Everything You Still Don't Know About Synchronization and How It Scales

Anshu Raina, Suhas Pai, Mohit Verma, Vikas Goel, and Yuvraj Patel

Department of Computer Sciences, University of Wisconsin-Madison
araina, spai, mohit93, vikasgoel, yuvraj@cs.wisc.edu

Abstract

Scaling the software systems to harness the parallelism offered by today's many-core systems is an extremely challenging task. One key component - synchronization is generally a hindrance to achieving scalability. It is extremely hard to reason about how the synchronization paradigms scale. In this study, we present an analysis of various layers - hardware atomic instructions, software synchronization primitives and applications which use these primitives. We also study the behavior of these synchronization primitives to understand how the thread placement (intra-socket, inter-socket, and hyperthreading) impacts synchronization. We further analyze how the size of the critical section impacts the performance of software synchronization primitives. Throughout the paper, we present various observations and explain the reason behind them.

1 Introduction and Motivation

As predicted by Moore's Law, transistor density gains and reductions in the lithographic feature size has continued but is unlikely to continue for long [1]. The rise in the clock speed and performance of single-core has somewhat saturated. Instead, modern computer architects have started focusing on providing more parallelism within a single chip by providing many cores within a single chip [2, 3].

In order to harness this implicit parallelism within a chip, a lot of research has been conducted to scale systems. Efforts have been put in at various software stack levels to improve the performance of the system by making the system run in parallel [4, 5, 6, 7, 8, 9]. However, one thing that is commonly pointed out in all prior work is that scaling systems is a challenging task. One of the major hindrance to scale systems is synchronization. Extra care has to be taken to ensure synchronization does not end up degrading the performance of the system.

Synchronization refers to an act of coordinating among processes or threads to ensure correctness. In the absence of a proper coordination, the overall result of the computation in a multiple process environment can be incorrect. Most of the time, this coordination is ensured by serializing the execution of the processes. Looking from Amdahl's law [10] point of view, serialization always ends up slowing the system down defeating the purpose of scaling the system [11].

The most common problem faced while designing parallel systems is that the developers are not aware about the effects of scaling on synchronization. Moreover, if the system fails to deliver high performance, it is hard to understand which component within the synchronization primitive is a bottleneck. This is generally attributed to the fact that most of the time developers are not aware about the intricate details of the underlying hardware. It is hard to pin point which factor within the system is failing the scaling. The factors have a diverse range e.g. cache coherence affecting hardware atomics, higher level synchronization primitives w.r.t. the size of the critical section, thread placement and so on.

Understanding the effect of synchronization under scaling thus becomes extremely difficult. Tudor et. al. [12] make an attempt to explain synchronization from a many-core perspective. They explain synchronization in a multi-dimensional way and study four different hardware. However, their study still fails to explain the intricacies of how the latency of the atomic operations varies once the number of threads contending increases. Similarly, the study fails to explain how the software synchronization primitives like mutex, spinlock, semaphore behave when the number of contending threads increases.

In this paper, we extend the work done by [12] and understand synchronization from the following prespective

1. We study synchronization by varying the number of threads that try to contend hence helping us understand how the atomic instructions, soft-

ware synchronization primitives and applications behave when the number of contending threads increases.

2. We study synchronization from the thread placement perspective by placing threads in three different configurations - within the same socket, across sockets and in hyper-threaded fashion where two threads are running on the same physical core. This will help us understand how thread placement impacts synchronization and if there is any advantage when the threads execute within a socket or in hyper-threaded manner.

3. Another parameter we believe is important to understand synchronization is the size of critical section. We vary the size of the critical section in our experiments to understand what impact does size of the critical section have on synchronization.

We study the above parameters by conducting experiments around the following

1. Hardware artifacts - CAS, TAS and FAI

2. Software synchronization primitives - mutex, semaphore and spinlocks.

3. Applications - MongoDB

In our experiments, we also understand the internals of the software synchronization primitives to make sure if the implementation of these primitives is optimal.

Throughput the paper, we try answering certain questions to understand the impact on synchronization while scaling. Few of the interesting observations we observed while analyzing the experimental data are as follows

1. Inter socket communication is costly because of which cache coherency related traffic suffers, ending up making synchronization across sockets expensive.

2. Placing threads on Hyper-threaded cores performs better than just placing them on different physical cores in same socket - We observed that two threads running on the same physical core perform better as sharing the cache helps in lesser cache coherence traffic causing an overall decrease in the latency to execute the synchronization operations.

3. Current implementations of mutex and binary semaphores first try to perform CAS a couple of times and in case of failure, they spin for some time and then perform a system call. We argue that for larger critical sections, CPU cycles are wasted on unnecessary spins because eventually almost every thread performs a system call.

4. Spinlocks with more threads perform worse even on small critical sections - Common belief is that spinlocks are good when critical section size is small. But our study suggests that if the number of threads trying to acquire a spinlock increases, the latency of successfully acquiring the lock increases exponentially because spinlock repeatedly tries a CAS operation without backoff hence increasing the cache coherence overhead. Surprisingly, mutex and semaphore perform better than spinlocks in such a situation.

5. CAS operation is more costly than TAS as CAS is executed using cmpxchg instruction in x86-64 and achieves atomicity using lock prefix. There are many cache coherence state transitions which can increase the latency of CAS.

6. Locks are generally acquired in clusters - During our experiments, we found that the cores physically near the core that is holding a lock are more likely to acquire the lock next compared to the cores that are physically far apart. Generally, the cluster size we found is around 3 to 4 cores.

The rest of the paper is organized as follows. In Section 2, we cover the synchronization basics followed by the experiment details in Section 3. In Section 4, we discuss the key results obtained while conducting the experiments. In Section 5, we briefly discuss related work and in section 6 we talk about the future work. We conclude in Section 7.

2 Synchronization Background

Systems with multiple processing elements have become the norm in recent years. Such systems not only enable running multiple workloads at the same time but also allow programmers to take advantage of the inherent parallelism in a single workload and speed up its execution. Given enough information about the workload, the theoretical speedup of execution by exploiting parallelism can be predicted by Amdahl's law [10]. However, no matter how parallel these workloads are, they need to communicate with various threads. This is because they either need to enforce mutual exclusion when accessing a shared resource, or the nature of the workload requires aggregation and processing of all parallel computations by a single serial component. The process through

which the various parallel components communicate with each other to achieve this is known as synchronization.

Programmers achieve synchronization using various synchronization primitives provided by the operating system, such as spinlocks, mutex, semaphores etc. These synchronization primitives are built using atomic instructions provided by the underlying hardware. We describe the atomic operations provided by hardware and the higher-level synchronization primitives that are built using them in this section.

2.1 Basic Hardware Artifacts

Modern processors provide multiple instructions guaranteed to execute atomically. Each of these instructions has different semantics and can be used to design various synchronization schemes. We describe three key atomic instructions provided by the x86-64 ISA which we study in this work.

Compare-And-Swap (CAS): The CAS instruction compares a value in memory with an expected value; if they are equal, then the value in memory is updated with a new value. In either case, the original value at the memory location is returned. On x86-64 hardware, CAS is implemented using the lock cmpxchg instruction.

Test-And-Set (TAS): The TAS instruction returns the old value present at a memory location, while simultaneously updating it with a new value. This ensures that the memory location has no other intermediate values. On x86-64 hardware, TAS is implemented using the xchg instruction.

Fetch-And-Increment (FAI): The FAI instruction increments the value at a memory location atomically, while returning the original value. The atomic update ensures that the memory location holds no other intermediate value.

