acmqueue Scalability Techniques for Practical Synchronization Primitives

Designing locking primitives with performance in mind

Davidlohr Bueso, SUSE Labs

In an ideal world, applications are expected to scale automatically when executed on increasingly larger systems. In practice, however, not only does this scaling not occur, but it is common to see performance actually worsen on those larger systems.

While *performance* and *scalability* can be ambiguous terms, they become less so when problems present themselves at the lower end of the software stack. This is simply because the number of factors to consider when evaluating a performance problem decreases. As such, concurrent multithreaded programs such as operating-system kernels, hypervisors, and database engines can pay a high price when misusing hardware resources.

This translates into performance issues for applications executing higher up in the stack. One clear example is the design and implementation of synchronization primitives (locks) for shared memory systems. Locks are a way of allowing multiple threads to execute concurrently, providing safe and correct execution context through mutual exclusion. To achieve serialization, locks typically require hardware support through the use of atomic operations such as CAS (compare-and-swap), fetch-and-add, and arithmetic instructions. While details vary across different cache-coherent architectures, atomic operations will broadcast changes across the memory bus, updating the value of the shared variable for every core, forcing cache-line invalidations and, therefore, more cache-line misses. Software engineers often abuse these primitives, leading to significant performance degradation caused by poor lock granularity or high latency.

Both the correctness and the performance of locks depend on the underlying hardware architecture. That is why scalability and hardware implications are so important in the design of locking algorithms. Unfortunately, these are rare considerations in real-world software.

With the advent of increasingly larger multi- and many-core NUMA (non-uniform memory access) systems, the performance penalties of poor locking implementations become painfully evident. These penalties apply to the actual primitive's implementation, as well as its usage, the latter of which many developers directly control by designing locking schemes for data serialization. After decades of research, this is a well-known fact and has never been truer than today. Despite recent technologies such as lock elision and transactional memory, however, concurrency, parallel programming, and synchronization are still challenging topics for practitioners.¹⁰

Furthermore, because a transactional memory system such as TSX (Transactional Synchronization Extensions) still needs a fallback path with a regular lock when the transaction does not succeed, these challenges are not going to disappear any time soon. In addition, a transactional system does not guarantee specific progress beyond starvation freedom; therefore, this is not always a viable option for regular locking schemes. As a result of the complexities, not only can systems lack in scalability, but also their overall performance can be brought to its knees. Previous work has demonstrated that the cost of a poor, nonscalable, locking implementation grows as the number

of cores in the system increases.² The onset of the performance collapse can, in fact, happen very suddenly, with just a few more cores added to the equation. The issues with performance and scalability go well beyond synthetic workloads and benchmarks, affecting real-world software.

There have recently been significant efforts to address lock-scaling issues in the Linux kernel on large high-end servers.³ Many of the problems and solutions apply to similar system software. This article applies general ideas and lessons learned to a wider systems context, in the hope that it can be helpful to people who are encountering similar scaling problems. Of course, locks are important on any shared memory system, but optimizing them does not imply ignoring the more important aspects: how those locks are used and what is being serialized.

HYBRID LOCKING MODELS AND OPTIMISTIC SPINNING

Locking primitives are traditionally classified as busy-waiting or blocking, depending on what occurs when a lock is not immediately available. If lock hold times are small, such as only serializing a reference count operation, then it makes more sense to avoid all blocking overhead and just burn CPU cycles for a short period of time. This is typically implemented by looping CAS calls until an unlocked condition is met. The alternative is to block until that condition becomes true. Apart from the higher overhead and latency, blocking primitives can commonly have a strict dependency on the operating-system kernel's scheduler. As such, the thread-scheduling policies will increasingly depend on the previous level of execution in the software stack—for example, at hardware, kernel, or multiple user-space layers. Multiple scheduling policies can impact both lock fairness and determinism.

Thrashing is another factor to keep in mind when building sleeping locks. Practitioners must consider the system consequences under heavy lock contention. Popular examples of busy-wait primitives are memory barriers and spinlocks. Blocking mechanisms normally include semaphores and monitors. The Linux kernel, for example, has three primary kinds of sleeping semaphores: mutexes (binary) and two counting semaphores, which includes a widely used reader/writer variant.

