ABSTRACT

APSANGI, CHANDAN. Scalable Locks with Backoff Suspension for Manycore Systems. (Under the direction of Dr. Frank Mueller.)

A number of scalable locking mechanisms have been explored in the past to provide the best possible performance on many core systems. This includes various optimized versions of spin locks, ticket locks and queued locks. Of these, the queued lock variant proposed by Mellor Crummy and Scott(MCS) has been most widely adopted. In this paper we propose an enhanced version of MCS locks that aims to provide better scalability in terms of running time and saves useful CPU cycles and hence power consumption. Similar to MCS locks, scalability is provided by using a queue-based locking mechanism, where each thread spins on a locally cached variable. Savings in CPU cycles are achieved by means of a back-off mechanism, wherein the threads suspend after spinning for a predefined number of iterations. Experiments were conducted on a 64-core TilePro processor, running applications with 4 to 128 threads. Running time improvements in the order of 1.5x compared to existing locking mechanisms were noted as we scaled the number of threads as well as the number of cores. The results indicate scalability and power savings of this scheme and make it well suited for high-performance computing applications today and large scale main-stream many cores in the future.

Scalable Locks with Backoff Suspension for Manycore Systems

by
Chandan Apsangi

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2013

APPROVED BY:

Dr. George Rouskas

Dr. Xiaosong Ma

Dr. Frank Mueller
Chair of Advisory Committee

DEDICATION

To my wonderful parents and my most affectionate brother.

BIOGRAPHY

Chandan Apsangi was born and raised in Karnataka, India. After completing high school in Dharwad, he joined Sri Jayachamarajendra College of Engineering (SJCE), Mysore for a Bachelor's program in Information Science and Engineering. After working on a research project under the guidance of Dr. T.N.Nagabhushan at SJCE, he developed a greater passion to contribute back to the field of computer science. After obtaining the Bachelor's degree in 2008, with the intent of acquiring some industry experience, he joined Nokia India Pvt. Ltd., Bangalore as Mobile Software Engineer. After working for 3 years developing middleware software mostly on Symbian platform, his curiosity in System software drove him to pursue further studies in the Operating Systems area. In 2011, he joined North Carolina State University (NCSU) for a Master's program in Computer Science department. After an Internship with NetApp in the summer of 2012, he got an opportunity to pursue research under the guidance of Dr. Frank Mueller at NCSU. The focus of his research was on operating systems enhancements for many-core systems. He will be joining Intel in an Operating Systems Engineer role after completing his Master's degree.

ACKNOWLEDGEMENTS

In the retrospect, what seemed like a short and exciting time has been a pretty long and arduous journey. I can only credit all the wonderful people I have been surrounded by, for making this so simple. I would like to start by thanking my Parents who are the reason for my comfortable and adventurous life here. I am very grateful to you for instilling in me the values of hard work, honesty and generosity which made this a whole lot easier. I would like to thank my brother for being my personal and professional mentor since the day I was born. You are the classic definition of "leading by example" and I'm most grateful for all the support and encouragement you have provided all along.

I would like to thank my advisor Dr. Frank Mueller for his invaluable guidance and encouragement throughout my research work. I consider myself blessed to have been a witness to his incessant flow of ideas. It was really a great experience working with you Sir. I'd also like to thank my committee members – Dr. Rouskas and Dr.Ma, for taking time out of your busy schedule to support me through the defense. I would also like to thank my flatmates (present and past) and friends at Raleigh who'd alleviate any stress just by a few minutes of jokes and laughter. You folks are the prime reason why I'd cherish this place a lot. Special thanks to Dr.Chi and Dr.Adela Westedt from TU Berlin for providing access to Tilera-GX machine for the experiments. I would also like to thank National Science Foundation (NSF) for funding this project and encouraging me to give my best. In the short space available, I may have forgotten some important names. In such a case, I'd like to ask your forgiveness and sincerely hope that I will get another chance to thank you.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

A number of techniques for synchronization have been explored in the past. These can be broadly classified into two categories: 1) busy-waiting protocols and 2) blocking protocols. Busy waiting protocols perform well if either the length of the critical section is small or if there is no alternate work for the CPU until the lock is obtained. Test-and-set (T&S) locks implemented using atomics provide the basis for spin lock implementations. Anderson [1] deals with several different ways of optimizing T&S locks. The major performance degradation in case of T&S locks comes from the way the atomic operations are implemented at the hardware level. An atomic read operation invalidates the caches of all other processors (in case of invalidation based protocols). Since bus lines used for atomic updates are also used for general memory references, any general memory references may be slowed down as well due to contention. In order to mitigate this, spin lock alternatives such as test-and-test-and-set locks(TTS), ticket locks, etc. are considered. In case of TTS locks, when the lock is busy, the waiting threads spin on the locally cached variable without using atomics, i.e, without inducing network/bus transactions. When the lock is freed, the cache entry for the lock on all the spinning processors is invalidated, the new value is read-in, and each processor tries to atomically acquire the lock. Only one processor succeeds while all others continue to spin on their local cache variables. This helps reduce the network traffic substantially, but leads to what is known as "Thundering herd problem" [13]. Another problem associated with the pure spin implementation is that of Fairness. Since all the spinning processors are allowed to contend for the lock almost at the same time, the next lock holder is decided based on factors such as locality with respect to the unlocking processor, scheduling policies, cache coherence protocols, etc. Ticket locks [12] implemented in the Linux kernel fix this fairness problem. However, even with ticket locks it is not possible to get a constant bound on number of coherence transactions since each unlock (cache invalidation) takes O(P) transactions if P processors are waiting for the lock[2, 4]. This makes ticket locks unsuitable for high performance computing applications.

Considering these factors, Mellor-Crummey and Scott (MCS) [2] proposed a novel queuing based synchronization that addresses most of the problems discussed above. The MCS lock has the following properties: 1) It guarantees FIFO ordering of lock acquisition, i.e, ensures fairness. 2) Each thread spins on locally cached flags, i.e, it reduces network transactions. 3) A constant bound on network/bus transactions is established for unlocking. 4) A constant amount of space per lock is required. Mellor-Crummey and Scott [2] discuss the running time improvements provided by MCS locks as compared to other spin lock alternatives. They prove that the distributed nature of MCS locks provides better scalability on almost any architecture. We verified the same through our experiments where we compare MCS locks with backoff versions of T&S and TTS locks. As we scale the number of threads and number of cores, we note that in case of MCS locks, the increase in running time is directly proportional to the increase in the number of threads. In case of spin locks, the running times are very short for small number of threads but increases quickly for larger number of threads and cores.

