PACT 2017

Performance Improvement via Always-Abort HTM

Joseph Izraelevitz
Computer Science Department
University of Rochester
Rochester, NY, USA
Email: jhi1@cs.rochester.edu

Lingxiang Xiang
Parallel Computing Lab
Intel Corporation
Santa Clara, CA, USA
Email: lingxiang.xiang@intel.com

Michael L. Scott
Computer Science Department
University of Rochester
Rochester, NY, USA
Email: scott@cs.rochester.edu

***Abstract*—Several research groups have noted that hardware transactional memory (HTM), even in the case of aborts, can have the side effect of warming up the branch predictor and caches, thereby accelerating subsequent execution. We propose to employ this side effect deliberately, in cases where execution must wait for action in another thread. In doing so, we allow "warm-up" transactions to observe inconsistent state. We must therefore ensure that they never accidentally commit. To that end, we propose that the hardware allow the program to specify, at the start of a transaction, that it should in all cases abort, even if it (accidentally) executes a commit instruction. We discuss several scenarios in which always-abort HTM (AAHTM) can be useful, and present lock and barrier implementations that employ it. We demonstrate the value of these implementations on several real-world applications, obtaining performance improvements of up to 2.5× with almost no programmer effort.**

***Keywords*-hardware transactional memory; thread-level speculation; locks; barriers**

I. Introduction

Programs for multicore machines traditionally synchronize access to shared memory using blocking primitives like locks. Threads that always acquire a lock before accessing shared data prevent each other from performing conflicting accesses: lock-protected critical sections provide mutual exclusion and thereby atomicity. Unfortunately, it is typically extremely difficult to specify the minimal locking required for correctness, and overly conservative locking can waste significant processing resources on semantically unnecessary busy-waiting. Other blocking primitives, such as barriers, exhibit similar inefficiencies.

Several recent machines provide a mechanism—hardware transactional memory (HTM)—that reduces, but does not eliminate, the blocking due to synchronization. HTM guarantees that all code executed within a *transaction* will appear to be atomic to all other threads without requiring them to wait. During transaction execution, the hardware buffers writes in space invisible to other threads, and leverages the cache coherence protocol to detect conflicts with concurrent execution in those threads. If the transaction cannot complete and commit its changes (due to a conflict with another thread, overflow of speculative hardware state, or the use of an instruction that cannot easily be isolated) it aborts, reverting all its changes. In the wake of an abort, the thread may either retry the transaction or revert to a different synchronization technique—typically a *fall-back lock*—to guarantee progress [1]. In contrast with locks, which conservatively prevent data races, transactions optimistically assume that races will not occur, and recover when this assumption is wrong. As a result, HTM may result in higher concurrency than would otherwise be achievable. It cannot, however, replace locking in all scenarios: some transactions will never or rarely complete under HTM. Transactions with large working sets, with I/O operations, or with frequent conflicts with other transactions will typically perform better if rewritten to use locks. Even on machines with HTM, some programs must wait for synchronization.

We argue that the wasted cycles spent waiting for a lock (or other blocking construct) should be turned into useful work whenever possible. Our inspiration is an observation made by a number of previous authors: HTM can improve the performance of a program, even when it never succeeds. This effect occurs because there is sometimes a significant "prefetching" effect in which a failed transaction, despite leaving behind no changes to semantic state, serves to warm up various hardware structures—in particular, the branch predictor and caches—for future executions of the transaction [2], [3], [4], [5]. If a failed transaction executes sufficiently far before aborting, its subsequent attempts (protected by either HTM or a fall-back lock) will execute significantly faster due to this accidental prefetching. Under certain conditions, the speed-up can be quite significant, even if the transaction never completes under HTM. In this sense, HTM acts as accidental thread-level speculation.

A trivial (and incorrect) use of wasted waiting cycles for prefetching would be to start an HTM transaction while waiting for a lock, then speculatively execute the protected critical section. This speculative execution of the critical section would serve to warm up the cache and branch predictor of the waiting thread, even if the transaction is unsafe to commit (the speculator is reading inconsistent state, so its

This work was supported in part by NSF grants CCF-0963759, CCF-1116055, CNS-1116109, CNS-1319417, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Study and a Google Faculty Research award.

