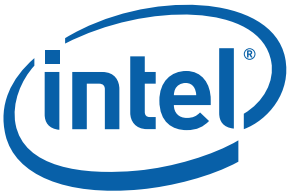


Lock Scaling Analysis on Intel® Xeon® Processors

April 2013

Reference Number: 328878-001, Revision: 1.01

Contents

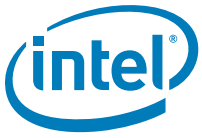
Figures

Tables

Revision History

Document Number	Revision Number	Description	Date
328878-001	1.01	• Initial Release	April 2013

§



CHAPTER 1
LOCK SCALING ANALYSIS ON INTEL® XEON® PROCESSORS

1.1 INTRODUCTION

This white paper discusses the performance of locking algorithms and how they scale under contention on Intel® Xeon® processors. We compare the performance of contended locks on the Intel® Xeon® processor X5600 series and the Intel® Xeon® processor E5-2600 series under different conditions, and conclude with a list of recommendations for better lock scalability.

Table 1-1. Configurations[1]

	System 1 (Codenamed Westmere)	System 2 (Codenamed Romley)
Processor	Intel® Xeon® Processor X5680, 12MB L3 cache, 3.3GHz, 6.4GT/s Intel® QuickPath Interconnect, 6 cores per processor	Intel® Xeon® Processor E5-2680, 20MB cache, 2.7Ghz, 8.0GT/s Intel QuickPath Interconnect, 8 cores per processor
Platform	12 cores enabled running at 2.66GHz with Intel® Turbo Boost Technology disabled, frequency scaling disabled, Intel® Hyper-Threading Technology disabled, 18GB of DDR3 1067Mhz 1GB DIMMs	12 out of 16 cores enabled running at 2.7GHz with Intel® Turbo Boost Technology disabled, frequency scaling disabled, Intel® Hyper-Threading Technology disabled, 32GB of RAM in 16 2GB DIMMs, 1333Mhz Samsung M393B5273CH0-YH9
Software	Red Hat* Enterprise Linux* 6.0 (no updates) with updated Linux* 3.6.0 kernel	Fedora* FC16 updated with Linux 3.6.0 kernel

Notes:

1.For more information go to http://www.intel.com/performance

1.2 LOCK SCALING ANALYSIS

Performance of a lock algorithm is typically measured by its throughput, that is, the number of locks the algorithm is able to complete successfully in a fixed window of time[1]. The latency is also a useful metric, but because the cumulative time it takes to complete lock processing is proportional to the inverse of the algorithm's throughput, we will focus on throughput as our metric. The other important metric for performance is how the throughput of the algorithm varies with the number of cores participating in the algorithm. We will use the term lock scaling to refer to the algorithm's sensitivity to core count.

1. Throughput here refers to that of the overall algorithm and does not distinguish the rate of each individual thread's lock acquire throughput. This has subtle but key implications for how this throughput number is interpreted, especially when dealing with micro-benchmark measurements.

Lock algorithms have been an active area of work over the past decades. A few of the key algorithms used in software today are summarized in Appendix A. A full analysis of the performance and scaling characteristics of each is beyond the scope of this paper. In this white paper, we focus on locking algorithms that have a “polling” component in their lock acquire phase. Such algorithms are most commonly used in software today. An example is a spinlock, where the algorithm repeatedly tests to see if the lock is available prior to its attempt to acquire the lock. We use the spinlock algorithm's scaling as a proxy for algorithms that have a polling phase (typically referred to as the “test” phase of a lock algorithm), where threads periodically poll a shared address in memory to determine the status of a resource[1].

When a lock that has such a polling phase is heavily contended, the factor which most impacts scaling is the time it takes to sift through the polling threads when the lock address is being written. This is the primary component in the lock latency that degrades as the number of cores simultaneously polling the address increases. The poor scaling for an algorithm that polls a shared address in this way is an expected result of the polling.

In our lock scaling proxy, the spinlock, each thread polls the address to determine when the lock is free. Another traditional lock example, the ticket lock, has each thread poll the address to determine when its turn to acquire the lock has arrived. Though there are definite performance differences between the two, the general degradation in the time spent polling as core count increases is common to both. For this reason, we believe that the analysis that follows is generally applicable to any algorithm that includes a component where multiple threads are polling a shared address.

One additional point that we aim to convey is that understanding how a given algorithm behaves on a given system microarchitecture is not as simple as specifying the general type of lock algorithm and then measuring its behavior. It also depends on how the algorithm is implemented in software. In particular, it depends on how the software's latency choices interact with underlying hardware latencies.

1.2.1 Factors to consider in lock scaling

1.2.1.1 Importance of software latency assumptions

In the analysis that follows, we will show results for a spread of latencies for two of the key latencies relevant to most lock algorithms:

1. The time that each thread spends in the critical section (critical section time). This latency represents how much work is done by a thread each time it acquires the lock.
2. The time each thread waits, after leaving the critical section, before attempting to enter the critical section again (re-entry time). This latency represents how much work, that is not relevant to the critical shared resources, is being done by each thread, while it does not own the critical section resource.

