Implicit Acceleration of Critical Sections via Unsuccessful Speculation *

Joseph Izraelevitz
University of Rochester
jhi1@cs.rochester.edu

Alex Kogan
Oracle Labs
alex.kogan@oracle.com

Yossi Lev
Oracle Labs
yossi.lev@oracle.com

Contact author: Joseph Izraelevitz, jhi1@cs.rochester.edu. Department of Computer Science, University of Rochester, Rochester, NY 14627-0226, USA. +1-585-275-2527.

Abstract

The speculative execution of critical sections, whether done using HTM via the transactional lock elision pattern or using a software solution such as STM or a sequence lock, has the potential to improve software performance with minimal programmer effort. The technique improves performance by allowing critical sections to proceed in parallel as long as they do not conflict at run time. In this work we experimented with software speculative executions of critical sections on the STAMP benchmark suite and found that such speculative executions can improve overall performance even when they are *unsuccessful* — and, in fact, even when they *cannot* succeed.

Our investigation used the Oracle Adaptive Lock Elision (ALE) library which supports the integration of multiple speculative execution methods (in hardware and in software). This software suite collects extensive performance statistics; these statistics shed light on the interaction between these speculative execution methods and their effect on performance. Inspection of these statistics revealed that unsuccessful speculative executions can accelerate the performance of the program for two reasons: they can significantly reduce the time the lock is held in the subsequent non-speculative execution of the critical section by prefetching memory needed for that execution; additionally, they affect the interleaving between threads trying to acquire the lock, thus serving as a back-off and fairness mechanism. This paper describes our investigation and demonstrates how these factors affect the behavior of multiple STAMP benchmarks.

1. Introduction

The *speculative execution* of critical sections (also called *lock elision*) allows threads to execute critical sections concurrently as long as these executions do not conflict, and rolls them back otherwise (see Figure 1). Running critical sections speculatively can often accelerate parallel programs, especially those that use coarse grained locking, unnecessarily serializing executions of critical sections that almost never conflict. However, as we show in this work, speculative execution of critical sections can speed up programs even when the speculation never succeeds (e.g., because all critical sections conflict with each other on a shared memory access).

1.1 Speculation Techniques and the ALE library

Critical section speculation recently gained more popularity due to the introduction of the Hardware Transactional Memory (HTM) feature in commercially available processors [12, 13, 14]. Indeed, HTM can be used for critical section speculation with the Transactional Lock Elision (TLE) [19] technique. With TLE, a hardware transaction checks that the lock is not held, and then executes the critical section's code. Any conflict with a memory access done by another thread, including an acquisition of the lock, will cause the hardware transaction to fail (abort) and roll back the critical section's execution. On the other hand, if the hardware transaction commits successfully, the critical section execution has completed without needing to acquire the lock.

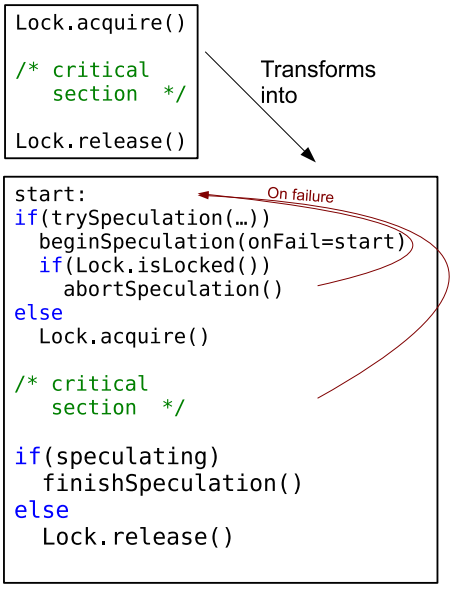

Figure 1: Lock Elision Transformation

Critical section speculation can also be done using software only techniques; we denote these with the general term "software optimistic techniques" (SWOPT) [9]. Examples of SWOPT techniques are software transactional memory mechanisms (STMs) [20], and sequence locks. STM solutions track all memory accesses, at some granularity, in order to detect conflict and roll back the transaction execution. Sequence locks [6], on the other hand, only allow threads to run speculatively as long as they do not execute any store to a shared memory location (see Figure 2). Once a thread needs to execute a write, it has to acquire the lock and increment a sequence number associated with it. Threads that are executing speculatively need to keep checking the sequence number, making sure that it has not changed since the beginning of the speculative execution. If they find a change, they abort their execution. Relative to HTM, sequence locks significantly limit the potential

* This work was supported in part by NSF Grants CCF-1422649 and CCF-1422649.