Despite all these advantages, MCS locks still belong to the first category of locking protocols - busy waiting protocols. Considerable research has been done to highlight the drawbacks of busy-waiting protocols [15, 5]. For instance, the TurboBoost feature present in most modern Intel processors [14] implies that the total power draw of a chip is capped, i.e, divided between the active cores. Hence, threads using spin locks, while spinning, deprive the other cores of their potential performance boost without getting any real work done. Also, when multiple threads are scheduled per core, the CPU time is divided between the spinning thread and other active threads. Even though spinning threads are not doing any useful work, an almost equal amount of CPU time is dedicated to them. This also leads to higher context switches as our experiments demonstrate. To address these issues, we develop a novel locking protocol on the lines of MCS but in combination with the second category of synchronization mentioned above – blocking protocols. This algorithm (referred to as Backoff-MCS or B-MCS henceforth) involves suspension of the waiting thread after a few initial spins until the lock becomes available. This results in better performance especially for longer critical sections and where over-subscription (more than one thread per core) and barrier synchronization is involved, which is true about *Single program multiple data* (SPMD)-style programs. We argue that this combination of backoff and queued locks provides a bifold advantage. On the one hand, we get equal, if not better performance as MCS locks, and on the other, we conserve CPU cycles which may become available to other active threads or provide savings in power consumption. Our experiments show that, as we scale the number of cores and threads, in case of applications with longer critical sections, about 95% of the waiting time is spent in the backoff/suspended state. Through our experiments we reaffirm some of the problems with the traditional spin locks - fairness, quiescence and wastage of CPU cycles, and present how B-MCS addresses these problems. We also invalidate the context switch overhead argument against backoff based protocols and show that over-subscription can lead to higher context switching overhead in case of spin-only MCS locks as compared to Backoff-MCS locks.

Hypothesis

Considering the above discussed limitations of the exisiting locks, we aim to develop an enahnced version of MCS locks which would address these limitations. Hence the hypothesis of this work is: *The backoff-enhanced MCS locking algorithm provides a scalable and performant solution for reducing both lock contention and the number of context switches on large-scale multicore platforms that caters particularly to SPMD-style codes.*

Chapter 2

Related Work

Substantial research has been done in the past to identify the best possible synchronization solutions. There is no single best universal solution to all synchronization problems. Depending on various criteria such as computer architecture, application behaviour, application expectation (fast running time vs. high throughput), etc., some solutions look more attractive than others. Anderson et al.[1] deal with various enhancements for spin-based locks to achieve scalability. They emphasize the suitability and efficiency of spin-only locks for the uncontended case and backoff-based locks for the contended case. They highlight the scalability challenges posed by shared memory spin locks and propose a queue-based lock similar to Ticket locks in Linux. Our contribution is a novel queue-based lock with suspension of contending threads. In this paper, we highlight the situations where these locks would perform better and also the magnitude by which they do so compared to their predecessors.

Sandor et al.[6] experiment with existing spin locks (originally designed for multiprocessor systems) on multi-core systems. They conclude that with multi-cores, since the cost of core local memory is not significantly different from the global memory, both T&S and TTS show similar performance. However, under high contention and with large numbers of threads they recommend utilizing queue-based ticket locks as proposed by MCS [2]. Our work reaffirms the scalability problems inherent to T&S and TTS locks. We also present a comparison of our suspension-based queue locks with standard MCS locks and analyze the impact of suspension on the overall performance. This takes into account the time that threads remain suspended and the total number of context switches while providing similar running times as MCS locks. Boyd-Wickizer et al. [3] have shown that with a few modifications to the application benchmarks and the Linux kernel, existing OS mechanisms can scale well with many-core systems. This shifts the focus of research from enhancing existing OS mechanisms to developing new code around bottlenecks. In another contribution, Boyd-Wickizer et al.[4] have highlighted the impact of non-scalable locks on the overall system performance by developing a Markov model to explain why performance collapses. This, along with the Anderson's paper [1], serves as a motivation for our work on queue locks. In the Hierarchical CLH paper [9], Victor Luchangco and others design a new queue lock based on the CLH locks for CC-NUMA systems. David Dice and others propose a more robust design called "lock cohorting" [7], which involves passing on the lock ownership to the threads in the same NUMA cluster before releasing it for global contention. They employ a queue-based approach and rely on atomics for enqueuing onto the queues. In this paper, we design a similar solution based on atomic operations but for SMP systems, where a single queue per lock is maintained. Our implementation also provides the fairness and scalability that these locks generally promise.

Chapter 3

Design And Implementation

In order to evaluate our implementation in the light of existing algorithms, we have implemented three of the existing algorithms in C. Here, we present the pseudocode for the Backoff-T&S, Backoff-TTS and standard MCS locks. Later in this section, we present the implementation of Backoff-MCS and discuss the data structures and design decisions made.

3.1 Backoff-Test-and-set Locks

Algorithm 1 B-T&S Lock algorithm

```
procedure LOCK(thread_lock_t lock)
    size_t notdone
    repeat
        retry = 100
        repeat
            notdone = atomic_cas(lock.spin, 0, 1)
            retry = retry-1
        until (notdone = 0 or retry = 0)

        if (notdone) then
            notdone = atomic_cas(lock.spin, 0, 1)   ▷ Make another attempt to avoid the race
            if (notdone) then
                futex_down(lock.thread_futx)
            end if
        end if
    until notdone = 0
    return 0
end procedure
```

Algorithm 2 B-T&S Unlock algorithm

```
1: procedure UNLOCK(thread_lock_t lock)
2:     lock.spin = 0
3:     futex_up(lock.thread_futx)
4:     return 0
5: end procedure
```

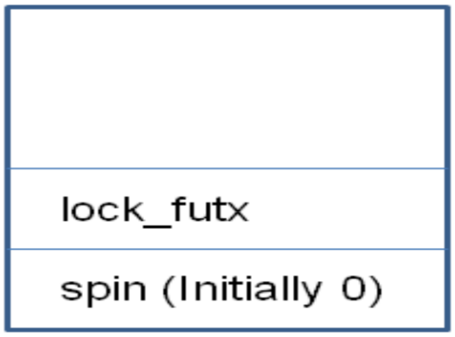

Figure 3.1: Lock Data structure for B-T&S

This is one of the most basic spin-lock algorithms. Our implementation uses an atomic compare_and_swap instruction (signature: *prev = atomic_cas(addr, old, new)*) for performing the test-and-set operation. We spin for 100 attempts to acquire the lock before suspending the thread. This speeds up the lock acquisition substantially when the size of critical section is small, as shown in the experiments later. The lock data structure used for test-and-set lock is shown in Figure 3.1. Here we have a *lock.spin* variable which is initialized to 0. The thread that manages to atomically set this to 1 gets the lock. The other variable that is used is *lock.lock_futx*. This is used to suspend the waiting threads. Note that all the threads are suspended on the lock futex. Hence in the unlock, all of them are woken up at once and contend for the lock.

3.2 Backoff-Test and Test-and-set Locks

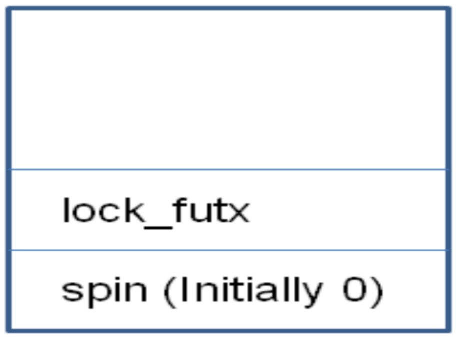

Figure 3.2: Lock Datastructure for B-TTS

This is similar to the test-and-set lock shown above except for an additional while loop in the lock code. This tests the value of the lock variable from the cache instead of making an atomic compare_and_swap each time. The unlock code remains the same. The lock data structure used for test and test-and-set lock is shown in Figure 3.2.