The use of a hybrid model is a common way of dealing with the tradeoffs of each lock type. The goal is to delay blocking as much as possible and optimistically busy-wait until the lock is available. Determining when the lock sleeps, however, has an effect on performance. The algorithm needs to perform equally well for workloads that might benefit differently. In other words, users who explicitly want the characteristics of sleeping locks cannot pay performance penalties when a CPU will first spin on a lock for a determined period of time—often, not even at all. In addition, users of general-purpose locking primitives should never be allowed to influence their algorithmic behavior. Misdesigning locking APIs can lead to unexpected consequences later, such as when the lock becomes heavily used. Simplicity is very much a virtue—and a consequence of a well-designed locking interface.

By using the notion of lock ownership, the Linux kernel keeps a pointer to the task that is currently holding the lock. The benefits of knowing the lock owner are twofold: it is a key piece of data when determining when to stop spinning; and it serves a debugging purpose—for example, deadlock detection. Similar strategies date back to 1975, when the notion of lock ownership in databases was first proposed.⁸ Because of the overhead of maintaining lock ownership, implementers can decide against reentrant locks, which also typically use a counter field.¹⁵

The rationale behind optimistic spinning is that if the thread that owns the lock is running, then

it is likely to release the lock soon. In practice, a Linux kernel mutex or rw_semaphore (reader-writer semaphore), the two most commonly used locks throughout the system, can follow up to three possible paths when acquiring the lock, depending on its current state:¹²

- **Fastpath.** It tries to acquire the lock atomically by modifying an internal counter such as `fetch_and_add` or atomic decrementing. This logic is architecture-specific. As shown here, with x86-64, the mutex-locking fastpath has only two instructions for lock and unlock calls:

```
00000000000000e10 <mutex_lock>:
e21:  f0 ff 0b                lock decl (%rbx)
e24:  79 08                   jns     e2e <mutex_lock+0x1e>

00000000000000bc0 <mutex_unlock>:
bc8:  f0 ff 07                lock incl (%rdi)
bcb:  7f 0a                   jg      bd7 <mutex_unlock+0x17>
```

- **Midpath** (aka optimistic spinning). It tries to spin for acquisition while the lock owner is running and there are no other higher-priority tasks ready to run, needing to reschedule. Spinner threads are queued up using an MCS lock so that only one spinner can compete for the lock.

- **Slowpath.** As a last resort, if the lock still cannot be acquired, the task is added to the wait queue and sleeps until the unlock path wakes it up.

Because hybrid locks can still block, these primitives need to be used safely in a sleeping context. Optimistic spinning has proved its value in operating-system kernels such as Linux and Solaris. Even to this day, simply delaying any kind of blocking overhead can have important impacts on system software. On a Linux VFS (Virtual File System) create+unlink microbenchmark to stress mutexes, optimistic spinning gave a boost in throughput of about 3.5 times on a commodity desktop system, as opposed to immediately sleeping. Similarly, for rw_semaphores on AIM7 workloads, hybrid locking provided an increase in throughput of about 1.8 times.³

One notable difference between rw_semaphores and mutexes in the Linux kernel is how they deal with lock ownership. When a lock is shared, the notion of ownership becomes more ambiguous than with exclusive locks. When workloads combine both readers and writers, there is a chance that optimistic spinning can make writers spin excessively since there is no owner to spin on when readers hold the lock. Strategies to overcome this problem exist, such as using heuristics and magic numbers to determine when to stop spinning for readers. Reader ownership can be particularly tricky, however, and tends to add extra complexity and overhead in optimistic-spinning fastpaths. Furthermore, the use of magic numbers in locking primitives can bring unexpected consequences, and they must not be used lightly. By their very nature, heuristics can help in particular scenarios while actually hurting performance in the common cases. Scalability is not about optimizing for the 1 percent and not thinking of the other 99 percent, but quite the opposite. In addition, solving the actual source of contention can be a valid alternative before considering overcomplicated primitives.

CONTRIBUTING FACTORS TO POOR LOCK SCALING

It is not uncommon to find multiple factors contributing to poor lock performance at any given time. Naturally, the impact will vary depending on each system and workload. The factors and lock

properties described here can be divided at a software-engineering level: between implementers, who typically design and implement the locking primitives, and users, who strictly apply them to their parallel or concurrent workloads and algorithms.