Lock Pass

```
Lock.acquire()
seqLock++
// seqLock odd

/* critical
   section  */

seqLock++
Lock.release()
// seqLock even
```

SWOPT (SeqLock) Pass

```
start:
v = seqLock
if(v%2!=0){goto start}

/* critical section  ... */
x = SHARED_READ(m)
if(v!=seqLock){goto start}
y = SHARED_READ(n)
if(v!=seqLock){goto start}
```

Figure 2: Sequence Lock SWOPT

parallelism between critical section executions, but they have relatively very low overhead — sometimes even lower than that of hardware transactions [9] — and require no special hardware. As sequence lock SWOPT was the only SWOPT technique we used in this work, we will refer to it for the remainder of this paper simply as SWOPT.

Speculative methods for lock elision share characteristic strengths and weaknesses. They all allow critical sections that are protected by the same lock to execute in parallel as long as it is guaranteed that no conflicts occurred. However, most speculative techniques may abort the execution even when a conflict did not occur: both SWOPT and HTM based solutions only detect conflict at some granularity (and hence may abort the execution due to a false conflict). The techniques may also abort for non–conflict–related reasons when the thread takes unsupported actions (e.g. I/O for HTM or a shared write for SWOPT). Additionally, neither HTM nor SWOPT guarantees that the critical section will complete under the speculative execution. The existence of true conflicts or the existence of unsupported actions may prevent progress indefinitely, necessitating "falling back" to a lock. Finally, care must be taken by the programmer to ensure that all mutual exclusion techniques actually mutually exclude one another.

Oracle's Adaptive Lock Elision library [9] addresses these shared properties and concerns by supporting lock elision techniques for existing lock–based software via limited instrumentation. In particular, the library handles the details of supporting mutual exclusion across several techniques: TLE using HTM, user provided software speculation techniques, and locks. The library also addresses the issue of strategy: which speculation techniques should be used for which critical section? And how many times should the techniques be tried before giving up and falling to the lock? More relevant for our purposes, ALE tracks a number of statistics for instrumented code; for each critical section, ALE records total number of executions, number of attempted speculative executions and their methods, average time for execution, average time waiting for the lock, among others.

We adapted ALE for use on STAMP [18], a well known software transactional memory benchmark suite that ports several real world programs into a transactional paradigm. By using ALE's statistics, we were able to study when speculative execution is useful for STAMP benchmarks and why. In particular, our exploration revealed that even benchmarks in which the speculative path always fail may be able to benefit from these failed speculative executions, due to both cache warm-up effects and a changing contention pattern on the fall-back lock.

While our exploration included experiments with and without TLE, the phenomena of acceleration via failed SWOPT attempts was most dominant in the experiments that did not involve HTM, and used sequence locks as the SWOPT method of choice. For the rest of the paper we will therefore focus on these experiments, and on the analysis that led us to these observations.

Implicit acceleration via failed speculative executions results from two base causes. First, the doomed speculative pass prefetches data that will be used in the future once the lock is acquired. This prefetch effect will help if the lock remains held for the duration (the lock is sufficiently saturated). Otherwise, the prefetch effect intrudes on the critical path, slowing execution. The prefetch pass must also abort before prefetching too much data, otherwise it might begin to evict cache lines needed at the critical section's start, effectively negating the gains.

Second, implicit acceleration occurs because the speculative executions act as a back–off technique for contended locks. In this case, speculative attempts can reduce contention on locks even if they do no useful work (e.g. prefetching). Speculative attempts also increase fairness in critical section execution across threads since threads that attempt speculative passes release locks for longer periods. Depending on the application, fairness can improve or seriously hamper performance.

1.2 Related Work

Our work is part of the continuing effort to identify and quantify the emergent effects of parallelism in shared memory software. Well known examples include the convoy effect [1], false sharing [2, 11, 21], and busy-wait cache line contention [17]. More recent examples include destructive interference across threads via cache contention [3] and cache residency imbalance [8].