Algorithm 3 B-TTS Lock algorithm

```
procedure LOCK(thread_lock_t lock)
    size_t notdone
    repeat
        retry = 100
        repeat
            while (lock.spin = 1) do
                                                        ▷ Busy wait
            end while
            notdone = atomic_cas(lock.spin, 0, 1)
            retry = retry-1
        until (notdone = 0 or retry = 0)

        if (notdone) then
            notdone = atomic_cas(lock.spin, 0, 1) ▷ Make another attempt to avoid the race
            if (notdone) then
                futex_down(lock.thread_futx)
            end if
        end if
    until notdone = 0
    return 0
end procedure
```

Algorithm 4 B-TTS Unlock algorithm

```
procedure UNLOCK(thread_lock_t lock)
    lock.spin = 0
    futex_up(lock.thread_futx)
    return 0
end procedure
```

3.3 MCS Locks

MCS locks are one of the most widely discussed locking mechanisms. The implementation shown here is based on the original MCS paper [2]. Figure 3.3 shows the lock datastructure used by our implementation of MCS locks. It demonstrates how a queue of waiters can be constructed by means of the *tail* pointer in the lock datastructure and *next* pointer in the thread_t datastructure. The top half of the Figure 3.3 shows the state of the lock data structures initially. The bottom half is the state when two threads have queued up on the lock.

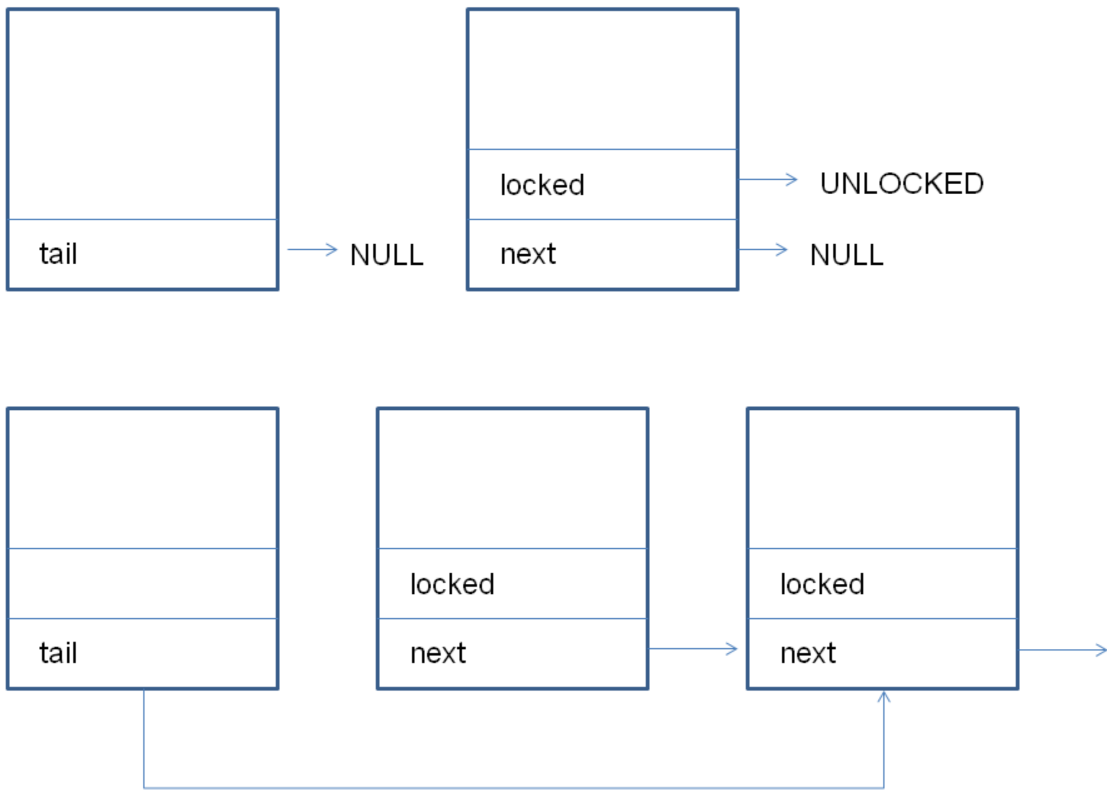

Figure 3.3: Lock Datastructure for MCS

Algorithm 5 MCS Lock algorithm

```
procedure LOCK(thread_lock_t lock, thread_t current)
    current.locked = UNLOCKED
    current.next = 0
    repeat
        opred = lock.tail
        pred = atomic_cas(lock.tail, opred, current)
    until pred = opred

    if pred != 0 then
        current.locked = LOCKED
        pred.next = current
        while (current.locked = LOCKED) do
                                                        ▷ Busy wait
        end while
    end if
    return 0
end procedure
```

Algorithm 6 MCS Unlock algorithm

```
procedure UNLOCK(thread_lock_t lock, thread_t current)
    if current.next = 0 then
        if atomic_cas(lock.tail, current, 0) = current then
            current.next = 0
            return 0
        end if

        while (current.next = 0) do
                                                        ▷ Busy wait
        end while
    end if

    current.next.locked = UNLOCKED
    current.next = 0
    return 0
end procedure
```