These two latencies are determined by how the software has chosen to partition and isolate its shared resources and can vary wildly in real applications, even when those applications are using the same general type of algorithm.

A micro-benchmark that aims to study lock performance cannot give a reasonable account of how a given lock algorithm will behave on real software, if the study of the micro-benchmark does not include a study of the sensitivity to these two work intervals and does not factor in whether the spread of latencies it has evaluated are representative of the application of interest.

1. This is in contrast to other types of algorithms that do not involve polling, such as sleeping locks.

Expected behavior for a given algorithm is usually predictable at relatively long latencies for the units of time represented by these work intervals, but the same is not true when the work intervals are very short. It gets increasingly difficult to predict behavior as the software latencies are decreased, particularly when the unit of time represented by these work intervals begins to approach the hardware's underlying cache-to-cache transfer latencies. If a lock becomes heavily contended with such short intervals, and the software tries to force lock transfers to occur at a rate that is faster than the rate at which the hardware can perform a single cache-to-cache transfer, then behavior becomes non-deterministic.

1.2.1.2 Importance of thread count

When evaluating lock performance across platforms, it is important to bear in mind that there is a general trend of increasing the number of cores in the socket with each generation. Contention between threads will go up as the number of threads that can simultaneously compete is increased. When evaluated for expected latency, most lock algorithms, even those not explicitly covered here, will reveal some period in the overall time we expect for the algorithm to achieve an ownership transfer, where forward progress to the next lock owner of the lock address is impeded by the hardware's need to arbitrate between the competing threads. That arbitration is going to take longer, as threads running on the additional cores are added. If software persists in operating in modes where all the threads that hardware makes available are simultaneously attempting to use the same hardware resources, the observed scaling will degrade. This is inevitable. It is going to become increasingly important, going forward, that software account for this in its algorithms.

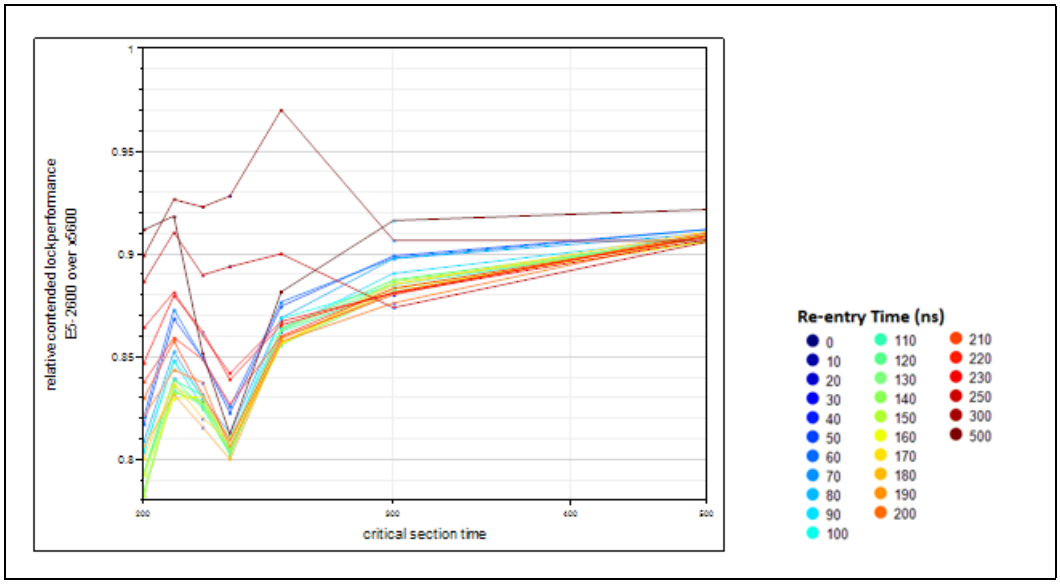
1.2.2 Lock scaling results

We now analyze the scaling performance of our proxy spinlock algorithm. We focus primarily on the difference between the Intel Xeon processor X5600 series and the Intel Xeon processor E5-2600 series. For a controlled experiment, we keep the same thread count (6 hardware threads per socket) across the two systems and use the same clock frequency for the processors on the two systems. In a normal platform-to-platform comparison, the processors will not be operating at the same frequency, and the idle latency differences observed will be different.

The relative throughput of the lock algorithm on the Intel Xeon processor E5-2600 series over the Intel Xeon processor X5600 series is plotted on the y-axis in the graphs shown. The x-axis shows the critical section time on a logarithmic scale, varying from 10ns to 500ns. Each curve corresponds to a different re-entry time (varying from 10 ns to 500 ns).