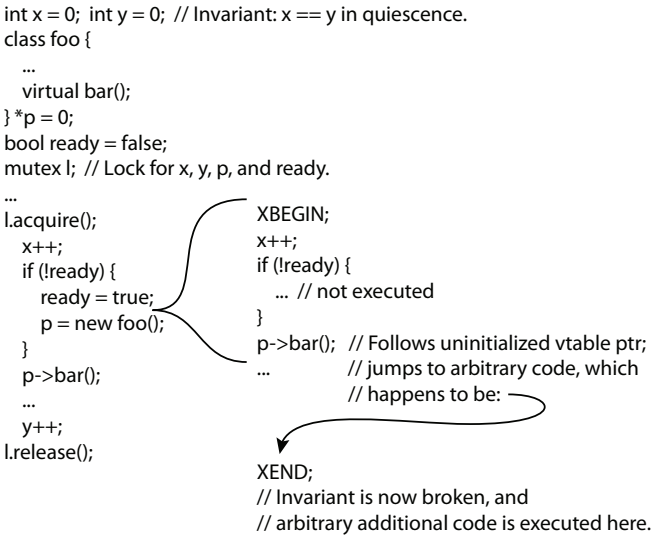


Figure 1: Program state corruption via lazy subscription. If an HTM transaction (right) executes between the indicated lines of a lock-based critical section (left), the transaction can read inconsistent state and accidentally commit, violating program invariants.

computation may not be correct). Unfortunately we *cannot guarantee* that the speculator will not accidentally commit its changes. Blind jumps to corrupted addresses in function pointers or the virtual function table could cause a thread to jump to a commit instruction and accidentally commit its speculative HTM transaction, thereby violating program invariants (this problem is a variant of the *lazy subscription problem* [6], [7]; an example is shown in Figure 1).

Our work attempts to use wasted waiting cycles for prefetching while avoiding the lazy subscription problem. Our proposed solution is a new hardware primitive which we call *always-abort hardware transactional memory*. Always-abort HTM (AAHTM) acts just like traditional HTM, with one exception: its transactions are guaranteed by the hardware to always abort and never commit. In general, we envision the use of always-abort HTM as an alternative to busy-waiting in synchronization primitives: instead of waiting, we can do something useful to prepare for future execution.

The idea of program-controlled prefetching as an alternative to busy-waiting is widely applicable. We focus in this paper on synchronization primitives such as locks and barriers, but always-abort HTM is likely to be useful wherever waiting is required, such as in synchronous communication or requests to hardware accelerators. AAHTM execution is always safe, because it can never affect semantic state. It is also significantly more flexible than traditional hardware prefetching: its speculative path can be explicitly controlled and tuned by the programmer to achieve higher accuracy and utility than are possible purely by watching the (pre-wait) instruction stream. For processors that already support HTM, the AAHTM extension should be straightforward to implement—all that is required is a slightly different execution mode. Significantly improved hardware prefetching is likely to require a much larger hardware investment.

The rest of this paper explores the uses and utility of always-abort HTM. We begin with a motivating micro-benchmark, which demonstrates the potential gains and pitfalls of our technique. Using the lessons learned there, we propose several synchronization primitives that incorporate AAHTM. We continue with performance results on micro- and real-world benchmarks. Our results confirm that AAHTM based primitives can significantly outperform both lock and HTM solutions. Finally, we turn to a review of related work, and conclude with additional ideas regarding other potential uses of our hardware primitive. Source code for this project is available at http://github.com/greenshell/aahtm.

II. Motivation and Implementation

Our argument for always-abort HTM is twofold: under amenable conditions, it provides significant performance benefits over busy-waiting or traditional HTM alternatives, and, furthermore, hardware implementation of the technique is likely to be trivial on a machine that already supports HTM.