3.4 Backoff-MCS Locks

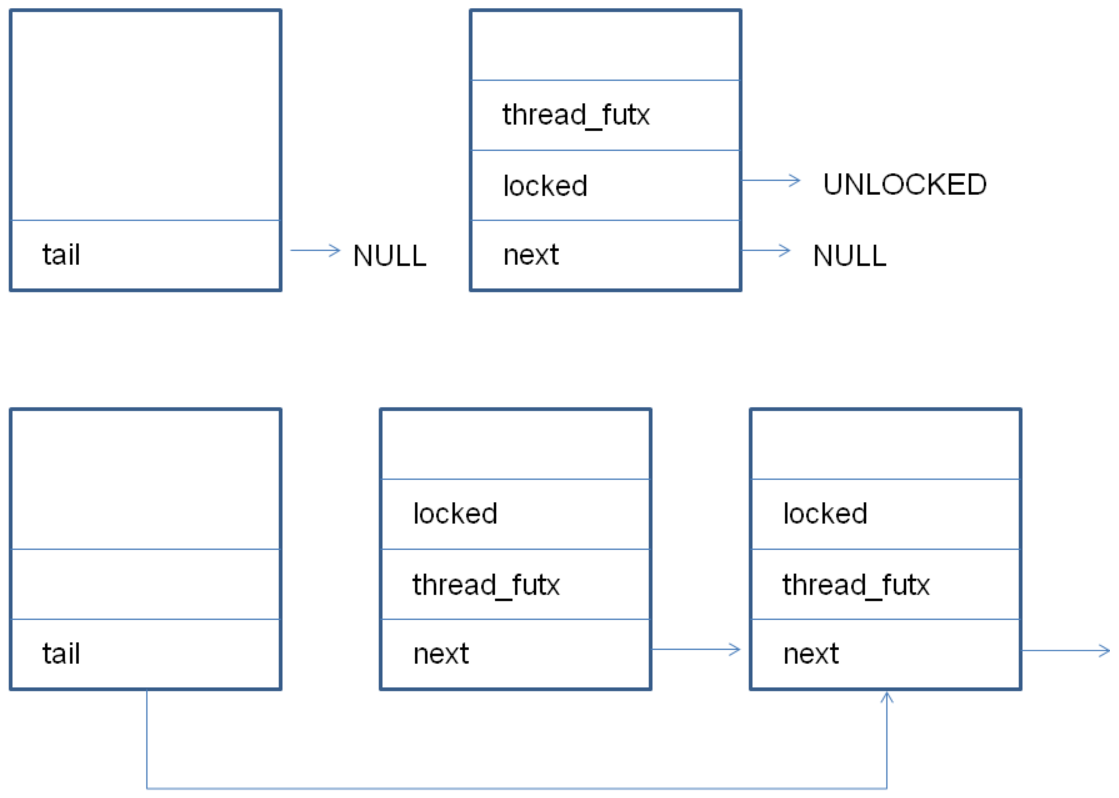

Figure 3.4: Lock Data structure for B-MCS

Just as MCS locks, we rely on atomic operations to update a queue of waiters. We use atomic compare_and_swap to check the status of the queue and to enqueue the waiters. We make use of two opaque data structures: 1) The lock data structure – thread_lock_t, a pointer passed as an argument to the lock() and unlock() functions. The user instantiates this once, and may use it for synchronization multiple times. 2) Per thread data structure – thread_t, which is used to enqueue a thread onto the lock queue (The user is unaware of the existence of this data structure). We maintain a state variable (called *locked*) in thread_t data structure, which can take any of the three states: LOCKED, UNLOCKED and BACKOFF. The BACKOFF state indicates that the thread has suspended itself and needs to be woken up. We use futex APIs [11, 10] provided by the Linux Kernel to implement thread blocking and unblocking. Figure 3.4

shows the state of these two data structures for different cases. The lock data structure has a tail pointer which enables the requesting threads to enqueue onto the lock queue. The per thread data structure contains a struct futex pointer (shown as *thread_futx*), which is used to block/unblock a thread. When the lock is free, the tail pointer is set to NULL.

Algorithm 7 B-MCS Lock algorithm

```
procedure LOCK(thread_lock_t lock)
    thread_t current = thread_self()
    repeat
        opred = lock.tail
        pred = atomic_cas(lock.tail, opred, current)
    until pred = opred

    if pred != 0 then
        current.locked = LOCKED
        pred.next = current

        retry = 100
        while (current.locked = LOCKED) & (retry > 0) do
            retry = retry-1
        end while

        if (atomic_cas(current.locked, LOCKED, BACKOFF)
                        = LOCKED) & (retry = 0) then
            futex_down(current.thread_futx)
        end if
    end if
    return 0
end procedure
```

The psuedocode for the lock() operation is shown in Algorithm 7. Each thread attempts to add itself to the *tail* of the lock atomically using a compare_and_swap instruction. Only the thread that finds the previous value of the *tail* to be NULL obtains the lock. This is similar to MCS lock implementation. In case of contention (non-null *tail* pointer), we update the predecessors *next* pointer to point to the current thread by means of the *next* pointer of the thread_t data structure. Note that the user simply calls lock() with the thread_lock_t parameter. The creation and enqueuing on the per thread data structure is transparent to the user. After a few iterations (100 in the above algorithm), the thread atomically sets its state to BACKOFF and transitions to the suspended state using a futex_down() call.

Algorithm 8 B-MCS Unlock algorithm

```
procedure Unlock(thread_lock_t lock)
    thread_t next
    thread_t current = thread_self()
    if atomic_cas(lock.tail,current,0) = current then
        current.next = 0
        return 0
    end if
    while current.next = 0 do
                                                        ▷ busy wait
    end while

    next = current.next
    current.next = 0;
    if atomic_cas(next.locked, LOCKED, UNLOCKED)
                        = BACKOFF then
        next.locked = UNLOCKED;
        futex_up(next.thread_futx);
    end if
    return 0
end procedure
```

The psuedocode for the unlock() operation is shown in Algorithm 8 . We try to atomically set the lock.tail to NULL. If some other thread added itself before this instruction, then we have to wait for it to update the current.next and hence the subsequent while loop. Then we consider the current.next as the next thread to be granted the lock. We update current.next to NULL and unlock/wake up the immediate waiter.

Chapter 4

Empirical Study

4.1 Experimental Setup

We conducted our experiments on a Tilera board with 64 cores under Linux (2.6.x kernel). Cores are arranged in a 2D array of compute engines (cores) called tiles. Each tile consists of a complete, full-featured processor with a frequency of 700MHz to 866MHz. Each tile has a physically tagged split L1 cache and a shared L2 cache. Tilera uses a directory based cache coherence protocol. One of the tiles is dedicated to I/O handling and not available for our experimentation. Hence, we have a maximum of 63 tiles at our disposal.

4.2 Results

We compare and contrast the performance of our implementation (B-MCS locks) with the standard MCS implementation, a backoff test-and-set lock (B-T&S) and a backoff test-and-test-and-set lock (B-TTS). We have developed synthetic benchmarks consisting of 10 iterations of lock-unlock operations with the above mentioned algorithms. The benchmarks circumvent the effects of core relocation issues by pinning the threads to cores as they are created. To warm up the caches, each thread initially runs 40 iterations of a lock-unlock sequence before starting the actual measurements. In addition, these extra iterations help us control the adverse effects of the scheduler, which would potentially run the threads (and grant locks) as soon as they are created, instead of a fair competition with each other to acquire the locks. We also ensure that the data structures are padded and aligned with cache lines so that the false sharing of cache lines does not affect the results.