2.2 Higher level Synchronization Primitives

Most programmers are familiar with synchronization primitives like mutex, semaphore, and spin locks while writing parallel code. In this section, we discuss how these primitives are implemented using the basic hardware artifacts discussed in the previous subsection. We first provide a brief description of each of these primitives, where they are used, and then provide the algorithm for them used in musl[13]. We emphasize that different implementations[14] may provide some extra optimizations, and have support for additional features, but the basic algorithm remains the same.

Mutex: Mutex is used to provide mutual exclusion. These are most often used when exactly one thread can be allowed to enter the critical section. After the introduction of the futex system call in Linux [15], it became a practice to let the lock be acquired in user space, if possible, to avoid the expensive system call (and the user space to kernel space switching overhead). Thus, internally mutex tries to acquire the lock multiple times using a couple of CAS instructions. If successive CAS on the lock fails, it implies some contention on the lock, so mutex spins for some time in a loop, exiting the spin loop when a loop count of 100 is reached, or it detects there are no waiters, whichever comes first. Next, it tries a CAS again to get the lock, and if this also fails, it registers itself as a waiter on this lock and calls the futex sleep system call. If any of the above CAS succeeds, it implies that lock has been successfully acquired, and the mutex lock call returns. The pseudocode for the mutex primitive is provided below. Notice that the entire algorithm has been broken down into different APIs to allow the user to call any of them. Mutex unlock atomically updates the shared lock to unlock it and wakes up a waiter (if any).

```
pthread_mutex_lock() {
 Try CAS, return if successful
 pthread_mutex_timedlock()
}
pthread_mutex_timedlock() {
 Try CAS, return if successful
 r = pthread_mutex_trylock()
 if (r) return /*lock acquired*/
 spin - max 100 loops or until no waiter on lock
 while (pthread_mutex_trylock()) {
     register as waiter
    do futex sleep syscall
    /*wake up once unlock is called*/
 }
}
pthread_mutex_trylock() {
 Try CAS, return CAS result
}
pthread_mutex_unlock() {
 atomic swap
 if (waiters) futex wakeup syscall
}
```

Semaphore: A semaphore is an object with an integer value that we can manipulate with two routines; in the POSIX standard, these routines are sem_wait() and sem_post()[16]. Semaphores are mostly used for signaling purposes. They may allow multiple threads to enter the critical section, and signal the waiters once they are done. As in mutex, internally semaphore tries CAS a couple of times, before spinning and subsequently blocking. Note that the CAS operation here is implemented in a while(allowed entries > 0) loop because the CAS may succeed in another attempt as long as it is okay for the thread to enter the critical section.

Since multiple threads may be in the critical section at once, semaphore post needs to update the

internal shared count in a while loop till it is successful. The pseudocode for semaphore is provided below. sem_wait() API tries to get the ownership of lock and sem_post() API gives up the ownership of lock.

```
sem_wait() {}
 sem_timedwait()
}
sem_timedwait() {
 sem_trywait()
 spin - max 100 loops or until no waiter on lock
 while (sem_trywait()) {
     register as waiter-atomic_inc
    do futex sleep syscall
    /*wake up once unlock is called*/
 }
}
sem_trywait() {
 while(allowed entries > 0)
 Try CAS, return 0 if successful, else 1
}
sem_post() {
 do {
 } while(CAS allowed entries ++)
 if (waiters) futex wakeup syscall
}
```

Spinlocks: Spinlocks are used to provide mutual exclusion when the critical section size is known to be small. The thread which is trying to acquire a lock is stuck in a while loop to get the lock. The implementation of spinlock involves continuously trying a CAS operation in a while loop, till it succeeds. We note that such an implementation will cause a lot of cache coherence traffic because of multiple threads being active at any given point of time. In our experiments, we used musl[13] library which implements spinlocks without any back-off. This means the threads don't wait before trying the next CAS operation. The pseudocode for spinlock is provided below.

```
spin_lock() {}
 while (CAS(lock));
}
spin_unlock() {
 atomic_store
}
```

3 Experimental Setup

The aim of our experiments is to study the effects of synchronization on different primitives ranging from hardware to software and see how these primitives scale as we increase the number of threads up to the number of cores present in the processor. So, we have divided our study into two layers. In the first layer, we do this study on various hardware artifacts like Compare and Swap (CAS), Test and Set (TAS) and Fetch and Add (FAI). In the second layer, we analyze software synchronization primitives like mutex, semaphore, and spin locks and answer some important questions pertaining to scalability. These experiments are done by varying the placement of threads across different configurations e.g. all the threads present in a single socket, threads distributed across sockets and threads placed on the same physical core but different logical cores in case of hyper-threading.

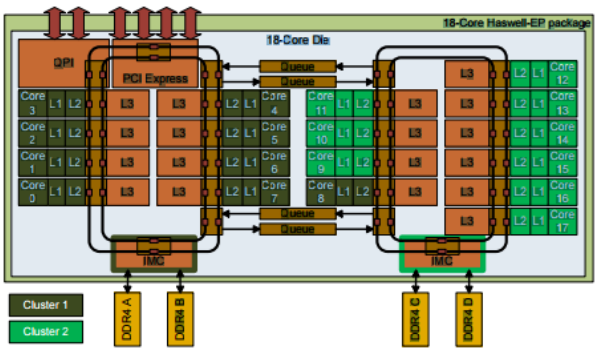

Figure 1: Haswell-EP block diagram: There are two rings with one memory controller each

3.1 Hardware

All our experiments are performed on Intel Xeon E5 v3 machine otherwise called as Haswell-EP. Haswell-EP has three variants-an eight-core die (4, 6, 8-core SKUs), a 12-core die (10, 12-core SKUs), and an 18-core die (14, 16, 18-core SKUs)[2]. The eight-core die uses a single bi-directional ring interconnect and 12-core and 18-core die's use a partitioned design as depicted in Figure. 1 [17]. Eight L3 slices, one memory controller, the QPI interface, and the PCIe controller are connected to one bidirectional ring. Remaining cores (4 or 10), L3 slices, and the second memory controller are connected to another bi-directional ring. The ring topology is hidden from the operating system by default.

The architecture uses MESIF protocol[18] to maintain cache coherence. Modified, exclusive, shared and invalid are inherited from MESI and a state forward is added to enable cache-to-cache transfers of shared lines. The protocol is implemented by caching agents (CAs) within each L3 slice and home agents (HAs) within each memory controller as depicted in Figure. 2[17]. MESIF protocol supports three snoop modes: source snoop mode, home snoop mode and cluter-on-die mode. In source snoop mode, snoops are sent by the caching agents and it is enabled by default. Note that in our experiments we are using a default configuration which supports source snooping. Also we have clocked our machine at 1.2GHz so as to keep uniformity in our results.

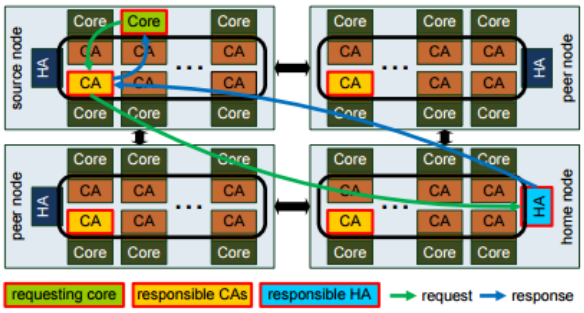

Figure 2: Cache Coherence Implementation

3.2 Basic Hardware Artifacts

To study the first layer of synchronization primitives, i.e. the atomic instructions provided by hardware, we make use of the open source benchmarking system called ccbench [19], developed at EPFL. We use this tool to spawn varying number of threads in the system and have each of them perform atomic operations in parallel. Each experiment creates a certain number of threads, all of which arrive at a barrier and then proceed to perform atomic operations on a shared memory location in a loop, until all of them succeed. We measure the time taken for execution in each thread using the RDTSC(read-time stamp counter) instruction in the x86-64 ISA. We ensure that these measurements are not affected by reordering of instructions in the out-of-order CPU by making use of fence instructions at appropriate locations. The latency numbers we report do not include the fence instruction latency. Each experiment is repeated for 100000 iterations and the median value of the latencies are reported.

