    0.00%  vmlinux  0xA000000100159C20  ld8           free_block                        0xE000000128DA55D8  objpp[i]

      2    0.00%  vmlinux  0xA000000100159C20  ld8           free_block                        0xE000000128DA55E0  objpp[i]
```

Figure 7: Cache line with alien `array_cache` struct

```
samples   remote  module   inst. address       inst.         function                          data address        data

      4  100.00%  vmlinux  0xA0000001002C3556  ld8           list_del                          0xE0000010067D0000  list.next (list_head*)
      1  100.00%  vmlinux  0xA0000001002C35B0  ld8           list_del                          0xE0000010067D0000  list.next (list_head*)
      1  100.00%  vmlinux  0xA0000001002C3726  ld8           __list_add                        0xE0000010067D0000  list.next (list_head*)

     88   98.86%  vmlinux  0xA0000001002C35C0  ld8           list_del                          0xE0000010067D0008  list.prev (list_head*)
     79  100.00%  vmlinux  0xA0000001002C36B6  ld8           __list_add                        0xE0000010067D0008  list.prev (list_head*)

     64  100.00%  vmlinux  0xA000000100158160  ld4           ____cache_alloc_node              0xE0000010067D0020  inuse
      2  100.00%  vmlinux  0xA000000100159DF6  ld4           free_block                        0xE0000010067D0020  inuse

      4  100.00%  vmlinux  0xA000000100158226  ld4           ____cache_alloc_node              0xE0000010067D0024  free (kmem_bufctl_t)

    513   45.03%  vmlinux  0xA000000100159636  ld2           kmem_cache_free                   0xE0000010067D0028  nodeid
```

Figure 8: Cache line with `slab` struct

and the AIO event buffer. A small percentage of the cache line contention comes from with I/O submit path where kernel needs to look up a kernel mapping for an AIO `ring_info` structure. A large percentage of the cache line contention comes from the I/O interrupt path, where `aio_complete` needs to lookup the kernel mapping of the AIO event buffer. In this case, the majority of contention comes from many I/O completions happening one after the other.

4.8 I/O: Block

Figure 11 illustrates contention over `request_queue` structures. The primary contention points are between the `queue_flags` used in the submit path to check block device queue's status and the `queue_lock` in the return path to perform a reference count on `device_busy`. This suggests further opportunity for structure ordering based on submit and complete paths.

Figure 12 illustrates the `blk_queue_tag` structure. Lack of alignment with this structure causes unnecessary cache line contention as the size of `blk_queue_tag` is less than half a cache line, so we end up with portions of two to three different tags sharing the same 128 byte cache line. Cache lines are frequently bounced between processors with multiple tags from independent devices sharing the same cache line.

4.9 I/O: SCSI

Both the `scsi_host` and `scsi_device` structures span several cache lines and may benefit from reordering. Both structures feature a single cache line with multiple contended fields, and several other lines which have a single field that is heavily contended.

Figure 13 illustrates the most heavily contended cache line of the `scsi_host` structure. The primary source of contention is in the submit path with an unconditional spin lock acquire in `__scsi_put_command` to put a `scsi_cmnd` on a local `free_list`, and reads of the `cmd_pool` field. Multiple locks on the same cache line is detrimental as it slows progress in the submit path

```
samples   remote  module   inst. address       inst.         function                            data address        data

      1    0.00%  vmlinux  0xA0000001001AB240  fetchadd4.rel lookup_ioctx                        0xE0000010252A1900  users
      2    0.00%  vmlinux  0xA0000001001ABD60  fetchadd4.rel sys_io_getevents                    0xE0000010252A1900  users

     44    2.27%  vmlinux  0xA0000001001AB1A6  ld8           lookup_ioctx                        0xE0000010252A1910  mm (mm_struct*)

     14   71.43%  vmlinux  0xA000000100066E56  cmpxchg4.acq  [spinlock] @ wake_up_common         0xE0000010252A1920  wait.lock (wait_queue_head)
      1    0.00%  vmlinux  0xA000000100009126  cmpxchg4.acq  ia64_spinlock_contention            0xE0000010252A1920  wait.lock (wait_queue_head)
      4    0.00%  vmlinux  0xA0000001000B9F76  cmpxchg4.acq  [spinlock] @ add_wait_queue_exc..   0xE0000010252A1920  wait.lock (wait_queue_head)
      6    0.00%  vmlinux  0xA0000001000B9FD6  cmpxchg4.acq  [spinlock] @ remove_wait_queue      0xE0000010252A1920  wait.lock (wait_queue_head)