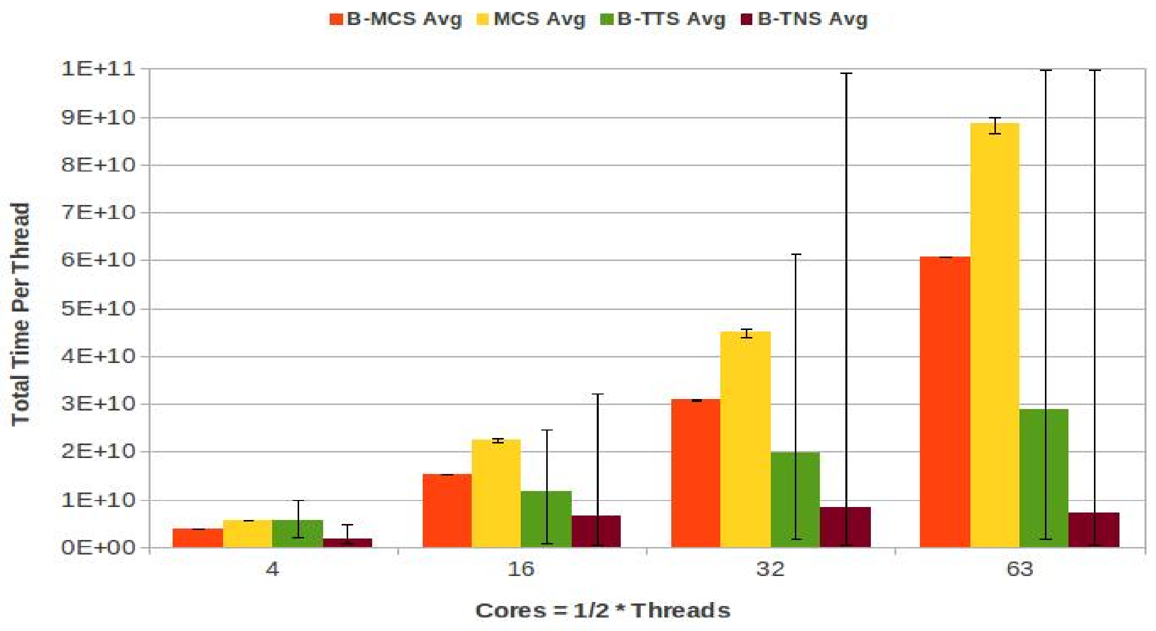

Figure 4.1: Average time taken when we do over-subscription. Also shows the minimum and maximum time taken.

4.2.1 Running time

In the first set of experiments we measure the per thread running time for 10 iterations of the lock-unlock sequence for each of the algorithms. In order to analyze the advantages of queued variants over the spin variants, we first consider the case of over-subscription where more than one thread is mapped to a single core. Figure 4.1 shows the average running time on the y axis along with the minimum and maximum variations (error bars) for different number of cores on the x axis, when two threads are assigned to a single core. We notice that even with large numbers of threads and cores (63 cores in our case) B-T&S and B-TTS outperform B-MCS and MCS in terms of average running time. This is due to the small size of the critical section, and the resulting overhead of queuing and suspension in case of queued locks, which adversely affects the overall running time. But if we look at the minimum and maximum running times, the queued algorithms show very little variation as opposed to the spin counterparts. In case of B-T&S and B-TTS, some threads will complete all 10 iterations much faster than others depending on the scheduler. But in case of MCS and B-MCS, the lock is granted in a first-come-first-serve (FCFS) manner. Hence, all threads complete each iteration at nearly the same time.

Anderson et al.[1] tune the spin lock performance by introducing a delay 1) at the beginning

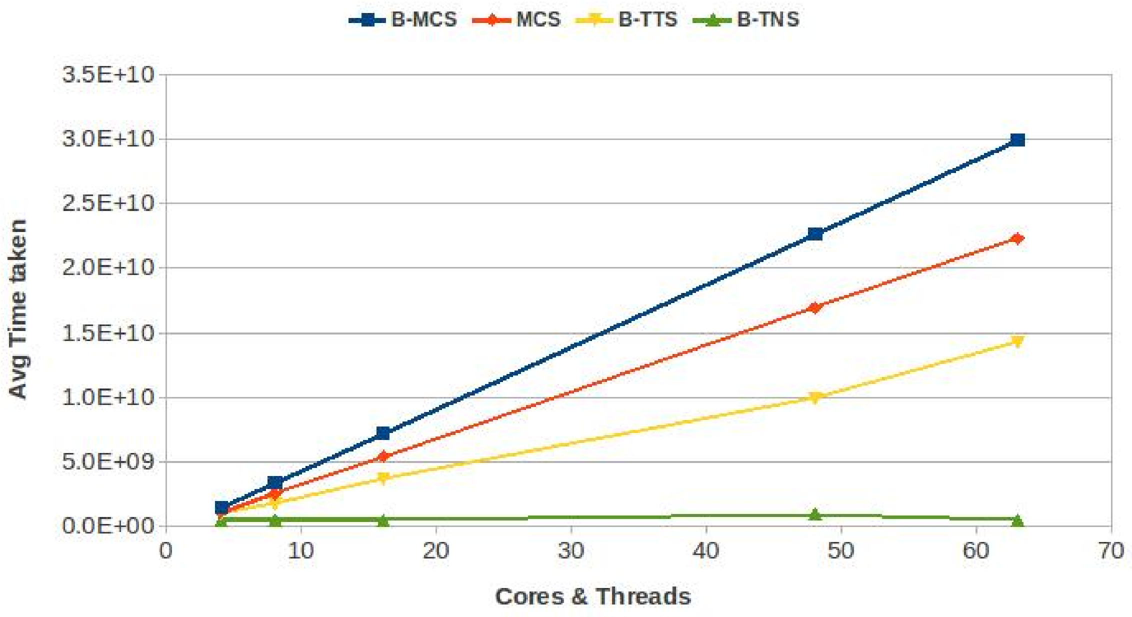

Figure 4.2: Average time taken when we have a single thread per core.

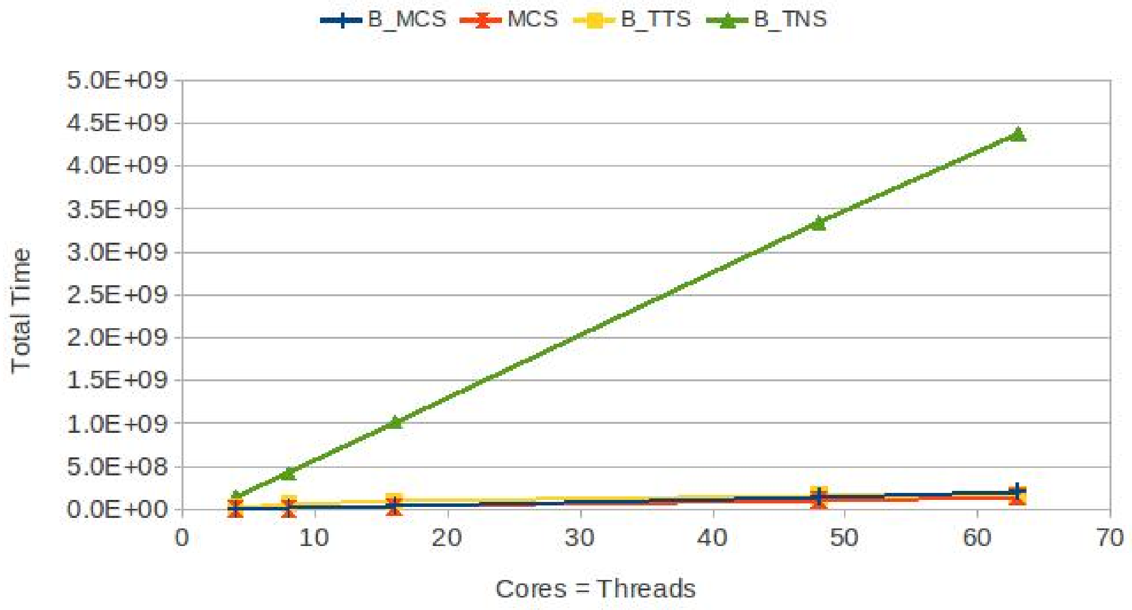

Figure 4.3: Average time taken when we have a single thread per core with a Barrier. There's no code in critical section.