We conduct these experiments for all three instructions described above: CAS, TAS and FAI. We vary the number of threads in the system from a minimum of 1 to the maximum number of logical cores in the system. We also place the threads in various configurations: single socket without hyper-threading, single socket with hyper-threading, two sockets without hyper-threading, two sockets with hyper-threading.

3.3 Higher level primitives

We use musl[13] to study the behavior of spinlocks, mutex and binary semaphore with scaling. Musl is a lightweight and simple standard C library that conforms to the requirements of the ISO C99 standard. We change the API definitions of the above primitives in musl, so that we can pass an additional structure as argument. This structure is maintained per thread, and records the latencies of various operations that take place. As explained before, time is measured using RDTSC instruction. Since we have clocked our CPU at a fixed frequency of 1.2GHz, we then easily convert the cycle count given by RDTSC into micro or nano seconds depending on the precision of the results required. We don't use glibc in our experiments as glibc is heavyweight and is difficult to change for instrumentation. We use the following critical section across which mutex/ semaphore/ spinlock is applied:

```
FOR(0 ... LOOP_COUNT) {
    count := count + 1;
}
```

LOOP_COUNT refers to the number of times the instructions within the for loop are executed. For our experiments, we use LOOP_COUNT of size 100, 1000 and 10000 which are representative of small, intermediate and larger critical section. Note that "count" is a volatile variable here and we have made it volatile to avoid any compiler optimizations in the critical section.

3.4 Applications

To understand how thread placement can impact the performance of the applications, we conduct experiments on MongoDB [20]. MongoDB is a free and open source document-oriented database. MongoDB provides high availability of the data by using replication. A record in MongoDB is a document which is a data structure composed of field and value pairs. MongoDB stores documents in collections and thus are analogous to the tables in relational databases. Databases hold collections of documents. MongoDB supports two storage engine namely WiredTiger and MMAP. We are using MongoDB version 3.2 which has WiredTiger storage engine as the default storage engine.

MongoDB provides a variety of acknowledgment behavior for write operations. MongoDB describes this level of acknowledgment in terms of write concern. A write concern is specified as

{
w: <value>, j: <boolean>, wtimeout: <number>
}

where, (i) w option specifies that the write operation has been propagated to a specified number of mongod instances. Here, if the value is 1, it means that the write operation has propagated to a standalone mongod instance. While a value of 0 means that no acknowledgment of the write operation. A higher value is used when the write operation needs to be propagated to many replicas. (ii) j option specifies that a write operation has been written to the journal. (iii) the wtimeout option specifies a time

limit to prevent write operations from blocking indefinitely.

For our experiments, we are using the w option with value 0 and j to be false. We believe that this is going to stress the system the maximum as there is no I/O path involved during the write operation. We study MongoDB for write and read operations respectively. We measure the overall throughput for reading and write operations when varying number of clients try to insert or read the documents in parallel.

For writes, we insert 50 million documents in total while the number of clients varies. Similarly, for reads, we read the same 50 million documents inserted earlier while varying the number of threads. We run the experiments 50 times and take an average of the completion times for reporting purpose.

As experimenting with MongoDB needs two machines, we are using cloudlab machines for the same. The configuration of the machines is the same as that described in Section 3.1. On one machine, MongoDB server runs and the data is persisted on SSD disk while the clients run on another machine and sends/receives data from the server via a 10 Gbps network.

4 Observation and Results

In this section, we present our study on the three layers. Throughout the following subsections, we first ask questions that can be answered by such a study, and we then present our observations and explanations for them.

4.1 Basic Hardware Artifacts

In the experiments for basic hardware artifacts, we try to find answers to the following questions:

- How do latencies of atomic instructions scale with varying levels of contention in the system?
- Does placement of threads in the system affect their latencies? For example, are latencies affected by relative placements of threads among sockets?
- Does the use of hardware multi-threading, when available, have any effect on scaling?

Through our experimental results, we see that the latencies of all three instructions that we study increase linearly with increasing contention in the system. Figures 3, 4 and 5 show this trend. Note that the latencies in Figure 3 for the CAS instruction is in microseconds, while the other two are in nanoseconds. This is because the CAS instruction in x86-64 is executed using the cmpxchg instruction

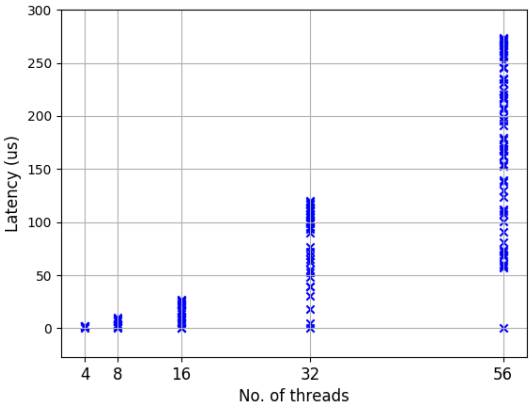

Figure 3: Latency trends for CAS with number of threads

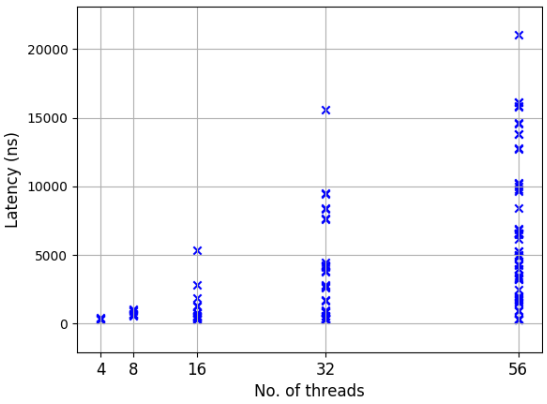

Figure 4: Latency trends for TAS with number of threads

which does a read-modify-write update of memory, and atomicity is achieved using the lock prefix [21]. As a result of this implementation, there are multiple cache coherence state transitions which happen when this instruction is executed, and this may increase the latency of CAS instruction compared to TAS.