- **Length of critical region.** Reducing the length of a critical region can certainly help alleviate lock contention. In addition, the type of primitive used to serialize concurrent threads in lock implementations can play a key role in performance. When holding a lock in a slowpath, such as those that handle contended scenarios when acquiring or releasing a lock, practitioners often need to manage internal wait queues for threads that are waiting to take some action. In these cases, implementers must ensure that the critical regions are short enough to not cause unnecessary internal contention. For example, prechecks or wake-ups (for sleeping primitives) can easily be issued asynchronously, without any additional serialization. Most recently, in the Linux kernel, efforts to shorten critical regions in mutexes and SysV semaphores (as well as other forms of inter-process communication) provided important performance benefits.^{3,13}

- **Lock overhead.** This is the resource cost of using a particular lock in terms of both size and latency. Locks embedded in data structures, for example, will bloat that type. Larger structure sizes mean more CPU cache and memory footprint. Thus, size is an important factor when a structure becomes frequently used throughout the system. Implementers also need to consider lock overhead when enlarging a lock type, after some nontrivial modification; this can lead to performance issues in unexpected places. For example, Linux kernel file-system and memory-management developers must take particular care of the size of VFS struct inode (index node) and struct page, optimizing as much as possible.⁴ These data structures represent, respectively, information about each file on the system and each of the physical page frames. As such, the more files or memory present, the more instances of these structures are handled by the kernel. It is not uncommon to see machines with tens of millions of cached inodes, so increasing the size of the inode by 4 percent is significant. That's enough to go from having a well-balanced workload to not being able to fit the working set of inodes in memory. Implementers must always keep in mind the size of the locking primitives.

As for latency, busy-wait primitives, because of their simplicity, have better latency than more complex locks. Calls to initialize, but particularly to acquire or release the lock, need to be cheap, incurring the fewest CPU cycles. The internal logic that governs the primitive must not be confused with other factors that affect call latency, such as hold times and contention. Blocking (or sleeping) locks are expected to be more expensive, as any algorithm must at least take into account events such as putting threads to sleep and waking them up when the lock becomes available. The tradeoff, of course, is that users choose blocking locks only when dealing with large critical regions or executing in a context that requires sleeping, such as when allocating memory. Thus, if the average time that a thread expects to wait is less than twice the context-switch time, then spinning will actually be faster than blocking.¹⁵ The quality-of-service guarantee is another factor to consider when choosing between spinning and sleeping locks, particularly in realtime systems. Blocking on larger NUMA systems can ultimately starve the system of resources.

In addition, reader-writer primitives may have different latencies when dealing with shared or exclusive paths. This is not an ideal situation, as users will have to consider the extra overhead penalties when, for example, sharing the lock. Making matters worse, users might not even realize this difference, and, thus, sharing the lock will in fact result in poorer performance than using an exclusive lock. The read:write ratio (mentioned later) is an important factor when determining

whether sharing the lock is actually worth the extra cost of using a reader lock. In general, choosing the wrong type of lock can impact performance, incurring unnecessary overhead.

- **Lock granularity.** This refers to the amount of data a lock protects, normally with a tradeoff between complexity and performance. Coarse granularity tends to be simpler, using fewer locks to protect large critical regions. Fine granularity, on the other hand, can improve performance at the cost of more involved locking schemes, particularly beneficial when a lock is contended. When designing concurrent algorithms, coarse-grained locks might be misused only because they can be simpler and, initially, more obvious than the finer-grained alternatives. Because bugs related to synchronization—such as deadlocks, race conditions, and general corruption—can be particularly hard to debug, programmers might prefer them only because of fear and uncertainty, assuming that protecting too much is better than not enough.

Making matters worse, these issues can easily make entire software systems unreliable—thus, practically useless. Even experienced lock practitioners can overlook the potential performance benefits of fine-grained locking, not noticing scaling problems until they are reported. Perhaps the most famous coarse-grained lock in the Linux kernel was the now-replaced BKL (Big Kernel Lock), serializing threads that enter kernel space to service system calls. Because the lock protects so much data, such primitives are also referred to as giant locks. Futexes are another area in the kernel that can suffer from coarse-grained locking schemes. With a chained hash-table architecture, spinlocks protecting the chain can become heavily contended only because of collisions. Finer graining and parallelizing these paths can be done by simply increasing the size of the hash table. This approach improved the hashing throughput by up to a factor of eight.³ This fine-grained technique is also known as lock stripping, improving concurrency by having multiple locks to serialize different, nondependent parts of an array, or list-based data structure.