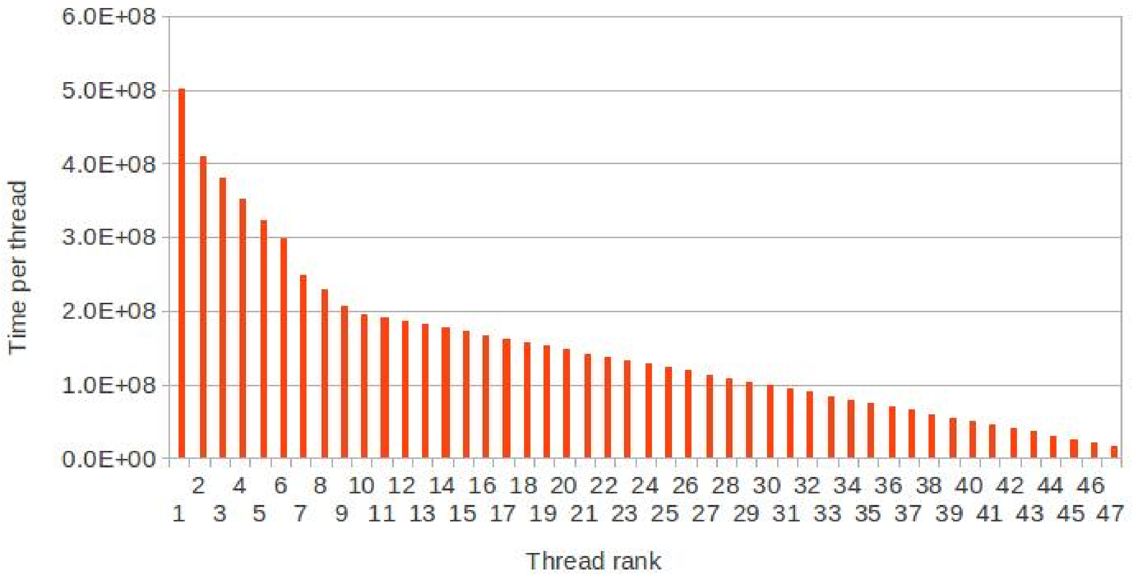

Figure 4.4: Time taken by each thread when using B-MCS with one thread per core along with a barrier.

of the critical section to wait for contending threads to quiesce before beginning the critical section and 2) at the end of each test-and-set call to prevent all threads from executing test-and-set at the same time, which would reduce the flooding of the bus. We consider a third case where we introduce a barrier at the end of each lock/critical section/unlock sequence. This is motivated by the BSP/SPMD paradigm of parallelizing codes, which relies on computational balance to achieve high processor utilization and best overall system performance. A barrier introduces fairness in case of B-T&S and B-TTS, as no thread is allowed to proceed to the next iteration of the lock-unlock sequence until all the threads finish the current iteration. Figure 4.3 shows the results. This experiment highlights the quiescence effect described by Anderson et al. [1]. The Quiescence effect is seen in case of B-T&S because of the large number of atomic operations and the associated bus transactions (cache invalidations). The number of invalidations performed is proportional to the number of threads spinning to acquire the lock. This is not seen in case where the barrier is not present (Figure 4.2) because the threads are allocated locks randomly and, hence, some threads finish all of their iterations while some others are yet to start. Hence, the number of invalidations would be drastically reduced. This effect is not seen in case of B-MCS and MCS because they involve invalidating only the cache of next thread in the queue without interfering with other threads. B-TTS also performs well because of the in-cache tests performed after a failure to acquire the lock. Another interesting result of this experiment is shown in Figure 4.4. In case of MCS and B-MCS the total time taken per thread decreases as we move towards the trailing end of thread queue. This can be attributed to the fact that in case of MCS and B-MCS an atomic compare_and_swap is used to enqueue onto the lock queue. For the first few threads this contention is higher than for the trailing threads. Our test case features empty critical sections. But if the critical section contained memory accesses, then due to bus contention between threads performance would degrade significantly.

4.2.2 Fairness

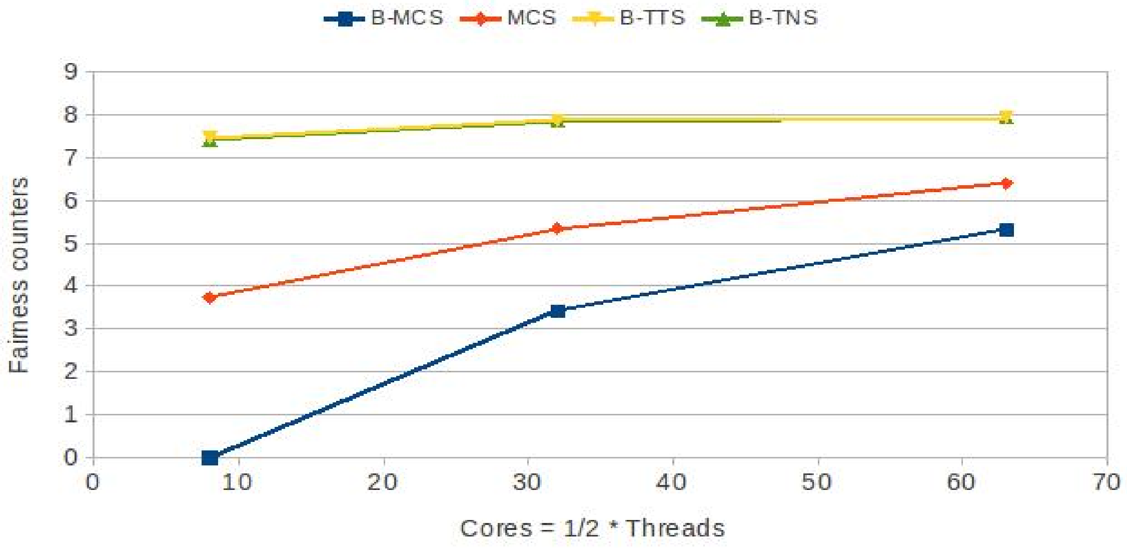

Figure 4.5: Average values of fairness counter when a two threads are assigned to each core.

The fairness guarantee provided by queued locks has been well established in past work [2][4]. Here, we reaffirm this using per-thread "fairness counters", which are incremented when a thread lags behind any other thread by more than one iteration. Figure 4.5 shows the results for 10 iterations when oversubscribed. Due to random lock allocation in case of B-T&S and B-TTS, we consistently observe an average lag of 7 to 8 iterations out of 10. This means that on an average some threads lag behind the others by 70 to 80%. This value is much lower (30-50%) in case of MCS and B-MCS confirming the fairness guarantee.