1.2.2.1 Lock scaling with >200ns critical section times

Figure 1-1. Single-socket contended spinlock performance (the Intel® Xeon® processor E5-2600 series over the Intel® Xeon® processor X5600 series) at the same-frequency (2.7GHz) and same thread count (6 threads participating) for critical section times from 200ns to 500ns

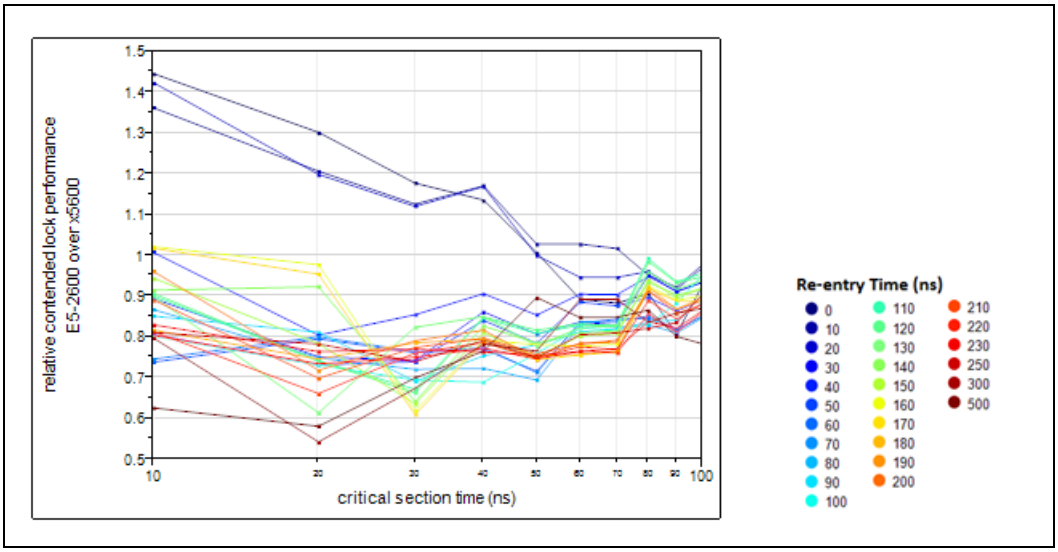


The graph shows two distinct regions. When critical section times exceed 300 ns (right side of the graph), the Intel Xeon processor E5-2600 series transfers spinlock ownership at a slower rate (in general about 8-12% slower) than the Intel Xeon processor X5600 series for all re-entry times. The variation is sharper (up to 18%) when the critical section time falls in the 200 ns to 300 ns interval. Evidence of volatility in the behavior on both processors being compared is also beginning to show itself at the lower end of the critical section time range, as some of the curves begin to diverge from the general trend.

At higher critical section and re-entry times, the locks are lightly contended. In this scenario, the performance difference observed is due to the differences in the idle cache-to-cache transfer latencies. The Intel Xeon processor E5-2600 series implements a ring architecture to support more cores, which results in slightly longer idle cache-to-cache transfer latencies (by about 6 ns at 2.7GHz frequency when both cores are in the same socket). The 8-12% delta, on the right end of the curve, tracks roughly what one would expect, based on those idle latency differences.

1.2.2.2 Lock scaling with <200ns critical section times

Figure 1-2. Single-socket contended spinlock performance (the Intel® Xeon® processor E5-2600 series over the Intel® Xeon® processor X5600 series) at the same-frequency (2.7GHz) and same thread count (6 threads participating) for critical section times from 10ns to 100ns



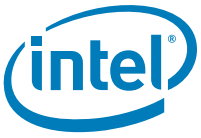
We now analyze spinlock scaling when the critical section time is less than 200 ns.

As we can see in the graph, there is no clear trend in locking algorithm performance at lower critical section times, though the results converge better in the 50-100ns range. Several factors contribute to the low performance numbers being reported by the micro-benchmark test, and we focus on two key factors in this discussion.

The first factor is due to changes in how the Intel Xeon processor E5-2600 series processes snoop invalidations compared to the Intel Xeon processor x5600 series. On Intel Xeon processor E5-2600 series, the snoop invalidations arrive faster than on the Intel Xeon processor X5600 series. This occurs due to enhancements in the Intel Xeon processor E5-2600 series microarchitecture, where the inter-core ownership transfers can occur sooner than on the Intel Xeon processor X5600 series.

The impact of this change shows up in the extremely contended case where a thread on an Intel Xeon processor X5600 series may re-acquire the lock it just released, and perform multiple updates before any other thread gets an opportunity to successfully acquire the lock. In contrast, on an Intel Xeon processor ES-2600 series-based system, because of the faster snoop invalidations, other threads get a better chance of acquiring the lock before the releasing thread re-acquires the lock. This causes a perceived reduction in throughput when the Intel Xeon processor ES 2600 series is compared to the Intel Xeon processor X5600 series. However, this is misleading because benchmarking a thread's ability to quickly pass lock ownership back to itself in the face of contention with other threads is not likely to be representative of realistic application behavior. This is particularly true because in general the performance of workloads that share data is improved when cache-to-cache transfer latency is reduced.