Coarse-grained locks certainly do have their place, nonetheless. One important factor to consider when addressing coarse-grained locks is the overhead of an individual lock. In some cases, the extra memory footprint of embedding additional locks will overshadow the benefits of alleviating the contention. Abusing fine-grained locking can have just as negative an effect on performance as abusing coarse granularity. Furthermore, coarse granularity can be particularly good when contention is not a problem and critical regions and hold times are short, albeit rare.

Both practice and research have shown the performance benefits of combining coarse and fine granularity, although it's usually more complex. Hybrid strategies have been proposed for locking in the Hurricane kernel, combining the benefits of both kinds of granularity.¹⁶ For more than 20 years, the Linux kernel's SysV semaphore implementation suffered from coarse-grained locking when dealing with `semtimedop(2)` system calls. When a finer-grained locking model was introduced for these calls to handle the common case of a task waiting to manipulate a single semaphore within an array containing multiple semaphores, benchmark throughput improved by more than nine times.¹³ This hybrid strategy also directly impacts important RDBMS (relational database management system) workloads that rely heavily on these semaphores for internal locking, with contention decreasing from approximately 85 percent to 7 percent. Coarse-grained locking in IPC (inter-process communication) can still occur when manipulating more than a single semaphore. Choosing lock granularity must be a well-informed decision. Changing the granularity later in a program life cycle can be an expensive and error-prone task.

- **Read:write ratios.** This is the ratio between the number of read-only critical regions and the

number of regions where the data in question is modified. Reader/writer locks will take advantage of these scenarios by allowing multiple readers to hold the lock while acquiring it exclusively when modifying protected data. Multiple research and development efforts have tried to optimize primitives for read-mostly situations, making the cost of reader synchronization as minimal as possible—normally at a higher cost, or overhead, for writer threads. Examples include variations of the RCU (read-copy update) mechanism, sequence locks (seqlocks), and read-mostly locks (rmlocks) in FreeBSD.

It is well known that the Linux kernel makes heavy use of RCU, allowing lockless readers to coexist with writers. Because readers do not actually hold a lock, it is a particularly fast mechanism that avoids the overhead and hardware implications that regular reader/writer locks incur. RCU handles updates by: (1) making them visible to readers by single-pointer read and write, ensuring that readers execute before or after the modification, depending on whether they see the update in time; and (2) delaying the reclaiming of the data structure until all readers are done with their critical regions, as it is guaranteed that no readers hold references to the data structure, similar to garbage-collection approaches. Introduced in the 2.5 era, in the early 2000s, it is no coincidence that the adoption of RCU within the kernel has grown substantially, including, among many examples, scaling of the dentry (directory entry) cache, NMI (nonmaskable interrupt), and process ID handling.¹¹ Most recently, converting the epoll control interface from using a global mutex to RCU permitted significant performance improvements by allowing file-descriptor addition and removal to occur concurrently. Specifically, important Java-based workloads were boosted with throughput improvements of up to 2.5 times on large HP and SGI NUMA systems.

- **Fairness.** Most importantly, fairness avoids lock starvation by using strict semantics to choose which task is next in line to acquire the lock in contended scenarios. A common example of unfair spinlocks is any thread acquiring the lock without respecting if other threads were already waiting for it. This includes the same task constantly reacquiring the lock. Unfair locks tend to maximize throughput but incur higher call latencies. If such a scenario becomes pathological, it is a real concern for lock starvation and lack of progress—unacceptable in real-world software. A widely used solution to this is different variations of ticket spinlocks. Fair locks address starvation issues at the cost of increased preemption sensitivity,^{1,6} making them at times unsuitable when preemption cannot be controlled, such as in user-space applications. If the kernel's CPU scheduler preempts a task that is next to acquire the lock and the lock is released during this time, then it will cause the rest of the contending threads to wait until the preempted task is rescheduled. Similarly, the greater the cost of lock acquisition or release, the greater the chance of excessive queuing contributing to poor performance.

Experiments conclude that unfair locks are particularly useful when running more than one thread per core, as they can outperform fair alternatives in highly contended scenarios.⁷ Of course, since threads have to wait longer to acquire the lock, this problem can become significantly more pronounced when dealing with NUMA systems, particularly if the fair lock leads to expensive cache-line issues, as described later.

Statistically, it is possible to have different degrees of fairness in NUMA systems. For example, because of CPU node locality, threads can have a better chance of acquiring a lock if it is on the local memory node. By transforming any kind of busy-wait lock into a NUMA-aware primitive, cohort locks were developed to address some of these issues. In this scheme, writer locks are passed among

contending threads within the same NUMA node, while readers maintain shared resources within the same node.