4.2.3 Core-resident execution time

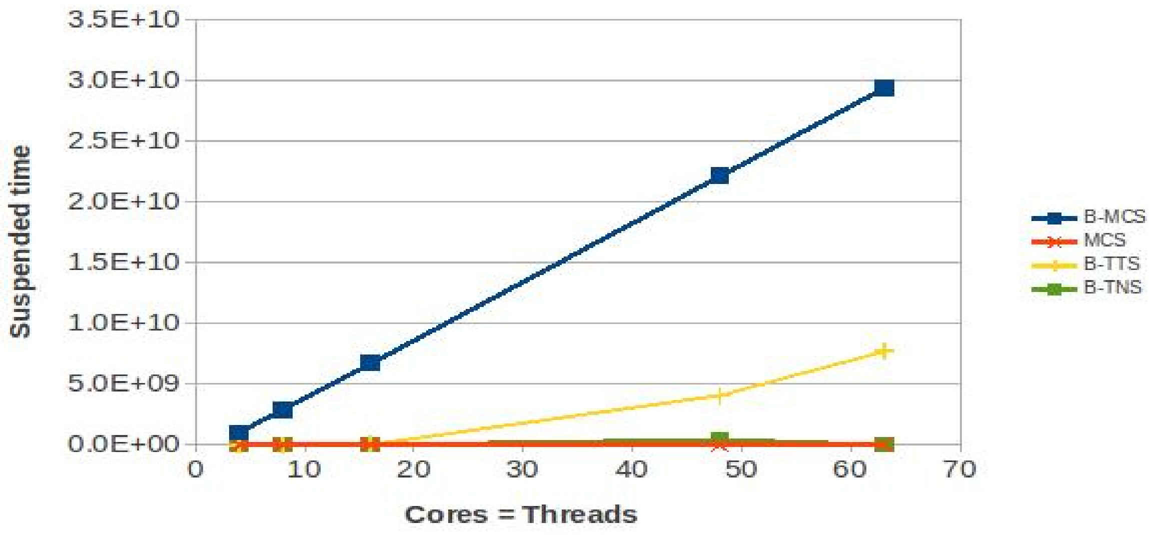

Figure 4.6: Average time suspended when a single thread is assigned to each core.

In another set of experiments, we analyze the tradeoff between the overhead of queuing plus context switching on one hand versus the fairness plus the savings in CPU cycles on the other, as seen in the backoff algorithms. As already noted in Figure 4.2, B-MCS is the slowest of the four algorithms. Figure 4.6 depicts the average amount of time threads are suspended. This fraction is zero for MCS, since it does not involve any backoff. It is also small for B-TNS. By comparing Figure 4.2 and Figure 4.6, we observe that in case of B-MCS, threads spend a majority of their time in the suspended state. Hence, we assess another metric to model the performance of the algorithms, namely Core-resident execution time, which represents the time that the threads are actually running on the cores. Figure 4.7 shows the core-resident execution time measured as the difference between total time and suspended time. Because of the large percentage of suspension time, B-MCS has almost exactly the same core resident execution time as B-TNS. In case of B-MCS, even though the overall time is larger, the core-resident time is much less, meaning that for a large part the cores remain available for execution of other background work. In this respect, MCS fares very poorly. Hence, when comparing various queued lock algorithms, we can consider core resident execution time as an additional metric to enable high throughput along with scalability and fairness capabilities of the queued locks.

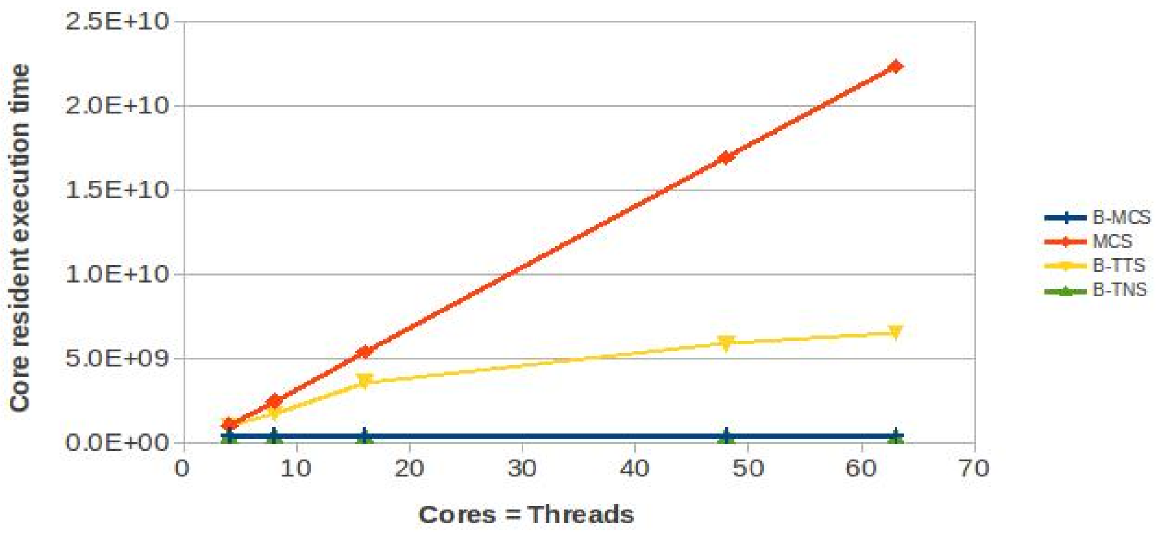

Figure 4.7: Average Core resident execution time when a single thread is assigned to each core.

4.2.4 Context switch overhead

Context switch overhead has been considered as a major deterrent in adopting backoff-based locking protocols [8]. Even in our implementation of the B-MCS protocol, context switch overhead along with the queuing overhead seem to be the major hindrances for achieving short run times. In another experiment, we measured the number of context switches performed during the benchmark run for different algorithms. Figure 4.8 and Figure 4.9 show the total number of context switches due to self-suspension and due to preemption, respectively. We have reverted back to using over-subscription experiments in order to observe the behavior of non-backoff (in our case MCS) algorithms when multiple ready tasks are available on each core. Here, 2 threads are pinned to each core as we scale the number of cores. As can be seen, the number of self-suspensions (due to futex system calls) is about 6000 in the worst case for B-MCS. This is true for B-TNS as well, since for the given benchmark the size of the critical section is causing more threads to suspend. However, note the number of preemptions in case of MCS, which is close to 250,000. For B-MCS the number of preemptions is just in the hundreds as threads voluntarily relinquish cores. This suggests that when oversubscribed, it is best to utilize B-MCS as the context switch overhead is 1000s of orders lower compared to MCS and it is almost the same as B-T&S.