Figures 6, 7 and 8 show the latencies for the three instructions when 8 threads are placed in different configurations in the system. Here, 1S No HT corresponds to placing threads on a single socket without the use of hyper-threaded cores, while 1S HT corresponds to threads being placed on a single socket, while half of them are placed on the same physical cores as the other half using hyper-threading. Similarly, 2S No HT and 2S HT represent the non-hyper-threaded and hyper-threaded configurations in the two sockets. In general, we see that use of hyper-threading provides some decrease in latency compared to the case when it is not used. This is because the use of hyper-threading implies the threads are placed on different logical cores while actually run-

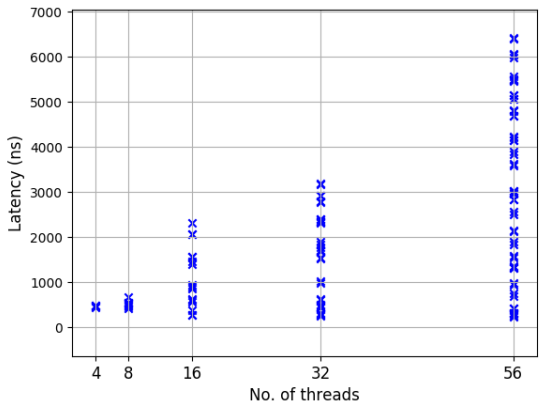

Figure 5: Latency trends for FAI with number of threads

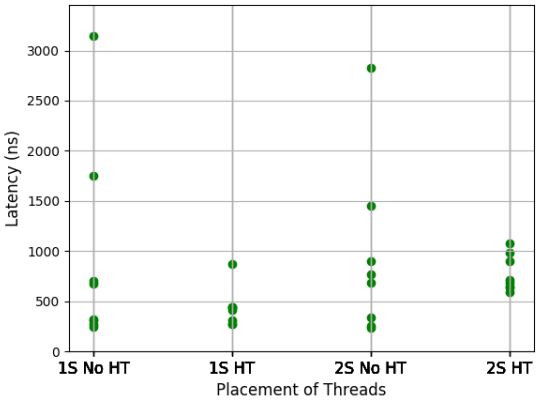

Figure 7: Latencies for TAS with 8 threads placed in different configurations

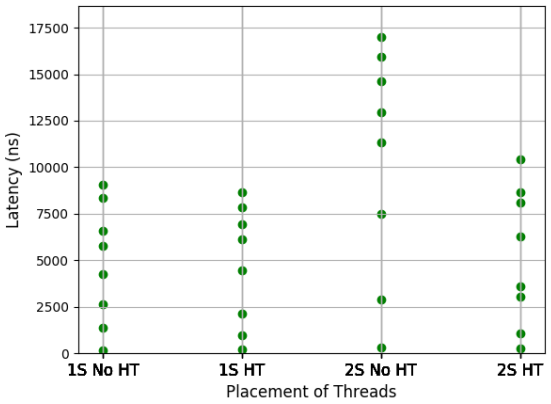

Figure 6: Latencies for CAS with 8 threads placed in different configurations

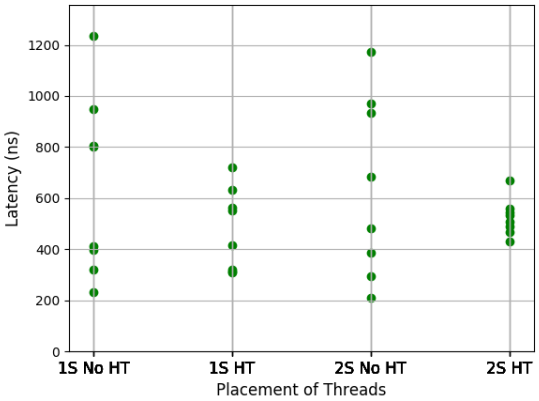

Figure 8: Latencies for FAI with 8 threads placed in different configurations

ning on the same physical core as another thread. As a result, caches are shared between these threads resulting in lower cache coherence traffic in the system and hence lesser delays.

Figure 9 shows a better visualization of effects of thread placements on latencies for CAS. It can be seen that the latency is least when two threads perform CAS on hyper-threaded cores due to reasons mentioned above. Performing a CAS operation across sockets has the highest latency due to inter-socket communication. It can be seen that the latency is about 1.5x for inter-socket compared to intra-socket.

4.2 Higher level primitives

In this subsection, we present our observations on higher level primitives. This subsection is structured as a question-answer discussion for better understanding.

4.2.1 How do synchronization primitives behave as critical section size varies?

To answer this question, we do our experiments for small, medium and large sized critical sections. For these experiments, 14 threads are contending on a single mutex lock. Figure 28 shows the timeline for the 14 contending threads placed on 2 different sockets when the critical section is small (100 loop count). The timeline is color-coded to identify different events: lock acquisition begin, lock acquisition successful, unlock begin, and unlock end. Note that in this case, a thread enters a lock acquisition phase, successfully acquires the lock, performs the critical section work, perform unlock and exits. So, there is no contention in the system, as critical section got over before other threads were even scheduled.

Figure 29 shows the timeline for 14 threads on 2 sockets when the critical section is large (10000 loop count). We find that almost every thread tried to acquire the lock, and does a system call to sleep, while one thread is in its critical section. When this

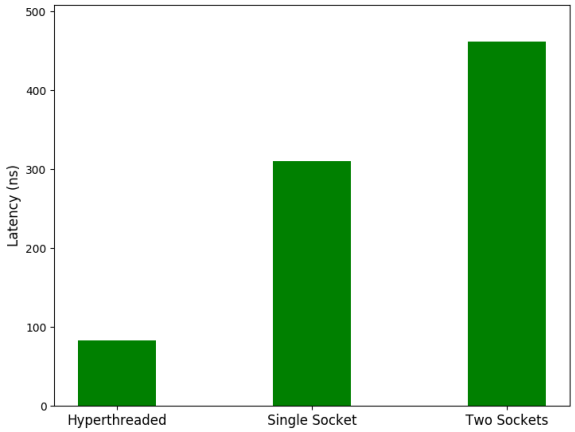

Figure 9: Relative performance of CAS for different thread placements

thread exits critical section, we observe that threads acquire the lock in the order they registered to acquire it. Thus, there is a first-come-first-serve ordering of thread wake-ups.

Figure 30 shows the timeline for 14 threads on 2 sockets when the critical section size is intermediate (1000 loop count). The timeline shows that some (not all) threads tried to acquire the lock when one thread was in critical section. Interestingly, when this thread exits its critical section, another thread which tried to acquire the lock later, ended up getting the lock. Thus, the thread woken-up due to the wake-up system call has to sleep again. In general, since the critical section is not big enough, some threads which were scheduled later start contending with the thread just woken up and create this mess. Thus, we conclude that a FCFS wake-up ordering doesn't imply a FCFS lock acquisition.

4.2.2 Performance scaling as the number of threads increase

Figure 10 shows that the maximum spin count of spinlock before it successfully acquires the lock increases exponentially as we increase the number of threads. In subsection 4.2.5, we show that spinlocks perform worse than mutex and semaphores even for small critical sections when the number of threads is more.

Mutex and semaphore also show an increase in the latency as the number of threads increases. Figure 11 shows the latency of mutex lock acquisition on a single socket with a critical section of relative size 100. Figure 12 shows that performance is even worse when the same analysis is done for threads placed across sockets. The trend continues when we increase the number of threads to 56 on a configuration where hyper-threading is turned on. There is a lot of con-

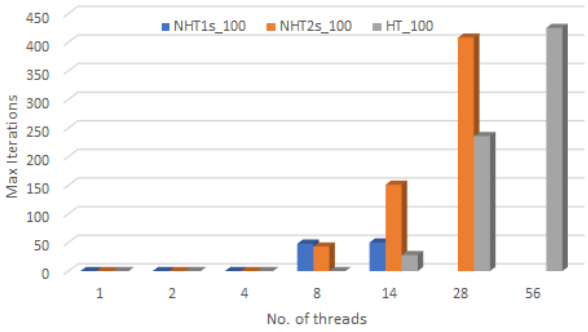

Figure 10: Maximum number of spins for Spinlock

tention when we go to 28 and 56 threads with hyper-threading as can be seen in Figure 13.