A. Performance

To quantify the potential benefits of AAHTM, we begin by exploring a simple micro-benchmark—**ArrayBench** (Figure 2)—which investigates what we see as a standard use case for our hardware primitive: using AAHTM as an alternative to busy waiting in lock acquisition. Note that this strategy is quite different from lock elision: instead of replacing locks with HTM, we retain true mutual exclusion but accelerate critical sections with AAHTM.

In ArrayBench, threads repeatedly write to pseudorandom and deterministic locations within a shared array `A`. The array is protected by a global lock. We vary critical section size by changing the number of locations touched under the lock, either ten in the "small footprint " case or one hundred in the "large footprint" case. In the "high contention" case, we start the critical section by writing to `A[0]`; in the "low contention" case, we skip that write. Our results report write throughput.

We benchmark several alternatives. The first is regular HTM supported by a fall-back test-and-set lock. On transaction abort, we either fall back to the lock immediately (**htm-1**) or after nine more tries (**htm-10**). The second is a simple test-and-test-and-set lock (**tatas**). The third alternative, **tatas-aahtm**, is an enhanced test-and-test-and-set lock in which, if the first test fails, we start an AAHTM execution of the critical section (pseudocode in Fig. 3). For our experiments, since AAHTM does not exist, we instead

use regular HTM in an unsafe manner, admitting the lazy subscription problem. We do not believe that the error case (accidental jump to a commit instruction) ever arises in our experiments.

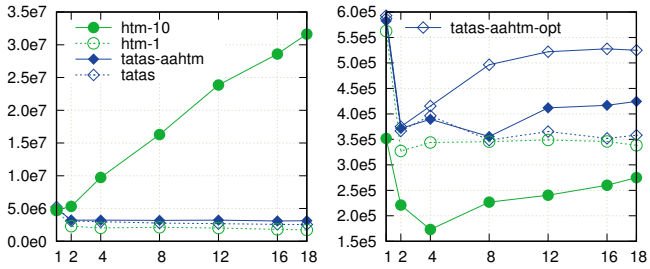
As expected, when critical section size is small and contention is low, HTM is by far the best choice, since most of the transactions complete without conflicts and without being terminated by time or space constraints. In contrast, when contention is high and transaction size is large, most transactions fail due to conflict and eventually revert to the fall-back lock, meaning that the simple lock (**tatas**) outperforms HTM. Furthermore, by using the busy-wait time to prefetch for the critical section, we can reduce the time threads hold the lock and increase throughput via always-abort HTM (**tatas-aahtm**).

Using ArrayBench we can also explore a optimized form of AAHTM prefetching—one in which the prefetching thread switches to an alternative code path, created by the programmer or the compiler. Since AAHTM is a sandbox that never writes anything to the memory, the alternative, speculative code path can aggressively circumvent abort-prone code (such as I/O operations or accesses to highly contended variables), without compromising correctness. In the ArrayBench experiment, we created a speculative path (**tatas-aahtm-opt**) that avoids the contentious access to the first element of the array when in AAHTM, reducing the likelihood of a conflict abort when prefetching.

This motivating example illustrates the best case for our new hardware primitive: high abort-rate critical sections with large memory footprints. In this case, AAHTM provides the best performance, surpassing both regular HTM and traditional locks. Regular HTM in particular is likely to fail (here due to contention, but conceivably also due to cache overflow or I/O), and the prefetching of AAHTM provides a real benefit by warming up the cache. Note that under non-optimal circumstances, AAHTM doesn't significantly hurt performance relative to a simple lock, but might deliver significantly worse performance than regular HTM since it requires all critical section executions to be serialized.

B. Implementation

We envision an implementation of always-abort HTM with two new instructions, AAHTM_BEGIN, which begins a new always-abort transaction, and AAHTM_TEST, which tests to see whether the thread is currently in an always-abort transaction (our instructions are analogous to Intel's TSX instructions XBEGIN and XTEST). We expect that other HTM instructions, namely XTEST and as XABORT, should work as normal within an always-abort transaction, but an XEND commit instruction is unsupported within an AAHTM transaction and results in an abort. Always-abort transactions will also abort wherever a normal HTM transaction would fail, such as at an unsupported instruction, or on interrupt or cache overflow.