      1    0.00%  vmlinux  0xA0000001002C3556  ld8           list_del                            0xE0000010252A1928  wait.next
      5   60.00%  vmlinux  0xA000000100065EB0  ld8           __wake_up_common                    0xE0000010252A1928  wait.next
      4   75.00%  vmlinux  0xA0000001001A9F90  ld8           aio_complete                        0xE0000010252A1928  wait.next

    372    1.34%  vmlinux  0xA0000001001AA2D6  cmpxchg4.acq  [spinlock] @ aio_run_iocb           0xE0000010252A1938  ctx_lock
   1066    1.13%  vmlinux  0xA0000001001AAA36  cmpxchg4.acq  [spinlock] @ __aio_get_req          0xE0000010252A1938  ctx_lock
     14    0.00%  vmlinux  0xA0000001001ACC36  cmpxchg4.acq  [spinlock] @ io_submit_one          0xE0000010252A1938  ctx_lock
   1214   62.03%  vmlinux  0xA0000001001A8FB6  cmpxchg4.acq  [spinlock] @ aio_complete           0xE0000010252A1938  ctx_lock
    181   34.25%  vmlinux  0xA000000100009106  ld4           ia64_spinlock_contention            0xE0000010252A1938  ctx_lock
     23   26.09%  vmlinux  0xA000000100009126  cmpxchg4.acq  ia64_spinlock_contention            0xE0000010252A1938  ctx_lock
     33   69.70%  vmlinux  0xA0000001001A9586  ld4.acq       __aio_put_req                       0xE0000010252A1938  ctx_lock
     39   82.05%  vmlinux  0xA0000001001A97D6  ld4.acq       __aio_put_req                       0xE0000010252A1938  ctx_lock
     52    0.00%  vmlinux  0xA0000001001A99B6  cmpxchg4.acq  [spinlock] @ aio_put_req            0xE0000010252A1938  ctx_lock

      4    0.00%  vmlinux  0xA0000001001AABD0  ld4           __aio_get_req                       0xE0000010252A193C  reqs_active
     28   64.29%  vmlinux  0xA0000001001A98C6  ld4           __aio_put_req                       0xE0000010252A193C  reqs_active

      2   50.00%  vmlinux  0xA0000001002C35C0  ld8           list_del                            0xE0000010252A1948  active_reqs.prev

     24    4.17%  vmlinux  0xA0000001001A80D0  ld8           aio_read_evt                        0xE0000010252A1978  ring_info.ring_pages (page**)
      7    0.00%  vmlinux  0xA0000001001A82B0  ld8           aio_read_evt                        0xE0000010252A1978  ring_info.ring_pages (page**)
      1    0.00%  vmlinux  0xA0000001001AAA96  ld8           __aio_get_req                       0xE0000010252A1978  ring_info.ring_pages (page**)
      3   33.33%  vmlinux  0xA0000001001A9C56  ld8           aio_complete                        0xE0000010252A1978  ring_info.ring_pages (page**)
     16   75.00%  vmlinux  0xA0000001001A9D90  ld8           aio_complete                        0xE0000010252A1978  ring_info.ring_pages (page**)
```

Figure 9: Cache line with `kioctx` struct

```
samples   remote  module   inst. address       inst.         function                            data address        data

     55    1.82%  vmlinux  0xA0000001001A8216  cmpxchg4.acq  [spinlock] @ aio_read_evt           0xE0000010252A1980  lock

      2    0.00%  vmlinux  0xA0000001001A8240  ld4           aio_read_evt                        0xE0000010252A1990  nr
      2    0.00%  vmlinux  0xA0000001001A83E0  ld4           aio_read_evt                        0xE0000010252A1990  nr

      3    0.00%  vmlinux  0xA0000001001A9CD0  ld4           aio_complete                        0xE0000010252A1994  tail

     41    0.00%  vmlinux  0xA0000001001A80E0  ld8           aio_read_evt                        0xE0000010252A1998  internal_pages[0]  (page*)
    415    1.20%  vmlinux  0xA0000001001AAAA6  ld8           __aio_get_req                       0xE0000010252A1998  internal_pages[0]  (page*)
   1033   58.95%  vmlinux  0xA0000001001A9C76  ld8           aio_complete                        0xE0000010252A1998  internal_pages[0]  (page*)
      1  100.00%  vmlinux  0xA0000001001A9DB0  ld8           aio_complete                        0xE0000010252A1998  internal_pages[0]  (page*)