The second factor is due to other micro-architecturally unique features on both the Intel Xeon processor E5-2600 series and the Intel Xeon processor X5600 series that cause non-deterministic performance behavior when these intervals are very low. This non-determinism shows itself, in the above graph, where the lines that represent different re-entry times cross each other as critical section time decreases.



1.2.2.3 Applicability of the analysis

In general, this analysis should apply to any locking algorithm that involves multiple threads polling a shared address waiting for a particular value to be written to that address. When such a lock algorithm takes the work interval close to zero, one can expect performance degradation and unpredictability.

When predictable performance is desired, such a lock algorithm needs reasonably long critical section and re-entry times. In a software implementation that takes either of those down to very small values, results become unpredictable and a tightly contended lock is guaranteed not to scale properly.[1]

1.3 LOCK COARSENING

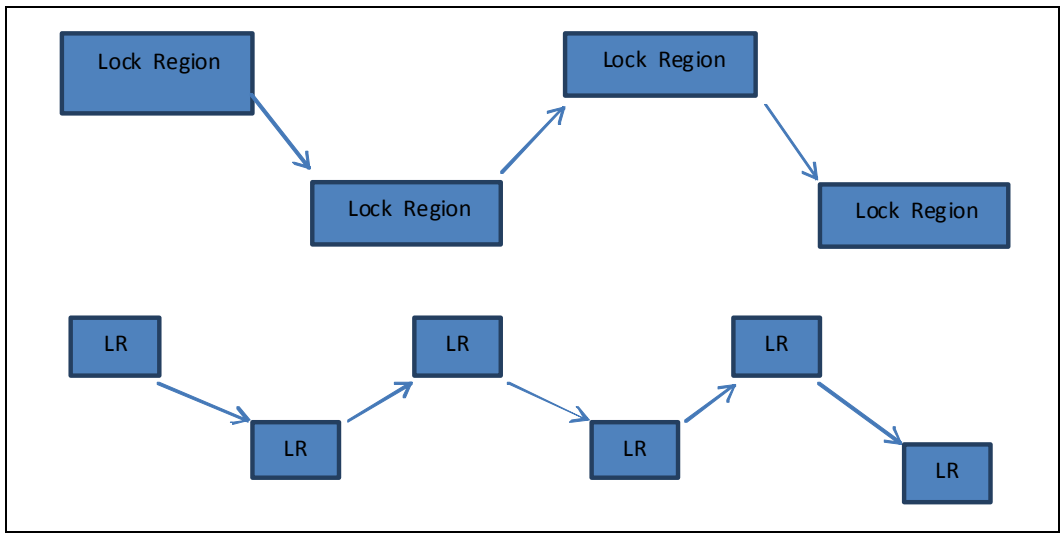
1.3.1 Lock granularity

Traditionally, lock optimization has tried to reduce the length of lock regions to minimize lock contention. This is done to minimize the blocking time for other threads waiting on the lock. For optimized code that already has short lock regions, the time to transfer the cache line of the lock between threads, when unlocking and relocking, can start to dominate when there is even moderate contention on any of the locks. This is more visible as the system gets larger.

The following diagram compares the effects of a contended longer lock region with a short lock region. With the short lock region, more time is spent communicating, that is, transferring the lock cache line between cores. The longer lock region has potentially longer block times, but minimizes the communication overhead of transferring the lock too frequently. This is a simple case with only two threads. With more cores in each socket, the queueing delays of transferring will increase. Similarly, on larger systems that have more sockets, the communication delays will also increase. Having too many small lock regions increases communication overhead, while having too many large lock regions increases blocking time.

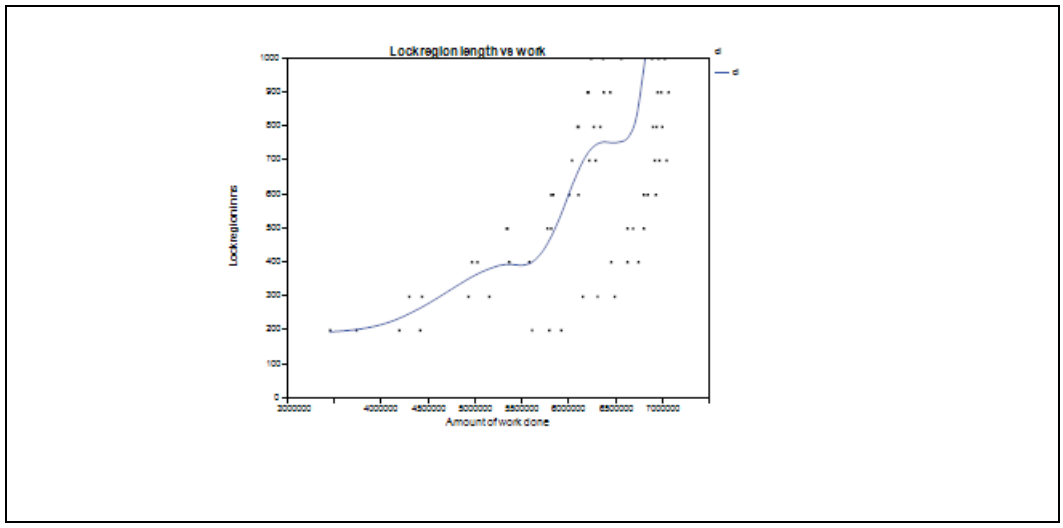
1. An additional point is that when these software latencies are too short, results may sometimes show what appears to be stellar "average" scaling by allowing a few threads to monopolize the resource, while starving out others. The latter can be observed, for example, if one uses, as one's metric for lock performance, a micro-benchmark that has every thread cycle through atomic operations to the same address as fast as possible with no regard for what value is being written or read, and with no regard for critical section time or re-entry time. Such a micro-benchmark is not very informative.