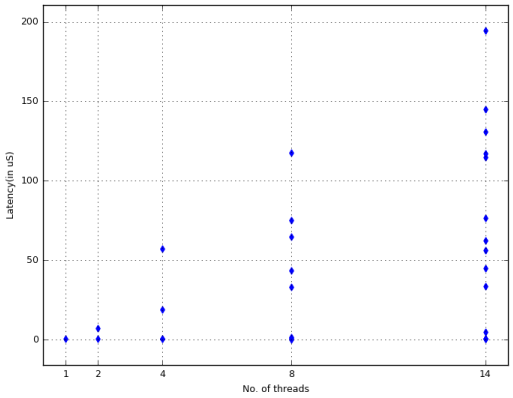

Figure 11: Latency of Mutex lock acquisition on 1 socket

We next present an analysis of the scaling of mutexes as the number of threads increase. Figure 14 shows the breakup of stages a thread undergoes in order to acquire a lock. As explained in section 3, mutex performs CAS operation a couple of times. If it fails to do a successful CAS, it spins for some time and then tries CAS again. In case, it still doesn't get the lock, it performs a futex system call to go to sleep. In Figure 14, the spike in the latency is due to threads contending for a lock. They perform CAS multiple numbers of times, spin for some time and then get the lock. While others get the lock in the first CAS, the lock acquisition time for the contending threads increase by 4x. Figure 15 presents a similar analysis when the threads are placed on hyper-threaded cores of a single socket. Figure 16 extends this observation to the two sockets in which we see more contention than the single socket. A couple of threads get the lock after three CAS operations, others have to spin

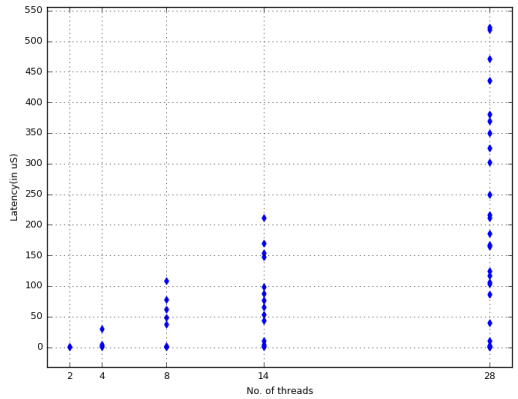

Figure 12: Latency of Mutex lock acquisition on 2 sockets

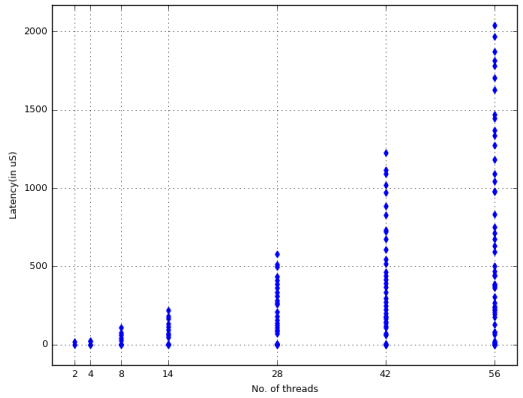

Figure 13: Latency of Mutex lock acquisition on 2 sockets with Hyperthreading

as well. Note the increase in lock acquisition time by 8x which is twice more than that observed in Figure 14. As shown in subsection 4.1, CAS across sockets can be 2x expensive which attributes to this result.

4.2.3 Impact of thread placement on performance

The next question which we answer is the impact of thread placement on performance. Table 1 shows the data for 14 threads contending as the thread placement varies. NHT stands for Non-Hyperthreaded thread placement, HT represents Hyperthreaded thread placement. 100 and 1000 represent the loop count of the critical section as explained in the earlier sections. The numbers for first, second and third CAS represent the percentage of the threads getting successful in their CAS attempts re-

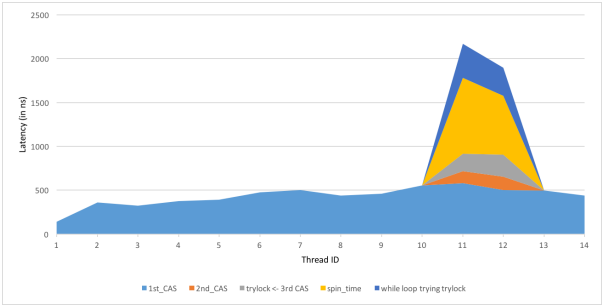

Figure 14: Mutex lock overhead breakup for 14 threads on 1 socket

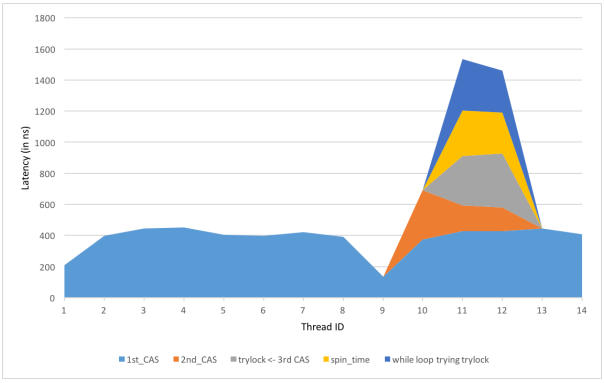

Figure 15: Mutex lock overhead breakup for 14 threads on 1 socket with hyper-threaded cores

spectively. As discussed in section 3, mutex tries CAS multiple times, backs-off if there is contention, spins, and does a system call. What we see is that percentage of threads succeeding the first CAS is more in single socket than in two sockets. We also note that the number of system calls in the threads rise when threads are placed across sockets. Table 1 shows that in the single socket non-hyper-threaded run for a critical section of relative size 1000, 7 threads perform system call once and 4 threads perform system call twice. In contrast, when threads are placed across sockets, 4 threads perform system call once, 6 threads twice and one thread does it thrice which makes the inter-socket performance worse than the intra-socket. Table 1 also shows that almost all threads end up doing system call(more than once) even though they spin. We conclude that it might not make sense to spin, and waste CPU cycles when the critical section is large. We did not show the results of binary semaphore because it is similar.

We perform a similar study for spinlock as given in Table 2 and find that the maximum number of spins in a loop is highest when the threads are placed across sockets. For threads placed on hyper-threaded

	1st CAS	2nd CAS	3rd CAS	while loop - try lock	Syscall	Spin didn't complete	No of syscalls
NHT1s_1000	22	0	0	0	78	1	7-1/4-2
NHT2s_1000	15	0	0	7	78	0	4-1/6-2/1-3
HT_1000	22	0	0	14	64	3	6-1/3-2
NHT1s_100	86	0	0	14	0	2	0
NHT2s_100	43	0	14	43	2	2	0
HT_100	79	7	0	14	0	0	0

Table 1: Mutex Percentage of threads completing at each stage.