      1  100.00%  vmlinux  0xA0000001001A9DB0  ld8           aio_complete                        0xE0000010252A19A0  internal_pages[1]  (page*)

      3    0.00%  vmlinux  0xA0000001001A82D0  ld8           aio_read_evt                        0xE0000010252A19A8  internal_pages[2]  (page*)
```

Figure 10: Cache line with `ring_info` struct

with the cache line bouncing between processors as they submit I/O.

Figure 14 illustrates contention in the `scsi_device` struct with `scsi_request_fn()` referencing the host pointer to process the I/O, and the low level driver checking the `scsi_device` `queue_depth` to determine whether it should change the `queue_depth` on a specific SCSI device. These reads lead to false sharing on the `queue_depth` field, as `scsi_adjust_queue_depth()` is not called during the workload. In this case, the `scsi_qla_host flags` could be extended to indicate whether `queue_depth` needs to be adjusted, resulting in the interrupt service routine making frequent local references rather than expensive remote references.

Figure 15 illustrates further contention between the I/O submit and complete paths as `sd_init_command()` reads of the `timeout` and `changed` fields in the submit path conflict with writes of `iodone_cnt` in the complete path.

5 Conclusions

Kernel memory latency increases significantly as we move from traditional SMP to NUMA systems, resulting in less processor time available for user workloads. The most expensive references, remote cache-to-cache transfers, primarily come from references to a select few structures in the process wakeup and I/O paths. Several approaches may provide mechanisms to alleviate

samples	remote	module	inst. address	inst.	function	data address	data
3	66.67%	vmlinux	0xA0000001001D1F26	ld8	__blockdev_direct_IO	0xE00000012C905C00	backing_dev_info.unplug_io_fn
1	100.00%	vmlinux	0xA00000010029A0A0	ld8	blk_backing_dev_unplug	0xE00000012C905C08	backing_dev_info.unplug_io_data
1	0.00%	scsi	0xA00000021E2A9B80	ld8	scsi_run_queue	0xE00000012C905C10	queuedata
147	**59.86%**	**vmlinux**	**0xA0000001002942E6**	**ld4.acq**	**generic_make_request**	**0xE00000012C905C28**	**queue_flags**
1	0.00%	vmlinux	0xA000000100294750	ld4.acq	__freed_request	0xE00000012C905C28	queue_flags
3	33.33%	vmlinux	0xA000000100294766	cmpxchg4.acq	__freed_request	0xE00000012C905C28	queue_flags
1	0.00%	vmlinux	0xA000000100294780	ld4.acq	__freed_request	0xE00000012C905C28	queue_flags
6	16.67%	vmlinux	0xA000000100294796	cmpxchg4.acq	__freed_request	0xE00000012C905C28	queue_flags
20	55.00%	vmlinux	0xA000000100296B66	cmpxchg4.acq	blk_remove_plug	0xE00000012C905C28	queue_flags
1	100.00%	scsi	0xA00000021E2AF2D6	cmpxchg4.acq	scsi_request_fn	0xE00000012C905C28	queue_flags
1	0.00%	vmlinux	0xA000000100297456	ld4.acq	blk_plug_device	0xE00000012C905C28	queue_flags
4	25.00%	vmlinux	0xA000000100297486	cmpxchg4.acq	blk_plug_device	0xE00000012C905C28	queue_flags
3	66.67%	vmlinux	0xA000000100295930	ld4.acq	get_request	0xE00000012C905C28	queue_flags
6	50.00%	vmlinux	0xA00000010028DE96	ld4.acq	elv_insert	0xE00000012C905C28	queue_flags
6	50.00%	vmlinux	0xA00000010029C046	cmpxchg4.acq	[spinlock] @ __make_request	0xE00000012C905C30	__queue_lock
3	66.67%	vmlinux	0xA000000100290156	cmpxchg4.acq	[spinlock] @ blk_run_queue	0xE00000012C905C30	__queue_lock
18	50.00%	vmlinux	0xA00000010029AB56	cmpxchg4.acq	[spinlock] @ generic_unplug_d..	0xE00000012C905C30	__queue_lock
5	40.00%	scsi	0xA00000021E2ACF06	cmpxchg4.acq	[spinlock] @ scsi_device_unbusy	0xE00000012C905C30	__queue_lock
28	28.57%	vmlinux	0xA000000100009106	ld4	ia64_spinlock_contention	0xE00000012C905C30	__queue_lock