Figure 1-3. Comparing the effects of a contended longer lock region with a short lock region



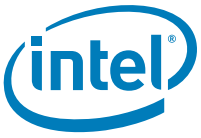
Consider a program that does a variable number of constant-size operations inside the lock. In a test program, we do hashes. Multiple threads lock each other out. Each thread can do work only when the other threads are waiting on the lock. Now, we vary the number of hashes inside the lock region and measure the total work done. For this test, increasing the length of the lock region improves the total amount of work. We get the best total result for a long lock region.

Figure 1-4. Lock region length vs. work



There is clearly a trade-off. Making the lock regions too large will increase blocked time again. But making them too small increases the communication overhead too much. Batching locks is a good idea. The best region size depends on the workload, but it is neither very small nor extremely large. We need to somehow find the critical section size that is “just right”.

Programs that are already highly optimized for scaling with fine grained locking often have very short lock regions. This can give good performance, but when there is a situ-



ation where the lock contention increases on one of these very short lock regions, the performance may suffer dramatically. One example of this would be a hash table with per hash bucket locking and a short critical region accessing the data in the hash table under the lock. Normally, because the hash values are well distributed, there is no contention on the bucket lock and the system will scale well. However, when there is a workload that has hash collisions on hot entries, the lock contention may go up dramatically on the bucket with the collision and perform poorly because the critical region is small. In this case, fine-grained locking can be a liability.

A common situation where this can happen is packet processing when each packet received is processed individually and takes its own read locks to look up the global state. Batching the locks and processing multiple packets (or larger packets, for example, as generated by Large Receive Offloading (LRO) of the NIC) per lock acquisition, can improve scaling by increasing the size of the critical region and lowering the number of local acquisitions. This is called “batching the lock” or “lock coarsening”. Similar techniques can be applied to other queue processing code patterns.

1.3.2 Fairness and unfairness

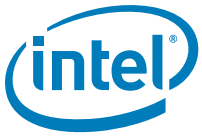
As shown earlier, though a spinlock algorithm cannot ensure fairness, the amount of unfairness observed by a spinlock can vary with the microarchitecture. A microarchitecture that is less fair will give natural batching to lock regions that are too short, because unfair lock acquisition will tend to reacquire the lock on the same thread multiple times in a row. This can improve performance with lock regions that are too short by minimizing communication costs (as shown above), but at the cost of fairness. As long as the unfairness does not lead to starvation, this may be acceptable, depending on the workload requirements. Microarchitectures that are more fair will not do this natural batching and will require longer critical regions to perform well. Essentially, the programmer has to do the batching.

With ticket locks (and, to a lesser degree, sleeping locks), fairness is guaranteed by the lock. This means no natural batching occurs and the programmer may also need to increase lock region sizes manually for good performance.

1.4 LOCK SCALING ON FUTURE INTEL PROCESSORS

We expect processors coming after the Intel Xeon processor E5-2600 series to follow similar trends, as far as lock performance is concerned. Increased core count is likely to lead to small increases in the idle cache-to-cache transfer latencies, which will slightly increase the time it takes to move ownership of a lock address among cores on the package. This may serve to further highlight the importance of avoiding the non-linear scaling effects that can occur when a lock is highly contended, and encourage software to avoid very short critical section and re entry times. But, we do not expect any new scaling trends to develop with future processors, aside from those which are already evident in the Intel Xeon processor generations that are currently available.

The next generation of Intel processors will support lock elision using Intel® Transactional Synchronization Extension (Intel® TSX). Intel TSX can improve lock scaling in many cases, in particular with coarse-grained locks, but may need additional tuning.



1.5 RECOMMENDATIONS

As mentioned previously in this paper, the performance of a lock algorithm under contention depends on the actual software implementation as well as how the software's latency characteristics interact with underlying hardware latencies.