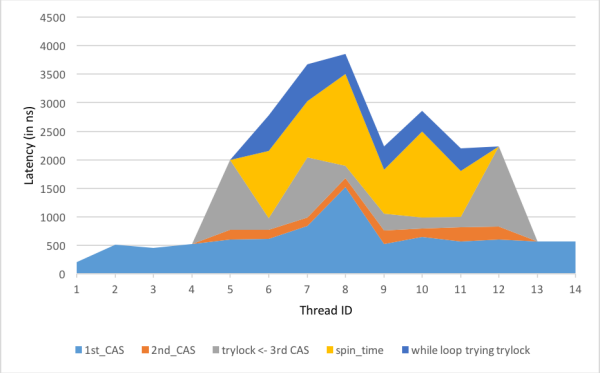

Figure 16: Mutex lock overhead breakup for 14 threads on 2 sockets

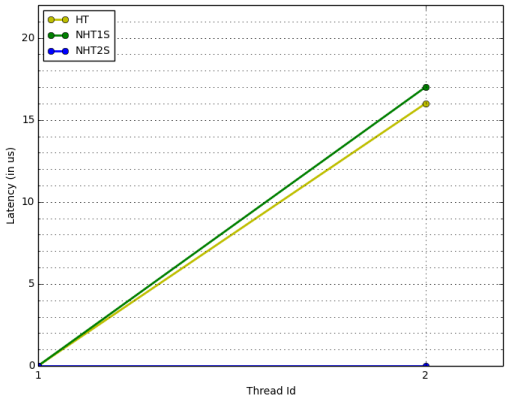

Figure 17: Semaphore latency variation for 2 threads with a relative critical section size of 1000

Configuration	CS size	Max spin count
NHT1s	1000	3719
NHT2s	1000	4275
HT	1000	3242
NHT1s	100	50
NHT2s	100	151
HT	100	28

Table 2: Spin count variation with thread placement for spinlock

cores, this spin count is the least. The trend is same if critical section size is changed.

4.2.4 Variation of max and min overheads as number of threads vary

In this section, we analyze the maximum and minimum overheads for a semaphore wait operation as we increase the number of threads. Figure 17 shows the latency of the sem-wait operation when the number of threads is two. As the number of threads increases to 4 in Figure 18, the sem-wait latency increases. Note that for a lesser number of threads, Figure 17 shows that the worst-case latency of sem-wait for two threads placed across sockets is lesser than when two threads are placed on the same sockets. A timeline of these threads shows that one thread completed execution even before other thread got scheduled.

The same trend continues as we increase the number of threads to 8 in Figure 19 and then 14 in Figure 20. Now, the inter-socket sem-wait latencies are worse than intra-socket thread placement. Notice that contention starts happening now, so inter-socket latencies worsen. For each of these cases. the latencies when threads are placed on hyper-threaded cores are same or slightly better than when threads are placed on separate physical cores.

4.2.5 Comparison of different primitives with each other

Figure 21 shows the variation of semaphore wait latency for 14 threads when the relative critical section size is 100. Figure 22 presents the same plot for mutex with the same critical section size. We can see from these graphs that the behavior of both semaphore and mutex is similar. Mutex is slightly more costly compared to semaphore as mutex issues one extra CAS than semaphore.

Figure 23 shows the variation of spinlock latency for 14 threads in different configurations with a relative critical section size of 100. This figure shows that the latency of spinlocks is more than the latency of mutex for 14 threads with a critical section of relative size 100. As discussed in subsection 4.1, mutex and semaphores have back off when successive CAS operations fail. Spinlock performs worse because as more

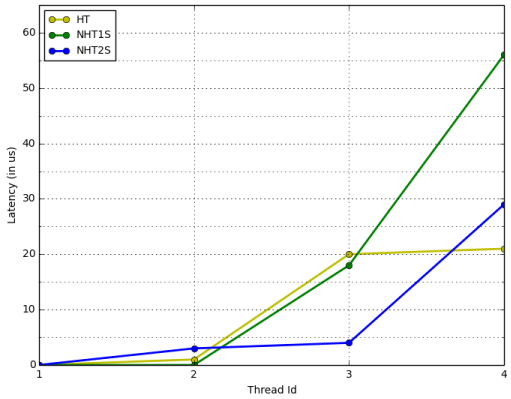

Figure 18: Semaphore latency variation for 4 threads with a relative critical section size of 1000

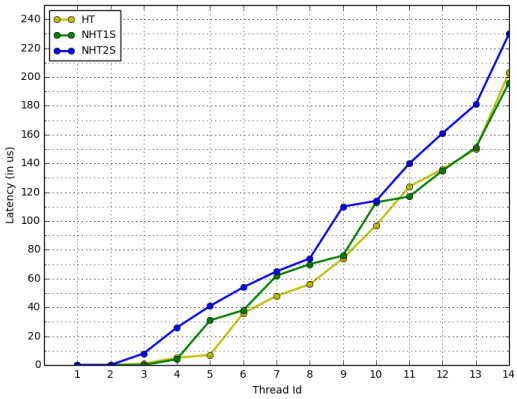

Figure 20: Semaphore latency variation for 14 threads with a relative critical section size of 1000

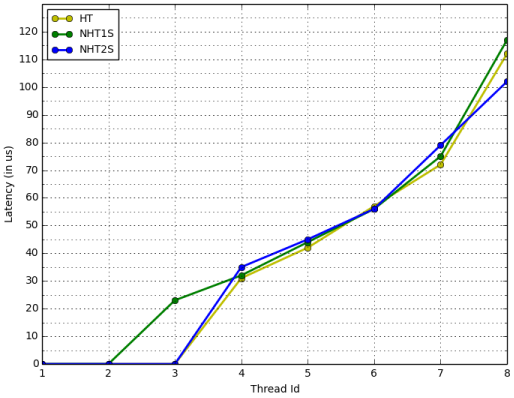

Figure 19: Semaphore latency variation for 8 threads with a relative critical section size of 1000

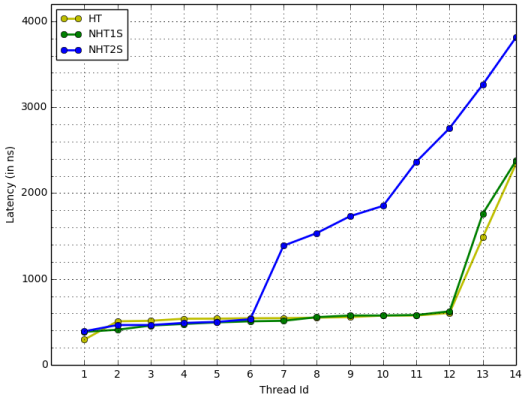

Figure 21: Semaphore wait latency variation for 14 threads in different configurations with a relative critical section size of 100

and more threads try to do a CAS in parallel, it leads to an increase in the cache coherence traffic resulting in larger latencies. We infer that if the number of threads is more, it makes sense not to use spinlocks.

4.3 Applications

In this section, we study how thread placement impacts the overall performance of an application (MongoDB). We believe that by intelligent thread placement, the cache coherence traffic can be reduced thereby improving the performance of the application.

4.3.1 Write Throughput

We conduct read and write experiments on MongoDB and report the overall read and write throughput in operations/second in Figure 24. As seen from the figure, the overall write throughput is more by a factor of $\tilde{1}$5-20% for the intra-socket case. Surprisingly, hyperthreading does not increase the throughput of the system. Rather the throughput is similar to the one seen in the inter-socket case.

The probable reason for such a behavior in case of hyperthreading seems to be from the fact that two threads share the same physical CPU and cache thereby limiting the overall effectiveness. As the physical resources are stressed, it may end up evicting cache lines multiple times in order to accommodate the data for the other thread. This competition eventually leads to lesser throughput compared to the intra-socket case.