13	23.08%	vmlinux	0xA00000010029B2B6	cmpxchg4.acq	[spinlock] @ __make_request	0xE00000012C905C30	__queue_lock
16	56.25%	scsi	0xA00000021E2A9336	cmpxchg4.acq	[spinlock] @ scsi_end_request	0xE00000012C905C30	__queue_lock
5	20.00%	scsi	0xA00000021E2AF806	cmpxchg4.acq	[spinlock] @ scsi_request_fn	0xE00000012C905C30	__queue_lock
3	33.33%	scsi	0xA00000021E2AFA36	cmpxchg4.acq	[spinlock] @ scsi_request_fn	0xE00000012C905C30	__queue_lock
1	0.00%	vmlinux	0xA00000010029C000	ld8	__make_request	0xE00000012C905C38	queue_lock*
124	**33.06%**	**scsi**	**0xA00000021E2ACEE6**	**ld8**	**scsi_device_unbusy**	**0xE00000012C905C38**	**queue_lock***
2	50.00%	vmlinux	0xA000000100298EE0	ld8	blk_run_queue	0xE00000012C905C38	queue_lock*
5	60.00%	vmlinux	0xA000000100299006	ld8	blk_run_queue	0xE00000012C905C38	queue_lock*
10	30.00%	scsi	0xA00000021E2AB540	ld8	scsi_end_request	0xE00000012C905C38	queue_lock*
1	0.00%	scsi	0xA00000021E2AB5C6	ld8	scsi_end_request	0xE00000012C905C38	queue_lock*
11	45.45%	scsi	0xA00000021E2AF7C6	ld8	scsi_request_fn	0xE00000012C905C38	queue_lock*
1	100.00%	scsi	0xA00000021E2AF9B6	ld8	scsi_request_fn	0xE00000012C905C38	queue_lock*
1	0.00%	scsi	0xA00000021E2AF9F6	ld8	scsi_request_fn	0xE00000012C905C38	queue_lock*

Figure 11: Cache line with `request_queue` struct

samples	remote	module	inst. address	inst.	function	data address	data
119	**56.30%**	**vmlinux**	**0xA000000100299F06**	**ld8**	**blk_queue_start_tag**	**0xE000000128E8BC08**	**tag_map**
128	**54.69%**	**vmlinux**	**0xA000000100299F00**	**ld4**	**blk_queue_start_tag**	**0xE000000128E8BC24**	**max_depth**
113	**28.32%**	**vmlinux**	**0xA000000100291C26**	**ld4**	**blk_queue_end_tag**	**0xE000000128E8BC28**	**real_max_depth**
122	**53.28%**	**vmlinux**	**0xA0000001002B30E0**	**ld8**	**find_next_zero_bit**	**0xE000000128E8BC40**	**tag_map**
96	12.50%	vmlinux	0xA000000100291C66	ld4	blk_queue_end_tag	0xE000000128E8BC40	tag_map
16	50.00%	vmlinux	0xA000000100299F76	cmpxchg4.acq	blk_queue_start_tag	0xE000000128E8BC40	tag_map

Figure 12: Cache line with `blk_queue_tag` struct

scalability issues in the future. Examples include hardware and software utilizing message-signaled interrupts (MSI-X) with per-cpu or per-node queues, and syslets. In the near term, several complementary approaches can be taken to improve scalability.

Improved code sequences reducing remote cache-to-cache transfers, similar to the optimization targeting `idle_cpu()` calls or reducing the number of `rq` cache lines referenced in a remote wakeup are beneficial and need to be pursued further. Proper alignment of structures also reduces cache-to-cache transfers, as illustrated in the `blk_queue_tag` structure. Opportunities exist in ordering structure fields to increase cache line sharing and reduce contention. Examples include separating read only data from contended read / write fields, and careful placement or isolation of frequently referenced spinlocks. In the case of `scsi_host` and to a lesser extent `kioctx`, several atomic semaphores placed on the same cache quickly increase contention for a structure.

Data ordering also extends to good use of `__read_mostly` attributes for global variables, which also have a significant impact. In the 2.6.9 kernel, false sharing occurred between `inode_lock` and a hot read-only global on the same cache line, ranking it number 1 in the top 50. In the 2.6.20 kernel, the hot read-only global was given the `__read_mostly` attribute, and the cache line with `inode_lock` dropped out of the top 50 cache lines. The combination of the two new cache lines contributed 45% less latency with the globals separated.

Operations that require a lock to be held for a single frequently-occurring operation, such a timer delete from