Several metrics play a critical role in determining the scalability of a locking algorithm:

1. The time each thread spends in the critical section holding the lock
2. The time each thread waits after releasing the lock and before trying to re-acquire the lock
3. The number of cores actively contending for the lock
4. The number of sockets in the system, over which those cores are distributed

When values for #1 and #2 are very low (on the order of less than a few hundred nanoseconds), the behavior can be unpredictable because such low latencies are approaching the hardware's idle cache-to-cache transfer latencies. Even at a fixed thread count, algorithms with low values for these metrics may not scale well across processor generations, and may even scale poorly with frequency across processors from the same processor generation.

That #3 and #4 are also a factor should be less surprising. As the number of threads polling the status of a lock address increases, the time it takes to process those polling requests will increase. Finally, the latency to transfer data across socket boundaries will always be an order of magnitude longer than the on-chip cache-to-cache transfer latencies. Such cross-socket transfers, if they are not effectively minimized by software, will negatively impact the performance of any lock algorithm that depends on them.

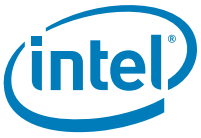
There are a few techniques that can be adopted to improve lock scalability:

1. Increase the amount of work done while holding the lock. If the time that the lock is held is more than a few hundred nanoseconds, then the time to transfer the lock between different threads does not negatively impact the scalability of the lock algorithm. While good software practice may recommend fine-grained locks, the benefits of fine-grained locking are reduced if the lock is highly contended. As a result, there is a tension between fine-grained locking and achieving scalable lock performance. Batching locks and doing more work, per lock, will improve the scalability if high contention is anticipated.
2. Partition resources on a per-socket basis. Because it is more expensive to transfer locks across sockets than among cores on the same socket, the software should explore whether the resources controlled by the lock can be partitioned on a per-socket basis, with dedicated locks for each partition. In some cases, it may even be valuable to further partition those resources across cores within the same socket as core counts increase.

1.6 SUMMARY

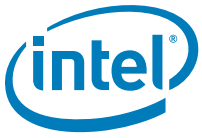
Micro-benchmarks with very tight loops of atomic operations on the same address can exhibit behavior, under heavy contention, that is highly sensitive to a hardware implementation's system latencies. It is not useful to measure the rate at which such a benchmark issues back-to-back atomics and then attempt to draw meaningful conclusions about lock scaling on real software from those measurements.

The two Intel Xeon processor generations discussed in this white paper provide a case in point. On the Intel Xeon processor X5600 series, under very brief lock re-entry times, it is possible for a core to do multiple back-to-back updates, while it has ownership of a lock, before losing that ownership to another core. This can occur because the time it



takes an Intel Xeon processor X5600 series to start the ownership transfer, from one core to the next, is long enough that multiple atomics can be issued by software before the hardware begins the transfer. These back-to-back updates, in the space of a single ownership transfer between cores, may be counted by the micro-benchmark as multiple successful ownership transfers. The Intel Xeon processor E5-2600 series, on the other hand, starts inter-core lock transfers more quickly, leaving less time for any one core to hang onto the lock and perform multiple sequential updates. When tight lock kernels are used as a metric, such subtle differences in hardware implementation may lead to misleading conclusions, when comparing the lock scaling of two different processors.

Unlike micro-benchmarks, real-life software should avoid such short re-entry and critical section times if consistent scaling across microarchitectures is desired. Specifically, the work done by real software, both while in the critical section and also between lock release and re-entry, should be of reasonably long duration. To avoid these anomalies, our recommendation is for this amount of work to be on the order of several hundred nanoseconds. Software should also consider the other techniques outlined in this paper (like lock batching and resource partitioning) for further improvements in lock scalability.



APPENDIX A
LOCKS

A.1 OVERVIEW

This appendix provides an overview of the various flavors of locks and how they interact with the operating system.

A.2 SPINLOCK

Spinlocks are one of the most common forms of locking. The following example shows one form of spinlock using an integer count for the lock. When the lock value is 1, the lock is free. The lock acquire atomically decrements the count. When the resulting value is zero, the lock is acquired and the critical section can execute. When it is negative, another thread is assumed to be holding the lock and the spinlock spins until the lock appears free and tries again. When the critical section finishes executing, the lock variable is set back to one, to indicate that it is free. Similar spinlocks can be implemented in other ways (for example, using CMPXCHG or simple XCHG instead of a count); this is merely a representative example implementation.

Figure A-1. Spinlock example using gcc 4.7+ atomic primitives

```
#include <xmmintrin.h>
static volatile int lock = 1; /* 1 means free */
lock:
        while (__atomic_sub_fetch(&lock, 1, __ATOMIC_ACQUIRE) < 0) {
                do
                        _mm_pause();
                while (__atomic_load_n(&lock, __ATOMIC_ACQUIRE) != 1);
        }

unlock:;
        int free_val = 1;
        __atomic_store(&lock, &free_val, __ATOMIC_RELEASE);
```

The lock instruction can be either a LOCK XADD or a LOCK DEC with a test of the condition code. The __atomic_* built-ins with the memory model specifiers ensure that the compiler's optimizer does not reorder the memory accesses. (The atomic primitives can be replaced with others, depending on the compiler or with an inline assembler.) Another important detail is having the PAUSE instruction inside the spinlock. The _mm_pause intrinsic uses the PAUSE instruction to yield the CPU to another hyperthread and to save power. It also serves to slightly limit the rate of accesses on the processor interconnect.