Speculation as prefetching has been documented in other work. Sun's Rock processor's simultaneous speculative threading (SST) [5] uses a hardware scout thread [4] to speculatively warm the cache ahead of the trailing fully specified thread. Xiang and Scott note that speculative execution of critical sections in hardware transactional memory can warm the cache [22] for later executions under locks; this observation was also made by both Dice et. al. and Kleen [7, 15].

2. Benchmark Integration

We chose STAMP [18] as a candidate for lock elision because its concurrent structure is well annotated with macros, easing integration with ALE.

STAMP is a benchmark suite designed for software transactional memory research. It annotates transaction begin and end, as well as shared reads and writes with macros. The included benchmarks are real world software adapted to use transactions from a variety of disciplines and exhibit a wide variety of structural parallelism behavior. STAMP is a popular benchmark suite to evaluate STM implementations [10], and was recently used to evaluate HTM lock elision [16, 23].

2.1 Implementation

As a transactional memory benchmark suite, STAMP does not use any locks, but conceptually has a single global lock on all shared state. Our experiments in lock elision effectively elided this global

lock using sequence lock SWOPT, and used a global pthreads lock for fall back. For integration between ALE and STAMP, our critical section speculation implemented definitions for STAMP's transaction macros: `BEGIN_TXN`, `END_TXN`, `SHARED_READ`, and `SHARED_WRITE`.

SWOPT Our sequence lock SWOPT pass was automatically generated using the STAMP macros. On lock acquire, we read the current sequence lock and verify that it is not held. We call `setjmp` to reserve our execution context, then enter the critical section. Using STAMP's `SHARED_READ`, we instrument each read to verify that the sequence lock has not changed. On a `SHARED_WRITE` or verification failure, we abort by calling `longjmp`.

Lock Unless otherwise specified, we used the default pthreads mutex as the global lock. We chose this lock as it is widely used and available implementation. Its performance with regards to contention is roughly equivalent to a simple test-and-set lock. For compatibility with speculative passes, acquisitions and releases of the global lock also increment the sequence lock. For some experiments, we also compared against the MCS lock [17], which serves waiting threads in a FIFO order.

2.2 ALE Policies

Even though ALE supports dynamic policies that adapt to the platform and workload, in this work we mainly concentrate on simpler static policies. In general we used static speculation policies, that is, for every critical section we would execute a predecided set of speculative actions, eventually falling to the lock. The policies we will talk about most here are:

1. SWOPTLock: Attempt to execute the critical section in sequence lock SWOPT a set number of times (by default 10), then fall to the global pthreads lock. If a speculative execution encounters a write, immediately fall back to the global pthreads lock.
2. LockOnly: Never attempt a speculative action, but rather only use the global lock for mutual exclusion. The LockOnly policy was used with both the standard pthreads lock and the MCS lock [17].

2.3 Experimental Setup

All of our tests were done on a single socket Intel Haswell Core i7-4770 containing four cores with two way hyperthreading (supports eight hardware threads). Each core has its own L1 and L2 cache, with the L3 cache being shared across the entire processor. For all tests, turbo mode was turned on. The machine is powered by Oracle Linux 7. Tests were run while we were the sole users of the machine.

STAMP and ALE are written in C and C++ respectively. All experiments were done using the g++ 4.8.1 compiler with the `-O3` flag enabled.

3. Speculation as Prefetching

One of the first interesting effects we noted was that failed speculative executions can act as software prefetchers for later successful attempts. While other authors have noted this effect [5, 22], to our knowledge none have investigated the use of software speculation for prefetching.

3.1 Prefetching benefits

The prefetching effect of speculative SWOPT executions can be so strong as to significantly improve run time performance and scalability of the SWOPTLock policy over LockOnly, **even** when speculative executions rarely or never succeed. The clearest example of