Fairness takes a different turn when dealing with read/write locks, depending on the context, workload, and reader:writer ratios. There will be occasions where it is more suitable to give preference to readers, and vice versa. Either way, practitioners should take special care that the primitive does not starve reader or writer threads to the point that it makes the lock perform poorly in some use cases. One alternative is implementing different variations of the same read/write primitive with particular fairness preferences, but this can also lead to developer misuse in different contexts. For performance reasons, the Linux kernel uses the concept of writer-lock stealing for `rw_semaphores`, breaking the strict FIFO (first-in first-out) fairness for writers. Writers can atomically acquire the lock, even if other writers are already queued up.

- **Cache-line mishandling.** Cache-line bouncing and contention are probably the two worst forms of performance degradations on large NUMA systems when it comes to low-level locking primitives. Tasks spinning on a contended lock will try to fetch the lock cache line repeatedly in some form of tight CAS loop. For every iteration, usually in an atomic context, the line containing the lock is moved from one CPU cache to another. It is easy to see how this bouncing will degrade performance with increasing CPU counts, incurring expensive memory-bus and interconnect usage penalties. Furthermore, if the lock-protected data structure is in the same cache line, it can significantly slow down the progress of the lock holder, leading to much longer lock hold times.

A straightforward way of addressing cache-line bouncing was proposed 30 years ago¹⁴ with the CCAS (compare compare-and-swap) technique. The idea is to do a simple read of the lock state and incur the read-modify-write CAS operation only when the lock becomes available. The Linux kernel now relies on CCAS techniques for the internal counter checks of both mutexes and read/write semaphores when attempting to acquire a lock exclusively (note that sharing the lock does not require such CAS storms). Concretely, for mutexes, Java-based benchmarks experienced up to a 90 percent throughput increase³ on a 16-socket, 240-core system. AIM7 benchmark workloads that are designed to execute mostly in kernel space also saw noticeable improvements on eight-node, 80-core Westmere systems with throughput increases of up to three times. As for read/write semaphores, CCAS results from `pgbench` on smaller, quad-core desktop systems showed improvements of up to 40 percent on 1-GB PostgreSQL databases. Most noticeably improving the cache-line bouncing that occurs when the `mmap_sem` is heavily contended, this lock is designed, among other things, to serialize concurrent address-space operations.

In general, experimentation shows that CCAS techniques will help on large high-end systems with four or more sockets, or NUMA nodes. Normally, the overhead of checking the counter is so low in noncontended cases that it will not impact performance negatively on smaller systems. Performance work must always ensure that no regressions are introduced in lower-end machines, particularly in the Linux kernel, which spans an enormous user base.

Alternatively, the use of backoff algorithms, which address expensive CAS operations when spinning on a lock, can help alleviate cache-line bouncing and memory-interconnect overhead. As with CCAS techniques, the idea is to delay read-modify-write calls in tight spinning loops, the main difference being the delay factor when the lock is not immediately available. Significant research has attempted to estimate the optimal delay factor across multiple systems and workloads; the different algorithms and heuristics can be classified as static and dynamic. Delaying for a static amount of

time can be optimized for a particular workload but, of course, won't necessarily perform well when generic locking solutions are needed. To this end, dynamic delays are far more realistic, but at the cost of extra overhead in the heuristics and the risk of miscalculating the backoff strategy: threads should not back off for too long, as this can eliminate any performance benefits that these strategies try to introduce.

Good heuristics for backoff algorithms are based on the number of CPUs trying to acquire the lock, such as proportional and exponential, with analogies drawn from CSMA (carrier sense multiple access) networks such as Ethernet. Delaying for the optimal time requires estimating the length of the critical region and the latency of the lock holder to release the lock so another thread can acquire it.^{6,15} Needless to say, this sort of information is rarely available for real-world workloads. In addition, backoff algorithms do not address the fact that, just as with regular CAS and CCAS spinlocks, all threads spin on the same shared location, causing cache-coherence traffic on every successful lock acquisition. With these caveats, it is not surprising that backoff algorithms are not suitable for the Linux kernel, even though promising results have been seen with ticket spinlocks and proportional backoff strategies.⁵

DEMONSTRATING THE EFFECTS OF CACHE-LINE CONTENTION

The sole purpose of locking and concurrency is improving system performance by allowing correct parallel executions of threads. Even on uniprocessor systems, concurrency permits the illusion of multitasking in preemptive scheduling environments. If locking primitives get in the way of performance, then they are simply broken. Of course, unlike correctness attributes, locks that perform poorly will matter more on larger systems.