```
samples   remote  module   inst. address       inst.           function                          data address        data

    852   50.23%  scsi     0xA00000021E29DA80  ld8             __scsi_get_command                0xE0000010260C8020  cmd_pool
      2  100.00%  scsi     0xA00000021E29DF10  ld8             __scsi_put_command                0xE0000010260C8020  cmd_pool

    292   69.86%  scsi     0xA00000021E29C536  cmpxchg4.acq   [spinlock] @ scsi_put_command     0xE0000010260C8028  free_list_lock

      3    0.00%  scsi     0xA00000021E29DE60  ld8             __scsi_put_command                0xE0000010260C8030  free_list.next

      5   60.00%  scsi     0xA00000021E2AA0F6  ld8             scsi_run_queue                    0xE0000010260C8040  starved_list.next

    292   40.75%  qla      0xA00000021E4063E6  cmpxchg4.acq    [spinlock] @ qla2x00_queuecom.. 0xE0000010260C8050  default_lock
     41   58.54%  scsi     0xA00000021E29C416  cmpxchg4.acq    [spinlock] @ scsi_dispatch_cmd  0xE0000010260C8050  default_lock
     98   50.00%  vmlinux  0xA000000100009106  ld4             ia64_spinlock_contention          0xE0000010260C8050  default_lock
     14   50.00%  vmlinux  0xA000000100009126  cmpxchg4.acq    ia64_spinlock_contention          0xE0000010260C8050  default_lock
     95   49.47%  scsi     0xA00000021E2AF456  cmpxchg4.acq    [spinlock] @ scsi_request_fn    0xE0000010260C8050  default_lock
     97   67.01%  scsi     0xA00000021E2A95B6  cmpxchg4.acq    [spinlock] @ scsi_device_unbu.. 0xE0000010260C8050  default_lock
     42   78.57%  scsi     0xA00000021E2A9736  cmpxchg4.acq    [spinlock] @ scsi_run_queue     0xE0000010260C8050  default_lock

     27   37.04%  qla      0xA00000021E4062B6  ld8             qla2x00_queuecommand              0xE0000010260C8058  host_lock*
    197   57.36%  qla      0xA00000021E4063A6  ld8             qla2x00_queuecommand              0xE0000010260C8058  host_lock*
     41   48.78%  scsi     0xA00000021E29E3A0  ld8             scsi_dispatch_cmd                 0xE0000010260C8058  host_lock*
      3   66.67%  scsi     0xA00000021E29E526  ld8             scsi_dispatch_cmd                 0xE0000010260C8058  host_lock*
    355   65.63%  scsi     0xA00000021E2ACDB0  ld8             scsi_device_unbusy                0xE0000010260C8058  host_lock*
     14   50.00%  scsi     0xA00000021E2ACEA6  ld8             scsi_device_unbusy                0xE0000010260C8058  host_lock*
    123   55.28%  scsi     0xA00000021E2AF436  ld8             scsi_request_fn                   0xE0000010260C8058  host_lock*
     34   50.00%  scsi     0xA00000021E2AF746  ld8             scsi_request_fn                   0xE0000010260C8058  host_lock*
     38   76.32%  scsi     0xA00000021E2A9E80  ld8             scsi_run_queue                    0xE0000010260C8058  host_lock*
```

Figure 13: Cache line with `scsi_host` struct

```
samples   remote  module   inst. address       inst.          function                          data address        data

     26   38.46%  scsi     0xA00000021E29EB10  ld8            scsi_put_command                  0xE00000012464C000  host (scsi_host*)
      4   25.00%  scsi     0xA00000021E29ED10  ld8            scsi_get_command                  0xE00000012464C000  host (scsi_host*)
     11   45.45%  scsi     0xA00000021E2ACD90  ld8            scsi_device_unbusy                0xE00000012464C000  host (scsi_host*)
     22   59.09%  scsi     0xA00000021E29CDE0  ld8            scsi_finish_command               0xE00000012464C000  host (scsi_host*)
     90   54.44%  scsi     0xA00000021E2AF0F0  ld8            scsi_request_fn                   0xE00000012464C000  host (scsi_host*)
      1  100.00%  scsi     0xA00000021E2A9B96  ld8            scsi_run_queue                    0xE00000012464C000  host (scsi_host*)
      2  100.00%  scsi     0xA00000021E29DFB0  ld8            scsi_dispatch_cmd                 0xE00000012464C000  host (scsi_host*)