An important detail of the implementation is that it spins reading, not writing, the lock variable. From the CPU's point of view, getting a cache line for writing is more expensive than reading. To optimize performance and lower the impact on the CPU interconnect, it

is best to limit the rate at which this is done. So, the locking code always waits until the lock appears to be free before trying to acquire it again.

Spinlocks can be implemented in other ways than shown in this example. However, using PAUSE and spinning on reading are important for any spinlock.

A.2.1 Interaction with the operating system

A classic spinlock is only as fair as the underlying microarchitecture. It makes no assumption about the order in which the critical sections will run. This has advantages when interacting with the OS scheduler. The OS scheduler chooses a particular order to run threads spinning on the lock, and whatever spinning thread gets scheduled has a chance to get the lock. However, when the thread that currently owns the lock is preempted, there may still be a slowdown.

This implies that with oversubscription (more threads running than available logical CPUs), the performance of spinlocks can depend heavily on the exact OS scheduler behavior, and may change drastically with operating system or VM updates.

Adding a yield system call into the spinlock is a common change. However, this can also cause problems with oversubscription and lower performance in some circumstances. Yield allows the OS to give up the time slice, which can cause starvation when another thread that is running consumes all spare CPU cycles. For general purpose spinlocks, yielding to the kernel should be avoided.

Some OSs may have specific system calls to block on locks that avoid the starvation problem with yield (for example, the futex system call on Linux*). Implementing such hybrid spinning/blocking locks is outside the scope of this documentation. However, when a hybrid lock is implemented, it is important not to violate the recommendations specified here.

A.3 TICKET LOCK

A ticket lock is a lock that enforces ordering and fairness. A simple ticket lock consists of two counters: the head and the tail. Each number represents a ticket. A locker takes a ticket by atomically incrementing the tail and saving the previous value (its ticket). This can be done with the XADD instruction. Then it spins until the head counter reaches the value of its ticket. To unlock, it increments the head counter by one (which need not be performed atomically because there can only be one thread unlocking at a time).

The ticket counters must be sized to accommodate the maximum number of threads that could be taking a lock in parallel. The head and tail can be combined into a single word, so that a single XADD instruction can atomically get both the head/tail values and increment the tail. In this case, the tail must be in the high part to avoid carry overflows corrupting the head.

One advantage of the ticket lock is that it will only ever write to the lock once, minimizing write traffic on the CPU interconnect.

The following shows an example of a simple global variant of a ticket lock. More sophisticated variants support spinning on private memory to optimize access patterns. This simple variant has the advantage that it needs only a single atomic operation on an uncontended lock.

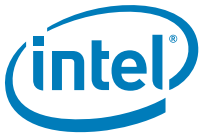


Figure A-2. Ticket lock example using gcc 4.7+ atomic primitives

```
#include <xmmintrin.h>
/* Lower 16 bits: head count, higher 16 bits: tail. Maximum 16k threads */
unsigned int ticket_lock = 0;
#define TAIL_SHIFT  16
#define HEAD_MASK   0xffff

lock:
        unsigned tail, head;
        tail = __atomic_fetch_add(&ticket_lock, 1 << TAIL_SHIFT,
        __ATOMIC_ACQUIRE);
        head = tail & HEAD_MASK;
        tail >>= TAIL_SHIFT;
        while (head != tail) {
                _mm_pause();
                head = __atomic_load_n(&ticket_lock, __ATOMIC_ACQUIRE) &
                        HEAD_MASK;
        }

unlock:
        /* Note: violates strict aliasing (use –fno-strict-aliasing with gcc) */
        /* Increment head (lower 16bit) only. Assumes Little-Endian */
        (*((unsigned short *)&ticket_lock))++;
        /* Compiler barrier */
        __atomic_thread_fence(__ATOMIC_RELEASE);
```

A.3.1 Interaction with the operating system

The ticket lock enforces the order through its tickets. However, this can conflict with the operating system (or virtual machine) scheduler when the CPUs are oversubscribed (more threads running than logical CPUs available). In this case, the scheduler may choose to schedule a thread with a later ticket, but block a thread with an earlier ticket. Because the thread with a later ticket can never acquire the lock before the earlier one, this just wastes CPU time slices and causes delays and poor performance. This implies that with oversubscription ticket locks can rely heavily on the exact scheduler behavior and may change drastically with operating system or VM updates.

When oversubscription is possible, it is better to avoid ticket locks.