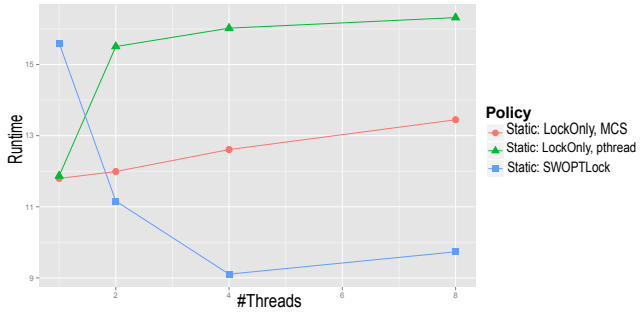

Figure 3: Runtime on *vacationlo*as a function of thread count

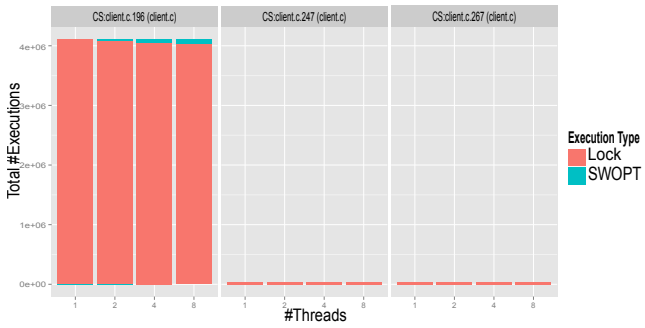

Figure 4: Policy of successful execution for critical sections in *vacationlo*

this behavior is on the *vacationlo* benchmark, where failed speculation improves performance by 1.5–2×.

The *vacation* benchmark simulates the transactional database of a travel agency, where clients make vacation reservations for flights, cars, and hotel rooms. For the duration of the benchmark, each thread repeatedly takes actions on the database, and each action is enclosed as a critical section. The *vacation* benchmark comes with both high and low contention configurations (*vacationhi* and *vacationlo*), where the difference between the two is the ratio of queries to writes.

The major action a thread takes is to make a set of reservations for a customer. This critical section (client.c: line 196) repeatedly queries for flights, cars, and rooms, recording the maximum price for each. After all queries, the thread reserves each trip component. This critical section dominates thread execution time. The other (rare) transactions on the database a thread can take is to delete a random customer (client.c: line 247) or update the available rooms, cars, and flights (client.c: line 267).

The database structure is implemented as four red–black trees, one each for customers, flights, rooms, and cars. Critical sections almost always execute a write. However, about 1% of the time, the vacation queries fail and no reservation is made.

As shown in Figure 3, despite the very rare read–only sections, the SWOPTLock policy outperforms both the MCS lock and pthreads lock significantly (1.5–2×) in parallel executions and allows the concurrent version to scale.

Figure 4 shows that the acceleration from SWOPT is not due to parallelism. The figure counts successful critical section executions for a SWOPTLock policy run. As shown, almost all critical section executions (99%) fail to execute successfully under SWOPT, indicating that SWOPT is not accelerating performance by parallelism.

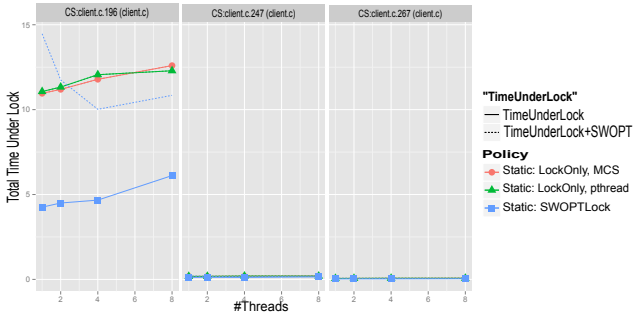

Figure 5: Time spent in critical section executions on *vacationlo*

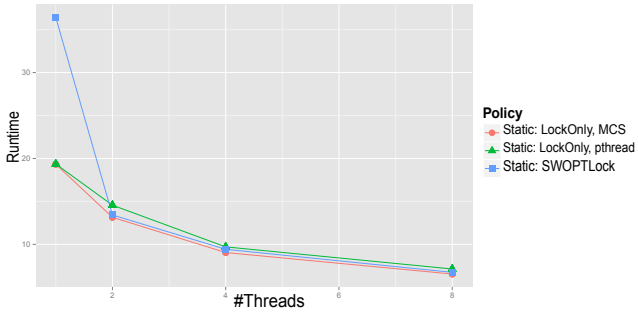

Figure 6: Execution times for various policies on *genome*

The mechanism by which prefetching improves performance is straight–forward. Speculative read–only executions of the database queries store relevant red–black tree paths in the waiting thread's cache. The execution aborts at the very end of the transaction when attempting to reserve some query result. Upon acquiring the lock, relevant red–black tree search paths and results are already stored in cache, reducing the thread's execution time under lock, and, consequently, increasing lock throughput.