      2    0.00%  scsi     0xA00000021E2AAF10  ld8            scsi_next_command                 0xE00000012464C008  request_queue (request_queue*)
      1    0.00%  scsi     0xA00000021E2ACF26  ld8            scsi_device_unbusy                0xE00000012464C008  request_queue (request_queue*)
      1  100.00%  scsi     0xA00000021E2AB826  ld8            scsi_io_completion                0xE00000012464C008  request_queue (request_queue*)

      6   33.33%  scsi     0xA00000021E29C2F6  cmpxchg4.acq   [spinlock] @ scsi_put_command     0xE00000012464C034  list_lock
     21   52.38%  scsi     0xA00000021E29C3B6  cmpxchg4.acq   [spinlock] @ scsi_get_command     0xE00000012464C034  list_lock

      2    0.00%  scsi     0xA00000021E2AF686  ld8            scsi_request_fn                   0xE00000012464C048  starved_entry.next

     92   35.87%  qla      0xA00000021E420D60  ld4            qla2x00_process_completed_req...  0xE00000012464C060  queue_depth

      2  100.00%  qla      0xA00000021E41EDF0  ld4            qla2x00_start_scsi                0xE00000012464C074  lun
```

Figure 14: First cache line of `scsi_device` struct

a `tvec_base`, can be batched so many operations can be completed with a single lock reference. In addition, iterating across all nodes for per-node operations may provide an opportunity to prefetch locks and data ahead for the next node to be processed, hiding the latency of expensive memory references. This technique may be utilized to improve freeing of slab objects.

Through a combination of improvements to the existing kernel sources, new feature development targeting the reduction of remote cache-to-cache transfers, and improvements in hardware and software capabilities to identify and characterize cache line contention, we can ensure improved scalability on future platforms.

6 Acknowledgements

Special thanks to Chinang Ma, Doug Nelson, Hubert Nueckel, and Peter Wang for measurement and analysis contributions to this paper. We would also like to thank Collin Terrell for draft reviews.

References

[1] *MSI-X ECN Against PCI Conventional 2.3*. `http://www.pcisig.com/specifications/conventional/`.

[2] Intel Corporation. Intel vtune performance analyzer for linux – *dsep and sfdump5*. `http://www.intel.com/cd/software/`

```
samples   remote  module  inst. address       inst.         function                      data address        data

      1  100.00%  sd      0xA00000021E247D80  ld4           sd_init_command               0xE00000012B304880  sector_size

      1    0.00%  qla     0xA00000021E406100  ld8           qla2x00_queuecommand          0xE00000012B304888  hostdata (void*)

     67   50.75%  sd      0xA00000021E2476E6  ld8           sd_init_command               0xE00000012B3048C8  changed
      7   71.43%  scsi    0xA00000021E2A9B90  ld8           scsi_run_queue                0xE00000012B3048C8  changed

     13   46.15%  scsi    0xA00000021E29E260  fetchadd4.rel scsi_dispatch_cmd             0xE00000012B3048D8  iorequest_cnt

    104   27.88%  scsi    0xA00000021E29D2B0  fetchadd4.rel __scsi_done                   0xE00000012B3048DC  iodone_cnt

    129   48.06%  sd      0xA00000021E247566  ld4           sd_init_command               0xE00000012B3048E4  timeout
```

Figure 15: Second cache line of `scsi_device` struct

`products/asmo-na/eng/vtune/239144.htm`.

[3] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manuals*. `http://developer.intel.com/products/processor/manuals/index.htm`.

[4] Intel Corporation. *Intel Itanium 2 Architecture Software Developer's Manuals*. `http://developer.intel.com/design/itanium2/documentation.htm`.

[5] Hewlett-Packard Laboratories. q-tools project. `http://www.hpl.hp.com/research/linux/q-tools/`.

[6] Ingo Molnar. *Syslets, generic asynchronous system call support*. `http://redhat.com/~mingo/syslet-patches/`.

[7] Easwaran Raman, Robert Hundt, and Sandya Mannarswamy. *Structure Layout Optimization for Multithreaded Programs*. In *Proceedings of the 2007 International Symposium on Code Generation and Optimization*, 2007.

7 Legal

Intel and Itanium are registered trademarks of Intel Corporation. Oracle and Oracle Database 10g Release 2 are registered trademarks of Oracle Corporation. Linux is a registered trademark of Linus Torvalds.

All other trademarks mentioned herein are the property of their respective owners.