A.4 READ/WRITE LOCKS

A read/write lock allows multiple readers, but only a single writer, in the critical section. Typically, this is implemented by a single counter or a small number of counters all

located on the same cache line. A reader changes the counter atomically and checks the writer status. If there is no writer, it continues. However, if there is a writer, it spins until the writer exits. When many readers are executing in parallel and at a high frequency, this results in many transfers of the cache line between cores and may result in poor scaling.

In general, read/write locks should be used only when the lock inter-arrival time is reasonably long and also, the reader-critical section is reasonably long. The performance of read/write locks also depends heavily on the write ratio. Many simple implementations have starvation problems, with frequent writers potentially starving readers (live lock).

Simple read locks do not scale as well as one would expect, in theory, because the communication cost of the cache line that is modified by each reader limits scaling. This is especially visible for short critical regions with a short inter-arrival time. Contended read locks should be used only for relatively long critical regions or long lock inter-arrival times and when there are far more readers than writers. The performance of read locks typically goes down dramatically as the number of writers increases.

For read locks with frequent reading from many cores and low inter-arrival times, consider using big read locks instead.

The recommendations for spinlocks apply to read-write locks, too. PAUSE should be used in the loop and any spin loop should spin reading, and write only when it sees the lock value change.

Simple spinlock implementations are often implemented using a spinlock to protect the rwlock state, and using multiple counters protected by that lock for reads/writes. Switching to a single atomic count, directly manipulated by atomic operations without an auxiliary lock, can improve scalability by optimizing cache line transitions. One way to do that is to use the upper bits of the counter for writers and the lower bits for readers. This can be atomically modified and checked with XADD. Similar schemes can be implemented with CMPXCHG.

A.4.1 Interaction with the operating system

The interaction of read/write locks with the operating system is the same as for classic spinlocks. Typically, read/write locks can also be implemented using operating system primitives to provide blocking read/write locks (for example, with futex on Linux*).

A.5 BIG READ LOCKS

If the number of readers is far higher than the number of writers, a reasonable alternative is a big read lock. Essentially, each thread gets its own read/write lock and it takes its own lock for reading. This minimizes communication overhead for the common reader case. A writer takes all the read/write locks for writing for all threads. This is a trade-off between reader and writer performance, favoring the readers at the cost of more expensive writers. The lock cache line will then be shared in each CPU's cache and readers will execute with minimal lock latency.

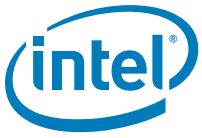


Figure A-3. Partitioned big read lock pseudo code

```
struct {
   union {
          read_lock_t rwlock;
          char cache_line_pad[ROUNDUP(sizeof(read_lock_t), 64)];
          /* Pad array element to avoid false sharing */
   } u;
} br_lock_table[MAX_THREADS];

Reader:
          read_lock(&br_lock_table[my_thread_id()].u.rwlock);  /* Lock */
          … read critical section …
          read_unlock(&br_lock_table[my_thread_id()].u.rwlock); ./* Unlock */

Writer:
          For_each_thread (thr):  /* Write-Lock */
                    write_lock(&br_lock_table[thr].u.rwlock);
          … writer critical section …
          For_each_thread (thr):  /* Write-Unlock */
                    write_unlock(&br_lock_table[thr].u.rwlock);
```

When it's not practical to give each thread its own lock and find all locks from writers, it's also possible to use a number of read/write locks in a table and use a hash mapping to map each thread to a specific lock in the table. This allows arbitrary trade-offs between reader and writer costs, by changing the lock table size.

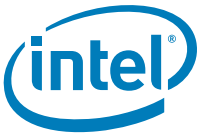
A.5.1 Interaction with the operating system

The interaction of big read lock with the OS is the same as for the read/write locks.

A.6 SLEEPING LOCKS

Operating systems usually also provide blocking locking interfaces. Instead of spinning, these put the blocking task to sleep and wake it up using operating system wake-up mechanisms (monitor/mwait, inter-processor interrupts). The minimum cost of contention is higher with a sleeping lock because it requires a system call with a much longer code path. Often, this cost can be amortized for brief block times by using a hybrid lock that spins for some time before sleeping.

Typically, sleeping locks have more overhead per lock, but can behave better with over-subscription. When a lock sleeps for a long time (such as when doing I/O), it may have less overhead than a spinning lock, as the kernel can often schedule other work onto the blocked CPUs.



On Linux, a user program interface for sleeping locks can either use the futex system call or the pthread lock functions. An overview of basic (not hybrid) futexes is in http://www.akkadia.org/drepper/futex.pdf.

The implementation of sleeping or hybrid spinning-sleeping locks is complex and out of scope of this paper. These locks can be affected by hardware properties, such as idle (C-state) latencies and operating system behavior. The optimal spin time varies with different system configurations. They typically depend less on the actual scheduler behavior than locks implemented by spinning, so their behavior can be more stable over operating system revisions. Generally, sleeping locks are more difficult to analyze than simpler locks because they have more states.