The total time spent in critical section executions is shown in Figure 5. The "time under lock" records the total time spent executing critical sections under the lock (after the lock has been acquired). The "time under lock + SWOPT" records the total time spent executing critical sections under either lock or SWOPT. Based on this chart, comparing the LockOnly policy with SWOPTLock, we can see that threads spend approximately 60% less time under the lock when given an opportunity to prefetch on the major critical section of the benchmark, and that the savings cover the overhead of the SWOPT pass. We can also see that for a single thread, the failed speculation becomes an expensive overhead. With a single thread, lock throughput is no longer a concern, and the failed SWOPT pass enters the application's critical path. This observation can be seen in both Figures 3 and 5 for a single thread.

3.2 Prefetching hazards

It should be clear, however, that over aggressive SWOPT passes may be detrimental. It is possible to entirely negate the benefits of prefetching via SWOPT by prefetching too much data. By the time the fall back to lock occurs, the cache lines from the beginning of the critical section have been evicted, effectively *over–fetching* data. In order to exploit the prefetching effect, care should be taken to find the optimal prefetch distance.

An example of overfetching can be seen in STAMP's *genome* benchmark — overall runtime for the benchmark is shown in Figure 6. The *genome* program does DNA assembly of a string from its component substrings (segments). The program is executed in four phases. The first phase removes duplicate segments from the input (sequencer.c: line 292). The next phase, grouping, uses k dictionaries to group segments by their prefixes and suffixes of length k (sequencer.c: lines 374, 400, 413). The third phase, using this grouping information, can link substring samples by their overlapping keys. The linking phase is repeated for smaller and smaller k, ensuring that high fidelity matches are linked first (sequencer.c: line 482). Finally, the fourth phase sequentially reconstructs the original sequence by tracing linked segments. The phases are separated by thread barriers, ensuring that each phase is completed before the next one begins. In *genome*, when few samples are provided (a more realistic scenario than the default oversampling [24]), the grouping critical section dominates execution time.

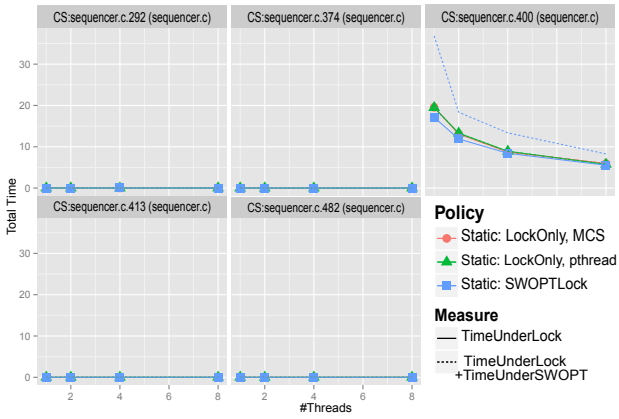

Figure 7: Time spent in critical section executions on *genome*

Overfetching in *genome* manifests in this grouping subphase. This subphase consists of long critical sections which, for each sample, iterate down sorted buckets to find the correct insertion point. These critical sections can be extremely long and do not execute a write until the correct insertion point is found (possibly near the end of the bucket).

For a single thread, the speculative execution does not result in improved performance under the lock, even though a long period is spent in the speculative pass. This lack of implicit acceleration occurs due to overfetching: the bucket traversal critical section is long enough that it evicts its early prefetches. When the lock fallback occurs, the incorrect lines have been prefetched and the "time under lock" remains the same as the LockOnly policy. The lack of a prefetching effect can be seen in Figure 7.

4. Lock Tuning

The addition of speculative executions can also affect lock performance by adjusting lock contention and acquisition order, even when prefetching effects are impossible.

4.1 Contention Control

Like the *vacation* benchmark, the *kmeans* benchmark comes in both high and low contention variants. The *kmeans* benchmark executes an implementation of the well known k-means clustering algorithm on sample data. The implementation is barriered – while each iteration of *kmeans* is done in parallel, all threads must complete before beginning the next iteration.