In a modern large-scale, multicore NUMA system, the effects of cache-line contention can go from incurring moderately acceptable overhead to being the culprit for severe system issues. Table 1 shows three high-end HP x86-64 servers used as test systems.³ All these machines follow a normal NUMA topology in which cores and memory resources are evenly distributed among nodes. There are, therefore, no exotic NUMA use cases.⁹ Each socket, or node, has 15 cores, and memory is equally divided as well. A simple spinlock microbenchmark is enough to measure the cost of cache-line contention in large NUMA systems—in this case, using a Linux 3.0-based spinlock implementation, iterating a million times in a tight loop just acquiring and releasing the lock, similar to what a torturing program such as rcutorture would do.

While this pathological behavior doesn't fall into the real-world category, it nicely exemplifies theoretical issues such as lock contention. Furthermore, real software with very similar behavior does exist out there, leading to horrendous contention. Based on the number of threads and the loop iteration count, the average number of operations per second per CPU can be calculated when N CPUs are involved in the cache-line contention. This serves as the benchmark throughput.

TABLE 1 Testing three high-end HP x86-64 servers

System	Sockets/Cores	Memory
HP DL580 Gen8 (Intel Xeon E7 - 4890 v2)	4 sockets/60 cores	2 TB
HP CS900 (Intel Xeon E7 - 4890 v2)	8 sockets/120 cores	6 TB
HP CS900 (Intel Xeon E7 - 4890 v2)	16 sockets/240 cores	12 TB

FIGURE 1

Effects of Cache-Line Contention within a Single Socket

For a full understanding of the effects of cache-line contention, the examination must include a single socket. Otherwise, factors such as NUMA awareness and interconnect cost couldn't be differentiated from the results. Furthermore, it should provide the least performance impact and therefore can provide a level ground to compare against other results that involve multisocket communication. Figure 1 shows the microbenchmark throughput regression by increasing core counts within a single socket.

It's easy to see how performance degrades smoothly as more cores contribute to cache-line contention. By maximizing resources and using all cores in a single socket (that is, by a factor of seven), the microbenchmark throughput decreases by 91.8 percent when compared with using only two cores. The situation ends up, therefore, being the exact opposite of what naive developers will normally intuit, expecting performance at least to improve "some," by blindly throwing in more cores. Furthermore, the 91.8 percent decrease can be considered a maximum performance penalty when evaluating systems that will logically partition their software architecture around NUMA topologies. This is commonly done to avoid latency penalties when dealing with remote memory, particularly on large-end machines.

Of course, the situation doesn't get any better. In fact, it gets a great deal worse. Table 2 compares the costs of cache-line contention in a single socket versus two sockets. Most noticeably, there's more than an 80 percent decrease in throughput when going from 15 to 30 cores.

Furthermore, the cost of having the lock's memory location on a remote node is clearly worse

TABLE 2 Comparison of cache-line contention costs in a single socket with two sockets

Execution Nodes	Memory Node	Sockets Used	Cores Used	ops/sec/core
Node 0	Node 1	1 socket	15 cores	474,519
-	-	-	-	-
Nodes 0-1	Node 1	2 sockets	30 cores	69,200
Nodes 1-2	Node 1	2 sockets	30 cores	67,126
Nodes 2-3	Node 1	2 sockets	30 cores	45,993

when using two sockets, reducing throughput by more than 90 percent when compared with a single socket. In this case, benchmark runtimes are even more concerning, going from two to 21 seconds to complete the million iterations when the memory is completely remote.

The number of cores is not the only factor to consider when measuring this sort of contention. The distribution method plays a key role in performance: cache-line contention is always better when all the contention is contained in as few sockets as possible.³ As core counts increase, the FF (first fill) method will always try to use the cores available in the same socket, while an RR (round robin) distribution will use cores from increasing socket counts.

Figure 2 exhibits the probability of inter- versus intra-cache-line contention, estimated by

computing $100/\text{num_sockets}$. Increasingly large multisocket systems will thus have a much lower chance that two random cores participating in the same cache-line contention will reside on the same socket.

This can have catastrophic effects on performance and demonstrates that contention in a two-socket system is extremely different from that in a 16-socket system. As such, applications tuned for smaller systems cannot be blindly executed on larger systems and immediately expect good results without previous careful thought and analysis. As previously seen, scaling based only on the number of CPUs is likely to introduce significant lock and cache-line contention inside the Linux kernel.