(a) low contention, small footprint. (b) high contention, large footprint.

Figure 2: ArrayBench throughput for TATAS lock and HTM. The X axis is the number of concurrent threads; the Y axis is transactions per second.

The primary hardware cost for always-abort HTM is an extra architectural state bit per hardware thread indicating that the transaction underway must abort. This bit is set by the AAHTM_BEGIN instruction and is queried by AAHTM_TEST. The only additional cost is a small amount of logic to verify, before committing a transaction, that the always-abort flag is not set.

III. Designs

We have developed implementations of synchronization primitives that use always-abort HTM as an alternative to busy waiting. Exactly how to incorporate the new HTM technique is not always obvious. In particular, our experiments revealed two important design considerations. The first consideration is that the value of prefetching declines with time: if a thread that has performed an AAHTM prefetch does not get to execute its "real" code soon, the prefetch may be wasted, as prefetched cache lines are stolen and overwritten by the lock holder or simply displaced by other lines. The second consideration is that it is best to limit the number of threads concurrently prefetching, especially when waiting for a lock. Since their write sets may overlap, concurrent AAHTM transactions may result in early aborts, negating the prefetching advantage. In general, it seems better to limit the number of prefetchers, particularly if (in keeping with the first consideration) threads that *do* prefetch have priority access to the critical section when the lock becomes available.

A. Busy-wait Alternative

The above considerations notwithstanding, always-abort HTM can be used as a simple drop-in replacement for busy-waiting. Some simple designs are shown in Figure 3, including an extension to pthread_mutex and a test-and-set lock. Basically, if a thread would normally busy-wait for lock acquisition, it enters an always-abort transaction instead and returns to user code, pretending that it holds the lock. When the transaction eventually fails, the thread's state is rolled back to the lock acquisition method. The thread then

```
// we emulate AAHTM using regular Intel x86 HTM.
#define AAHTM_BEGIN XBEGIN
#define AAHTM_TEST XTEST

// how we enter AAHTM
int enter_aahtm(){
  if(AAHTM_BEGIN() == HTM_SUCCESSFUL)
    return 1; // we are in AAHTM
  return 0; // we just finished executing AAHTM
}

// used for both simple and prioritized tatas lock
struct tatas_lock_t{
  union{
    struct{int32_t held; int32_t n_rdy;};
    int64_t all;
  };
};

// tatas lock with AAHTM as busy wait−alternative
void tatas_lock_aahtm(tatas_lock_t *lk){
  if(AAHTM_TEST()){return;}
  int tries = 0;
  while(lk→held || tas(&lk→held)){
    // if lock is held, start speculating
    if(enter_aahtm()){return;}
    else{tries++;}
    // revert to the lock if out of tries
    if(tries≥NUM_TRIES){
      while(lk→held || tas(&lk→held))
        pause(INTERVAL);
      break;
    }
  }
}
void tatas_unlock_aahtm(tatas_lock_t *lk){
  if(!AAHTM_TEST()){lk→held = 0;}
}

// pthreads lock with AAHTM as busy wait−alternative
void pthread_lock_aahtm(pthread_mutex_t *lk){
  if(AAHTM_TEST()){return;}
  int tries = 0;
  while(pthread_mutex_trylock(lk)≠0){
    if(enter_aahtm()){return;}
    else{tries++;}
    if(tries≥NUM_TRIES){
      pthread_mutex_lock(lk);
      break;
    }
  }
}
void pthread_unlock_aahtm(pthread_mutex_t *lk){
  if(!AAHTM_TEST()){pthread_mutex_unlock(lk);}
}
```

Figure 3: Busy-wait alternative

checks to see whether it can exit its busy wait by attempting to grab the lock. If it needs to wait again, it can either retry the always-abort transaction (to try to prefetch further in its speculation) or fall into the normal busy-wait.