Another point to note here is that the overall average CPU utilization as seen in Figure 25 is al-

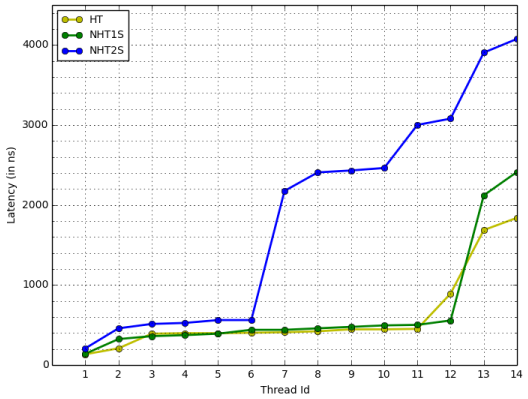

Figure 22: Mutex lock latency variation for 14 threads in different configurations with a critical section size of 100

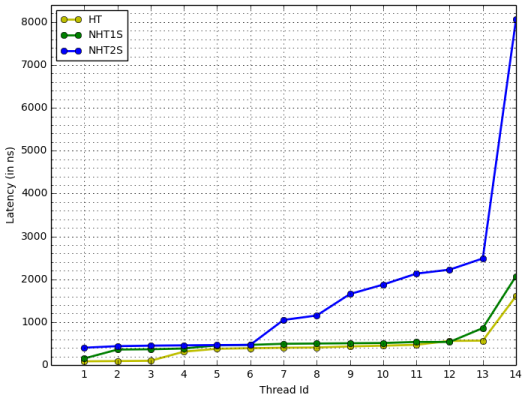

Figure 23: Spinlock latency variation for 14 threads in different configurations with a critical section size of 100

most similar in all the three cases. We looked at the amount of I/O issued and found that the system is not bottlenecked due to I/O. Similarly, the network is also not a bottleneck. This suggests that the threads are contending in order to get locks and hence wasting a lot of CPU without doing any useful work.

To understand the locking pattern for the write path, we use BCC [22] that relies on ebpf to do efficient kernel tracing. Using BCC, we instrument the code to trap every time a pthread_mutex_lock() call is issued. We then print the entire stack trace when mutex lock call is issued. We find that for every single write being issued by the client, MongoDB takes 16 locks. This explains the reason why as the number of threads increase, the contention increases, not letting the CPU do useful work. Providing a detailed analysis on each type of lock is out of scope of this project.

When the lock access happens across sockets, the overall cache coherence traffic increases thereby leading to poor performance compared to an intra-socket case.

4.3.2 Read Throughput

We do a similar exercise as discussed above for read workload that can be seen in Figure 26. As seen from the figure, the overall read throughput of inter and intra-socket case remains the same. As the number of threads on hyperthreaded cores increase, the read throughput decreases by about 10-12%.

We attribute this behavior to the increase in contention of physical resources by two threads running on the same physical core. Figure 27 shows the CPU utilization for all three cases and we see that the CPU utilization is more or less the same.

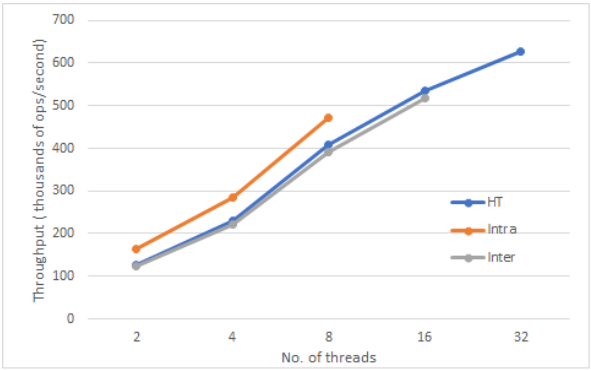

Figure 24: MongoDB write throughput for varying number of threads and different thread placement

Using a similar code instrumentation technique, we find that in the read path, there are 18 locks overall that are taken. We had expected that a fair amount of contention would degrade the performance of the inter socket case but that is not being observed during our experiments. Further analysis is needed to study the cache footprint for read operations and how it impacts the performance.

5 Related Work

Through this paper, we have tried to closely identify the problems that are seen during synchronization while scaling. As per our knowledge, the only work that comes close to our study is the one done by David et al. [12]. They also study synchronization in a layered manner and on various hardware platforms. However, their study only concentrates on understanding synchronization between two threads.

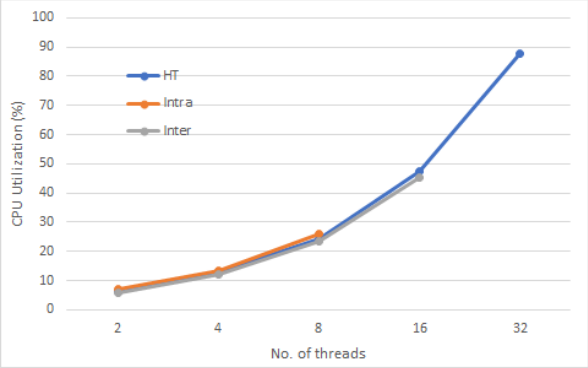

Figure 25: MongoDB CPU utilization for writes while varying number of threads and different thread placement

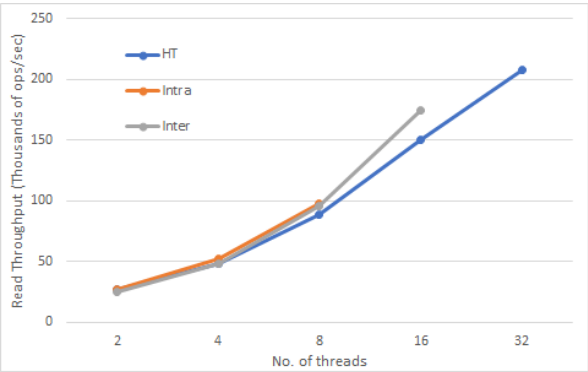

Figure 26: MongoDB read throughput for varying number of threads and different thread placement

It fails to answer what happens when the number of threads increases and thus contention increases. We try to explain this behavior through this paper.

Similarly, our study includes how various software level synchronization primitives behave internally in the presence of contention. They do not discuss how the internals of each primitive behave when multiple threads contend.

We also try to study the impact of the critical section on synchronization. We believe that critical section size does have a role to play while scaling. Our results have shown that spinlocks do not perform well even for smaller critical section size when the number of threads increases. Such interesting observations are missing in the other work so far.

Many others have tried to work around and build scalable systems by trying to reduce the level of synchronization needed for efficient working [5, 4]. They assume that designing systems to rely less on synchronization will help in scaling the systems to 100's

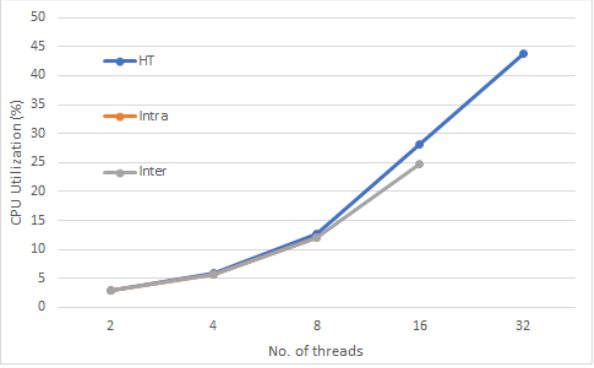

Figure 27: MongoDB CPU utilization for writes while varying number of threads and different thread placement

to 1000's of cores. However, they never study the root cause of why synchronization does not perform well with scaling. Our study tries to address such problems faced and our study does emphasize that the performance of software level primitives is dependent on the performance of hardware artifacts.

6 Future Work

We could not complete a lot of experiments that should be included in such a study. Without the following, the understanding of synchronization under scaling will not be complete. We point out that we could not complete all of this due to resource constraints.