QUEUED/MCS: SCALABLE LOCKING

As already discussed, even busy-wait locks with CCAS or backoff strategies can incur enough cache-line contention to impact overall system performance significantly on large machines. As such, their inherently unfair nature exposes them to an irregular number of cache misses when the lock becomes contended. On the other hand, a truly scalable lock is one that generates a constant number of cache misses, or remote accesses, per acquisition,² thus avoiding the sudden performance collapse that occurs with nonscalable alternatives. Queued locks achieve this by ensuring that each contending thread spins for a lock on the CPU's local memory, rather than the lock word. As a consequence of this determinism, queued locks are fair, commonly granting lock ownership to waiting threads in FIFO order. Another attractive property of these locks is the overhead, requiring only $O(n+j)$ space for n threads and j locks,¹⁵ as opposed to $O(nj)$ for nonscalable primitives. Most forms of queued locks—MCS, K42, and CLH, to name a few popular ones—maintain a queue of waiters, each spinning on its own queue entry. The differences in these locks are how the queue is maintained and the changes necessary for the lock's interfaces.

A common experiment is to replace a regular spinlock with an MCS lock,^{2,3} always with quite superb results in kernel space (normally using the Linux kernel). Figure 3 compares a vanilla 2.6 Linux spinlock with a prototype MCS implementation on an AIM7 file-system benchmark, serializing concurrent linked-lists operations with a single lock. Contention on this global spinlock is the cause of all the cache-line contention.

As more users are added to the workload, throughput (jobs per minute) in the spinlock case quickly flatlines, while with the MCS lock it steadily improves by a factor of 2.4. Similarly, system time drops from 54 percent to a mere 2 percent; thus, the bottleneck is entirely removed. As no additional optimization efforts are made, such as finer-graining the lock, it is safe to conclude that the MCS lock alone makes a huge impact by minimizing only the intersocket cache-line traffic.

To this end, MCS locks have been added to the Linux kernel,² particularly in the spinning logic of sleeping semaphores. In the case of mutexes, this provides a throughput boost of around 248 percent in a popular Java workload on a 16-socket, 240-core HP Converged-System 900 for SAP HANA. The following code shows the primitive's interfaces:

```
struct mcs_spinlock {
    struct mcs_spinlock *next;
    int locked;
};
```

FIGURE 3

Eight-Socket/80-Core HT-Enabled Xeon

```
void mcs_spin_lock(struct mcs_spinlock **lock, struct mcs_spinlock *node);
void mcs_spin_unlock(struct mcs_spinlock **lock, struct mcs_spinlock *node);
```

The idea is that when a thread needs to acquire the contended lock, it will pass its own local node as an argument, add itself to the queue in a wait-free manner, and spin on its own cache line (specifically on `node->locked`) until its turn comes to acquire the lock. The FIFO fairness comes naturally as the current lock holder will pass the lock along to the next CPU in the linked list when it releases the lock. One drawback of this algorithm is that, in order to add itself to the wait queue, a task must store the pointer of its local node in the `->next` field of its predecessor, potentially incurring extra overhead if the predecessor task needs to release the lock.

Unfortunately, using MCS locks in regular ticket spinlocks cannot be done without enlarging the size of the lock. Spinlocks are used throughout the kernel and cannot be larger than a 32-bit word size. To this end, other queued locking approaches¹⁷ offer valid alternatives to the current ticketing system.

CONCLUSIONS

Designing locking primitives with performance in mind is not only good practice in general, but also can mitigate problems in the future. This article was born from experience with larger-scale servers and on real issues that impacted real clients using Linux. While Donald E. Knuth stated that

“premature optimization is the root of all evil,” it is quite the opposite when implementing locking primitives. Empirical data and experiments have shown the cost of misusing locks or ignoring the underlying hardware implications of locking. In addition, both users and implementers must take special care in how these locks are used and the consequences of particular decisions during the life cycle of a lock, such as different degrees of contention.

In the future, as computer systems evolve and increase their processing capabilities, the practice and theory of locking primitives will need to adapt accordingly, ultimately making better use of the hardware architecture. Even today, very specialized locks are available, such as cohort and hierarchical primitives like HCLH that optimize for memory locality, addressing the fairness issues of backoff-based locks, while keeping the benefits of queued-based locks.