In each iteration of the *kmeans* benchmark, threads gradually pull points off a shared task queue in a critical section. For each point, after finding the nearest cluster center of the previous itera-

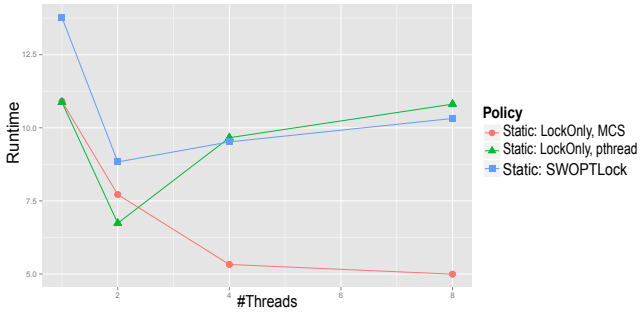

Figure 8: Runtime on *kmeanshi*

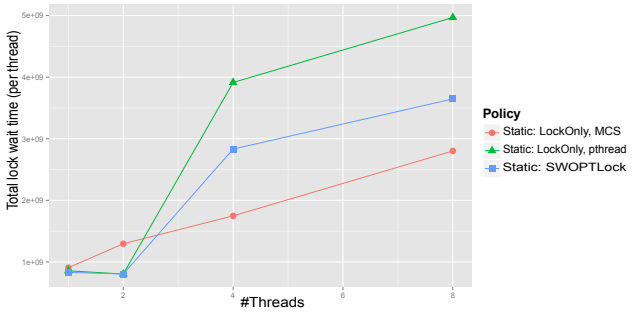

Figure 9: Time to acquire locks on *kmeanshi*

tion, the point is assigned membership to that cluster. This update is also done in a critical section. The final critical section of the iteration is done to update the total number of changed points across all threads in this iteration. If few enough points have changed, the clustering algorithm has converged.

After all points have been assigned, the master thread recalculates cluster centers based on membership information. After a barrier, all threads begin the next iteration with all points unassigned and in the task queue.

The *kmeans* benchmark is interesting in that every critical section almost immediately executes a shared write, preventing both SWOPT pass completion and negating any prefetching gains. However, SWOPT can even **still** improve performance (see Figure 8).

The inclusion of a short, doomed, SWOPT pass provides a small amount of backoff to the default pthreads spinlock, effectively adding contention control via SWOPT setup overhead. As a result, the time to acquire the lock is reduced and overall performance at eight threads improves. Figure 9 shows the time required to acquire the lock for the various policies, and demonstrates the reduction in contention for lock acquisition. When SWOPT is used, the lock is acquired approximately 33% faster.

Obviously, a presumably better performing solution is to use a different lock type more suitable to the situation. The use of a locally spinning FIFO MCS lock on the *kmeans* benchmark results in drastically better performance than either the default pthreads lock or the pthreads lock with doomed SWOPT attempts for contention control. The improvement can be seen in both lock acquisition latency (Figure 9) and overall performance (Figure 8).

4.2 Increasing Fairness

The inclusion of doomed SWOPT passes can also hurt performance if the lock is properly tuned, and especially if the lock is unfair. In general, for a saturated lock, fairness (FIFO) in servicing waiting

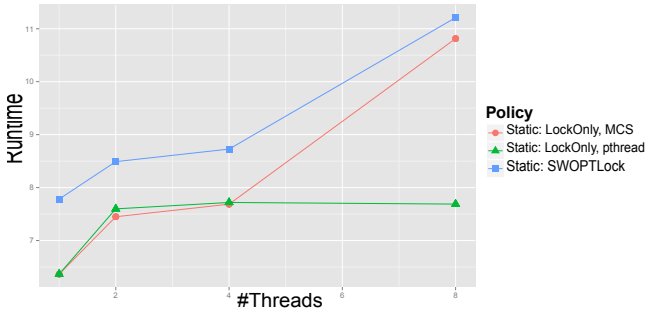

Figure 10: Execution runtime on *yada*

threads will result in decreased performance when compared to unfair (LIFO) orderings. In the LIFO case, the active thread is likely to have all necessary data already cached, whereas in FIFO cache lines will need to migrate from the previously active thread.

When total throughput is concerned, unfair locks are especially desirable when threads rapidly reacquire the lock upon releasing it (i.e. the amount of work outside of critical sections is minimal). This rapid reacquisition pattern occurs in the *yada* benchmark.