1. We run our current experiments only an Intel Xeon platform. But, we want to understand the impact of different cache coherence protocols (directory vs snooping protocols, for example) on the behavior of synchronization primitives. But due to the unavailability of the hardware, we couldn't conduct experiments on such a platform.

2. For now, we have only analyzed binary semaphore. However, it will be interesting to understand how counting semaphore will scale under the parameters which we have used for our measurement. As counting semaphore allows more than one threads to enter the critical section, our belief is that the contention will be more than what we have observed so far in our results.

3. We want to study more applications to understand if the thread placement can help in an application's performance. We believe thread

placement can help to decrease the cache coherence traffic generated by software synchronization primitives. The results obtained from such a study can help us understand if some kind of cooperative scheduling will help in improving the performance of the applications. The data we have collected so far from MongoDB analysis isn't enough to draw conclusions.

4. To expand our study, we want to include more software level primitives like conditional variables and barrier in our study. Expanding the current study to other primitives will help us understand the behavior of each primitive in more detail.

5. David et.al. [12] have measured throughput of various synchronization mechanisms. However, we believe that with varying length of non-critical section code, the behavior of CAS and TAS operations will be an important part of the study.

6. Lastly, we want to rewrite the software synchronization primitives using TAS instead of CAS and then measure how they behave so that we can get a better understanding and also propose the use of TAS as in our study as we have observed that performance of CAS is always worse than TAS.

7 Conclusion

Our work presents a detailed study of synchronization starting from hardware primitives and then moving up towards the software primitives to applications. We find that the basic hardware artifacts like CAS, TAS, and FAI are closely related with software synchronization primitives like mutex, spinlocks, and semaphores. One needs to have a good understanding of the underlying architecture in order to get performance while scaling. In our experiments we try to answer the number of questions like performance scaling of a given primitive as the number of threads are increased, the effect of placement of threads in different configurations and the comparison of different primitives with each other. Synchronization is hard but with the right information about the hardware, what synchronization primitives to use and application level technical knowledge, one can design scalable system well.

References

[1] C. A. Mack, "Fifty years of moore's law," *IEEE Transactions on Semiconductor Manufacturing*, vol. 24, no. 2, pp. 202–207, May 2011.

[2] "Intel® xeon® processor e5 v3 product family - processor specification update, intel, september 2016." [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v3-spec-update.pdf

[3] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "Tile64 - processor: A 64-core soc with mesh interconnect," in *2008 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, Feb 2008, pp. 88–598.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629579

[5] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): The case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 76–85, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1531793.1531805

[6] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 43–57. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855745

[7] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of linux scalability to many cores," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924944

[8] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss:

An evaluation of concurrency control with one thousand cores," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 209–220, Nov. 2014. [Online]. Available: http://dx.doi.org/10.14778/2735508.2735511

[9] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-core key-value store," in *Proceedings of the 2011 International Green Computing Conference and Workshops*, ser. IGCC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/IGCC.2011.6008565

[10] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[11] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.209

[12] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 33–48. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522714

[13] "musl libc." [Online]. Available: http://www.musl-libc.org

[14] "gnu libc." [Online]. Available: https://www.gnu.org/software/libc/

[15] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," in *AUUG Conference Proceedings*, vol. 85. AUUG, Inc., 2002.

[16] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*. Arpaci-Dusseau Books Wisconsin, 2014, vol. 151.

[17] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel, "Cache coherence protocol and memory performance of the intel haswell-ep architecture," in *Parallel Processing (ICPP), 2015 44th International Conference on*. IEEE, 2015, pp. 739–748.

[18] A. Intel, "Introduction to the intel quickpath interconnect," *White Paper*, 2009.

[19] V. Trigonakis, "Ccbench." [Online]. Available: http://lpd.epfl.ch/site/ccbench

[20] MongoDB, "MongoDB," https://www.mongodb.org/.

[21] D. Dice, "atomic fetch-and-add vs compare-and-swap | Oracle David Dice's Blog." [Online]. Available: https://blogs.oracle.com/dave/atomic-fetch-and-add-vs-compare-and-swap

[22] "Bpf compiler collection - bcc." [Online]. Available: https://github.com/iovisor/bcc

0						5-16- entry								
249						5-16- lock exit								
2494						5-16- unlock start								
2897						5-16- exit								
5141								7-17- entry						
5757								7-17- lock exit						
8074								7-17- unlock start						
8690								7-17- exit						
9453										9-18- entry				
10003										9-18- lock exit				
12343										9-18- unlock start				
12973										9-18- exit				
18048												11-19- entry		
18679												11-19- lock exit		
19558									8-4- entry					
20893												11-19- unlock start		
22169									8-4- lock exit					
22448												11-19- exit		
22558														13-20- entry
24538									8-4- unlock start					
25264											10-5- entry			
25418									8-4- exit					
26159														13-20- lock exit
28439							6-3- entry							
28454														13-20- unlock start
29319														13-20- exit
29950											10-5- lock exit			
32311											10-5- unlock start			
33132											10-5- exit			
33331							6-3- lock exit							
34042													12-6- entry	
35398							6-3- unlock start							
36462	0-0- entry													
36843							6-3- exit							
36997													12-6- lock exit	
39182													12-6- unlock start	
39996													12-6- exit	
40158	0-0- lock exit													
40246					4-2- entry									
42277	0-0- unlock start													
43113	0-0- exit													
43164					4-2- lock exit									
45144				3-15- entry										
45489					4-2- unlock start									
47220					4-2- exit									
48033				3-15- lock exit										
50439				3-15- unlock start										
50842				3-15- exit										
65040			2-1- entry											
65714			2-1- lock exit											
68230			2-1- unlock start											
69176			2-1- exit											
137295		1-14- entry												
137969		1-14- lock exit												
140844		1-14- unlock start												
141350		1-14- exit												

Figure 28: Timeline when relative critical section size is 100

Figure 29: Timeline when relative critical section size is 10000

0						5-16- entry								
638						5-16- lock exit								
10142				3-15- entry										
15019						5-16- unlock start								
15781				3-15- lock exit										
15928						5-16- exit								
17659								7-17- entry						
19162							6-3- entry							
24809										9-18- entry				
26855									8-4- entry					
29656														13-20- entry
29854				3-15- unlock start										
30741												11-19- entry		
32457												11-19- lock exit		
33902											10-5- entry			
38434	0-0- entry													
39659		1-14- entry												
42137				3-15- exit										
44491			2-1- entry											
48085												11-19- unlock start		
52536		1-14- lock exit												
53387													12-6- entry	
61021												11-19- exit		
64049					4-2- entry									
66462		1-14- unlock start												
71544							6-3- lock exit							
80975		1-14- exit												
85939							6-3- unlock start							
88983										9-18- lock exit				
95407							6-3- exit							
103114										9-18- unlock start				
105673									8-4- lock exit					
113051										9-18- exit				
119746									8-4- unlock start					
121902														13-20- lock exit
128524									8-4- exit					
135997														13-20- unlock start
138608											10-5- lock exit			
143777														13-20- exit
152731											10-5- unlock start			
157293	0-0- lock exit													
161150											10-5- exit			
171218	0-0- unlock start													
181236	0-0- exit													
194400								7-17- lock exit						
215615								7-17- unlock start						
226615								7-17- exit						
237461													12-6- lock exit	
251526													12-6- unlock start	
260473													12-6- exit	
267116					4-2- lock exit									
281233					4-2- unlock start									
290649					4-2- exit									
297689			2-1- lock exit											
311849			2-1- unlock start											
312627			2-1- exit											

Figure 30: Timeline when relative critical section size is 1000