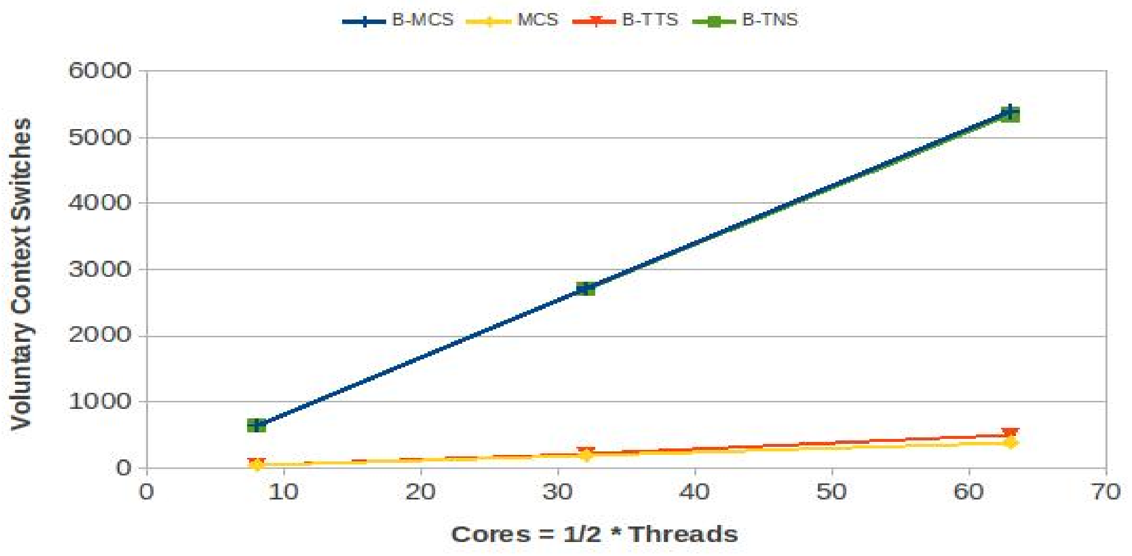

Figure 4.8: Total number of Voluntary context switches when over-subscription is involved.

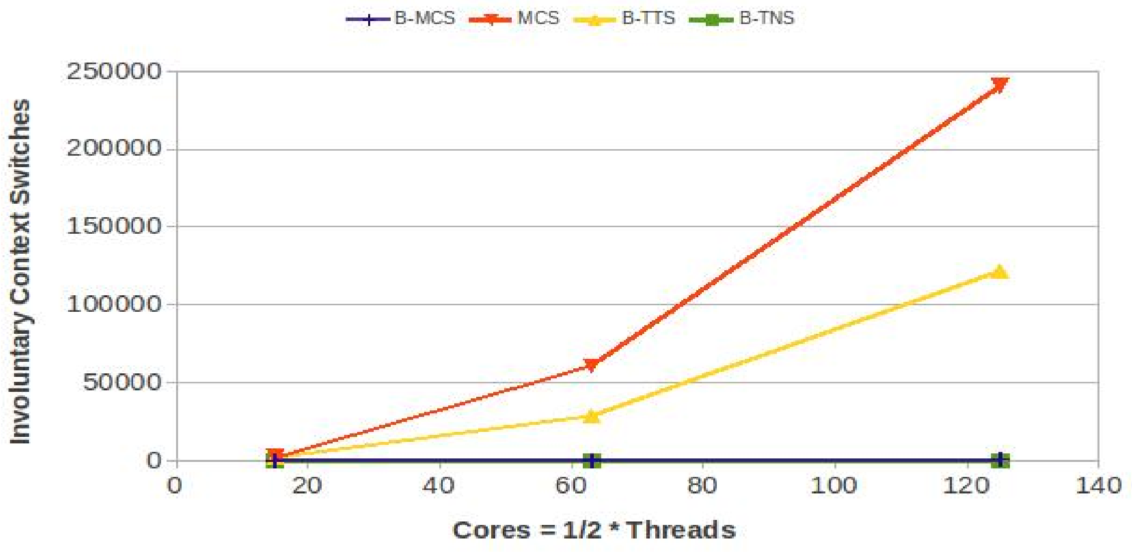

Figure 4.9: Total number of Involuntary context switches when over-subscription is involved.

Chapter 5

Conclusions and Future work

Conclusion

Our experiments confirm many of the established results about spin locks and queued locks. With barrier experiments and fairness counters we confirmed the impact of fairness and quiescence effect on the running times. In order to better evaluate queued lock algorithms we proposed a new metric, namely core-resident execution time, to account for the saved CPU cycles in case of backoff-based algorithms. This metric provides a holistic measurement of performance including throughput and saved CPU cycles, which helps reduce power and assesses the ability to accommodate background jobs effectively. The over-subscription experiments indicated a slow down of spin-based algorithms, especially MCS locks (Figure 4.1). We also highlight the dramatic increase in context switch overhead of spin-only locks when oversubscribed. Our experiments confirm the Thesis statement, namely that the backoff-enhanced MCS locking algorithm provides a scalable and performant solution for reducing both lock contention and the number of context switches on large-scale multicore platforms that caters particularly to SPMD-style codes.

Future Work

Our experiments involve synthetic benchmarks which do not always simulate the real world applications. It would be interesting to see how real world benchmarks (especially when oversubscribed) would perform with B-MCS locks. We could also quantify the percentage of power savings this scheme proposes to achieve. This should be substantial, since threads using B-MCS spend a majority of time being suspended while the lock is busy. As pointed out, another way to utilize the savings in CPU cycles would be to schedule background job on the idle cores. It would be interesting to note how the quiescence effect (seen B-T&S) impacts the running time of background tasks and if it is alleviated in case of B-MCS.

REFERENCES

[1] Thomas E Anderson, *Performance of Spin lock alternatives for shared memory multiprocessors*, IEEE Transactions on parallel and distributed systems, 1990.

[2] M. Crummey and Scott, *Algorithms for scalable synchronization for shared memory multiprocessors*, ACM Transactions on Computer Systems, 1991.

[3] S. Boyd-Wickizer, F. Kaashoek, R. Morris and N. Zeldovich, et al., *Analysis of Linux Scalability to Many Cores*, Proceedings of the 9th USENIX conference on Operating systems design and implementation, 2010.

[4] S. Boyd-Wickizer, F. Kaashoek, R. Morris and N. Zeldovich, *Non-scalable locks are dangerous*, Linux Symposium, Ottawa, Canada, 2012.

[5] Vladimir Cakarevic, Petar Radojkovic, Javier Verdu, Alejandro Pajuelo, et al., *Understanding the overhead of the spin-lock loop in CMT architectures*, 2008.

[6] Sandor Juhasz, Akos Dudas, Tamas Schradi, *Cost of mutual exclusion with spin locks on multi-core CPUs*, Proceedings of the 1st international conference on Biologically Inspired Computation(BICA), 2012.

[7] David Dice, Virendra J. Marathe, Nir Shavit, *Lock cohorting: a general technique for designing NUMA locks*, Proceedings of the 17th ACM SIGPLAN symposium on PPoPP, 2012.

[8] Ryan Johnson, Manos Athanassoulis, Radu Stoica, Anastasia Ailamaki, *A new look at the roles of spinning and blocking*, DaMoN, 2009.

[9] Victor Luchangco, Dan Nussbaum, Nir Shavit, *A hierarchical CLH queue lock*, Proceedings of the 12th international conference on Parallel Processing, 2006.

[10] Ulrich Drepper, *Futexes are Tricky*, 2011.

[11] Franke, Russell and Kirkwood, *Fuss, Futexes and Furwocks: Fast User level Locking*, Proceedings of 2002 Ottawa Linux Summit, 2002.

[12] J Corbet, *http://lwn.net/Articles/267968/*, LWN.net.

[13] *Thundering Herd Problem*, http://en.wikipedia.org/wiki/Thundering_herd_problem.

[14] *Intel TurboBoost*, http://en.wikipedia.org/wiki/Intel_Turbo_Boost.

[15] Bruce Dawson, *http://www.gamasutra.com/view/news/172072/*, 2012.