In the *yada* benchmark, threads execute a Delauney refinement using a variation of Ruppert's algorithm. After receiving an initial triangular mesh as input, the algorithm finds "bad" triangles which are too narrow and "bad" edges which are too close to neighboring points. These elements (edges and triangles), and their potentially bad surrounding neighbors, are refined by removing them and redoing the mesh for the region.

To do the refinement, each thread executes a tight loop, consisting of five critical sections. In the first critical section, the thread pops an element off the work queue. In the second critical section it checks that the element is not garbage (that is, it was removed by another thread doing retriangulation). In the third critical section, the thread grows the region around the bad element, to include neighboring elements and those within its diametrical circle, then retriangulates the affected region – this critical section (yada.c: line 228) dominates runtime. In the fourth critical section, the thread marks the removed element as garbage. Finally, in the fifth critical section, the thread adds any bad elements from its new region into the work queue.

Since each critical section is part of a single step in the algorithm, they are all executed the same number of times (barring conflicts over garbage elements). Furthermore, there is almost no work in between critical sections, meaning a thread is in a critical section for practically the entire execution.

As a result of lack of work external to critical sections, threads contend for the lock immediately after releasing it. Due to the implementation of the default pthreads lock, this lack of external work means that threads tend to reacquire the lock after releasing, exhibiting massive unfairness and consequently significantly improving caching effects. SWOPT usage equalizes access to the lock by injecting wasted work in between lock release and acquire, meaning that the lock becomes more fair and long run scenarios are avoided. Similarly, the MCS lock enforces fairness. Consequently, as seen in Figure 10, the unfair pthreads lock outperforms the fairer alternatives, especially at eight threads once hyperthreading is turned on.

Figure 11 shows the time it takes to acquire a lock for different lock policies. A "nonbusy" lock acquisition indicates that the lock was immediately acquired when requested (e.g. its first CAS succeeded), either because the lock was uncontended, or because a thread released and immediately reacquired it. The bar heights

Figure 11: Lock acquisition times and busy proportions on *yada*

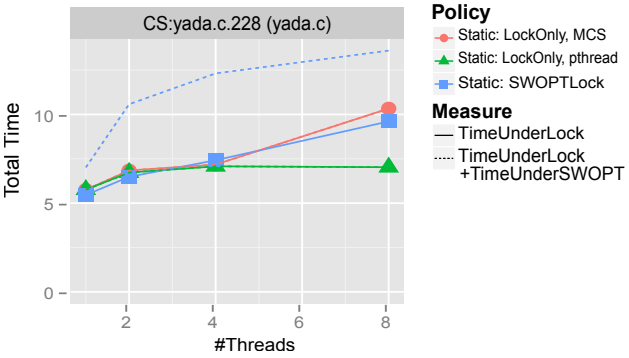

Figure 12: Time under lock on *yada*'s dominant critical section

show the time spent waiting for the lock, the percentages show the percentage of "busy" acquires. The chart shows that the pthreads lock is almost always acquired in a "nonbusy" manner, but the overall lock acquire time is entirely dominated by the few "busy" acquisitions — this signature is indicative of massive unfairness. In contrast, both SWOPT and the MCS lock have a higher proportion of "busy" acquisitions.

These fairer interleavings result in poorer cache performance once executing under the lock, especially at eight threads, when hyperthreading shrinks each thread's effective cache by sharing between hyperthreads. Figure 12 shows the time spent under the lock for the dominant critical section in *yada*. At eight threads, under the fair MCS lock and SWOPT policies, the time under lock grows drastically. With a fair servicing of waiting threads, the number of cache misses increase within the critical section.

5. Conclusion

With the results of this investigation we hope to provide some insights into the various emergent performance effects of lock elision. As should be clear, the correct speculative parameters for lock elision are often code and data dependent. The correct strategy may be obscured or influenced by non-obvious impacts of speculative actions, such as prefetching or contention control.

In future work, we hope to leverage these results to make better lock elision decisions and improve existing code, and allowing the adaptive policies of ALE to better predict performance. By taking into account the presented effects, we feel it is likely we can improve the performance of lock–elided software.

References

[1] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. "The convoy phenomenon". In: *SIGOPS Oper. Syst. Rev.* 13.2 (1979), pp. 20–25.