Of course, there is no single recipe for locking performance and scalability, and there are many more related topics that the reader can follow up on, including lockless data structures and addressing particular constraints for specialized systems such as priority inversion and inheritance for realtime environments. An article of this nature should, at least, serve as a way of creating awareness among practitioners when dealing with severe scaling problems on large systems.

ACKNOWLEDGMENTS

The scaling work in the Linux kernel has been a team effort, both internally in the Hewlett-Packard Linux performance group and the upstream kernel community for invaluable feedback and entertaining discussions. Many thanks to Scott J. Norton. Without his analysis and empirical data, it would have been very hard to realize the deep architectural implications of poorly implemented primitives. I am in debt to Paul E. Mckenney for his support and Samy Al Bahra for careful review and suggestions, leading to a better article. Thanks to Jim Maurer and the rest of the *ACM Queue* team for their support and feedback.

Legal Statement

This work represents the views of the author and does not necessarily represent the view of SUSE LLC. Linux is a registered trademark of Linus Torvalds. Other company, product, and service names may be trademarks or service marks of others.

REFERENCES

1. Al Bahra, S. 2013. Nonblocking algorithms and scalable multicore programming. *ACM Queue* 11(5).
2. Boyd-Wickizer, S. Kaashoek, F. M., Morris, R. Zeldovich, N. 2012. Non-scalable locks are dangerous. *Proceedings of the Linux Symposium*. Ottawa, Canada.
3. Bueso, D., Norton, S. J. 2014. An overview of kernel lock improvements. LinuxCon North America, Chicago, IL; <http://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf>.
4. Corbet, J. 2013. Cramming more into struct page. LWN.net; <http://lwn.net/Articles/565097/>.
5. Corbet, J. 2013. Improving ticket spinlocks. LWN.net; <http://lwn.net/Articles/531254/>.
6. Crummey-Mellor, J. M., Scott, M. L. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions of Computer Systems* 9(1): 21-65.
7. Fuerst, S. 2014. Unfairness and locking. Lockless Inc.; <http://locklessinc.com/articles/unfairness/>.
8. Gray, J. N., Lorie, R. A., Putzolu, G. R., Traiger, I. L. 1975. Granularity of locks and degrees of

- consistency in a shared data base. San Jose, CA: IBM Research Laboratory.
9. Lameter, C. 2014. Normal and exotic use cases for NUMA features. Linux Foundation Collaboration Summit, Napa, CA.
 10. McKenney, P. E. 2014. Is parallel programming hard, and, if so, what can you do about it?; <https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html>.
 11. McKenney, P. E., Boyd-Wickizer, S., Walpole, J. 2013. RCU usage in the Linux kernel: one decade later; <http://www2.rdrop.com/users/paulmck/techreports/RCUUsage.2013.02.24a.pdf>.
 12. Molnar, I. Bueso, D. 2014. Design of the generic mutex subsystem. Linux kernel source code: documentation/mutex-design.txt.
 13. van Riel, R., Bueso, D. 2013. ipc,sem: sysv semaphore scalability. LWN.net; <http://lwn.net/Articles/543659/>.
 14. Rudolph, L., Segall, Z. 1984. Dynamic decentralized cache schemes for MIMD parallel processors. *Proceedings of the 11th Annual International Symposium on Computer Architecture*: 340-347.
 15. Scott, M. L. 2013. *Shared-memory synchronization. Synthesis Lectures on Computer Architecture*. San Rafael, CA: Morgan & Claypool Publishers.
 16. Unrau, R. C. Krieger, O. Gamsa, B., Stumm, M. 1994. Experiences with locking in a NUMA multiprocessor operating system kernel. Symposium on Operating Systems Design and Implementation; https://www.usenix.org/legacy/publications/library/proceedings/osdi/full_papers/unrau.a.
 17. Zijlstra, P., Long, W. 2014. locking: qspinlock. LWN.net; <http://lwn.net/Articles/590189/>.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

DAVIDLOHR BUESO is a performance engineer at SUSE Labs. He is an active Linux kernel contributor in areas such as synchronization primitives, memory management, and IPC—all with a special focus on scalability. His interest in performance goes back to graduate school where he explored ways of improving hypervisor memory management for his M.Sc. thesis. More recently, as a software engineer at Hewlett-Packard, he participated in important Linux kernel scaling efforts on large high-end x86-64 systems.

© 2014 ACM 1542-7730/14/1100 \$10.00