[2] William J. Bolosky and Michael L. Scott. "False sharing and its effect on shared memory performance". In: *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*. Sedms'93. San Diego, California, 1993, p. 3.

[3] B. Brett, P. Kumar, Minjang Kim, and Hyesoon Kim. "Chip: a profiler to measure the effect of cache contention on scalability". In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*. 2013, pp. 1565–1574.

[4] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. "High-performance throughput computing". In: *Micro, IEEE* 25.3 (2005), pp. 32–45.

[5] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. "Simultaneous speculative threading: a novel pipeline architecture implemented in sun's rock processor". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA, 2009, pp. 484–495.

[6] Jonathan Corbet. *Driver porting: Mutual exclusion with seqlocks*. `lwn.net/Articles/22818/`. Article. 2003.

[7] Dave Dice, Alex Kogan, and Yossi Lev. "Refined transactional lock elision". In: *10th ACM SIGPLAN Workshop on Transactional Computing*. TRANSACT '15. Portland, OR, USA, 2015.

[8] Dave Dice, Virendra J. Marathe, and Nir Shavit. "Brief announcement: persistent unfairness arising from cache residency imbalance". In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '14. Prague, Czech Republic, 2014, pp. 82–83.

[9] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. "Adaptive integration of hardware and software lock elision techniques". In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '14. New York, NY, USA, 2014, pp. 188–197.

[10] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. "Why STM can be more than a research toy". In: *Commun. ACM* 54.4 (2011), pp. 70–77.

[11] S. J. Eggers and T. E. Jeremiassen. "Eliminating false sharing". In: *Proceedings of the 1991 International Conference on Parallel Processing*. Vol. I. 1991, Architecture:377–381.

[12] Per Hammarlund et al. "Haswell: the fourth-generation intel core processor". In: *IEEE Micro* 34.2 (2014), pp. 6–20.

[13] Maurice Herlihy and J. Eliot B. Moss. "Transactional memory: architectural support for lock-free data structures". In: *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ISCA '93. San Diego, California, USA, 1993, pp. 289–300.

[14] C. Jacobi, T. Slegel, and D. Greiner. "Transactional memory architecture and implementation for IBM System Z". In: *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*. 2012, pp. 25–36.

[15] Andi Kleen. "Scaling existing lock-based applications with lock elision". In: *Commun. ACM* 57.3 (2014), pp. 52–56.

[16] M. Machado Pereira, M. Gaudet, J.N. Amaral, and G. Araujo. "Multi-dimensional evaluation of Haswell's transactional memory performance". In: *Computer Architecture*

and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on. 2014, pp. 144–151.

[17] John M. Mellor-Crummey and Michael L. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors". In: *ACM Trans. Comput. Syst.* 9.1 (1991), pp. 21–65.

[18] Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. "STAMP: Stanford transactional applications for multi-processing". In: *Workload Characterization 2008, IISWC*. 2008, pp. 35–46.

[19] Ravi Rajwar and James R. Goodman. "Speculative lock elision: Enabling highly concurrent multithreaded execution". In: *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 34. Washington, DC, USA, 2001, pp. 294–305.

[20] Nir Shavit and Dan Touitou. "Software transactional memory". In: *Proceedings of the 1995 ACM Symposium on Principles of Distributed Computing*. PODC '95. Ottowa, Ontario, Canada, 1995, pp. 204–213.

[21] J. Torrellas, M. S. Lam, and J. L. Hennessy. "Shared data placement optimizations to reduce multiprocessor cache miss rates". In: *Proceedings of the 1990 International Conference on Parallel Processing*. Vol. II. 1990, Software:266–270.

[22] Lingxiang Xiang and Michael L. Scott. "Software partitioning of hardware transactions". In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP 2015. San Francisco, CA, USA, 2015, pp. 76–86.

[23] R.M. Yoo, C.J. Hughes, K. Lai, and R. Rajwar. "Performance evaluation of Intel transactional synchronization extensions for high-performance computing". In: *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. 2013, pp. 1–11.

[24] Wenyu Zhang, Jiajia Chen, Yang Yang, Yifei Tang, Jing Shang, and Bairong Shen. "A practical comparison of de novo genome assembly software tools for next-generation sequencing technologies". In: *PLoS ONE* 6.3 (2011), e17915.