B. Test-and-Test-and-Set Priority Lock

The test-and-test-and-set lock of Figure 3 incorporates AAHTM, but exerts little control over the waiting threads. In particular, arbitrary numbers of threads might speculatively prefetch, increasing early aborts and diminishing the utility of the technique. Furthermore, once a thread has completed its prefetch, it might not acquire the lock for a while, conceivably negating the prefetch's benefits. To solve these problems, we designed a slightly more complicated test-and-test-and-set lock (Figure 4) which strictly prioritizes threads that have completed their prefetch and roughly controls the number of speculating threads. To control speculation, this lock uses a counter (n_rdy) to track the number of threads that have completed an AAHTM execution (i.e. "warm threads"). If n_rdy is non-zero, no thread that has not yet speculated in AAHTM can acquire the lock, thereby guaranteeing that the lock will be acquired by a warm thread, or that the lock is uncontended. As in the previous locks, if a thread finds that the lock cannot be acquired (either because the lock is held or because a warm thread is waiting), it enters an AAHTM transaction and returns to user code, pretending that it has the lock.

```
// tatas priority lock with AAHTM
void tatas_pri_lock_aahtm(tatas_lock_t *lk){
  if(AAHTM_TEST()){return;}
  int tries = 0;
  tatas_lock_t cp;
  while(true){
    cp.all = lk→all;
    if(cp.n_rdy == 0 && !lk→held && !tas(&lk→held))
      break;
    if(cp.n_rdy < MAX_SPECS){
      if(enter_aahtm()){return;}
      else{
        fai(&lk→n_rdy,1);
        while(lk→held || tas(&lk→held)){}
        fai(&lk→n_rdy,−1);
        break;
      }
    }
    pause(INTERVAL);
  }
}
void tatas_pri_unlock_aahtm(tatas_lock_t *lk){
  tatas_unlock_aahtm(lk);
}
```

Figure 4: Test-and-test-and-set lock with priority

C. Ticket Lock

The test-and-test-and-set priority lock presented above provides a number of advantages over simple busy-wait elision. However, test-and-set locks in general are unfair, and the presented priority design is no exception. It is possible for a thread to fail to get the lock indefinitely by repeatedly losing the tatas attempt. To provide fairness to waiting threads, other locks may be used—for instance, the ticket lock [8].

In a traditional ticket lock, threads increment two counters: next_ticket and now_serving. A thread that wishes to acquire the lock atomically increments the next_ticket

```
struct ticket_lock_t{
  int next_ticket;
  int now_serving;
};
void ticket_lock_aahtm(ticket_lock_t *lk){
  if(AAHTM_TEST()){return;}
  int tries = 0;
  int my_ticket = fai(&lk→next_ticket, 1);
  int dist = 0;
  while( (dist=(my_ticket−lk→now_serving) ) > 0){
    if(dist≤MAX_DIST && dist≥MIN_DIST
     && tries<NUM_TRIES){
      if(enter_aahtm()){return;}
      else{tries++;}
    }
    else{pause(INTERVAL);}
  }
}
void ticket_unlock_aahtm(ticket_lock_t *lk){
  if(!AAHTM_TEST()){lk→now_serving++;}
}
```

Figure 5: Ticket lock

counter. It then spins on the now_serving counter. When the counter matches the ticket number obtained from next_ticket, the thread has acquired the lock. Upon exiting its critical section, the thread increments the now_serving counter, passing the lock to the next thread in line. Figure 5 shows the implementation of a ticket lock with always-abort HTM. An advantage to the ticket lock is that there is a fixed order in which threads will pass through the lock. Consequently, we can delay speculation until we are close to acquiring the lock by monitoring the now_serving counter, only beginning AAHTM speculation when we are approaching the head of the line. With a similar mechanism, we can also control how many threads are concurrently speculating.

D. Barrier

In addition to locks, always-abort HTM is useful for barriers. Once early threads have reached the barrier, they can speculate into the next phase of execution while waiting for the straggling threads. There are two major design points to note. First, in contrast to locks, the speculation is generally restricted to local data; no data races typically exist within a barrier phase. Consequently, the likelihood of a conflict between concurrently speculating threads is very low; the only conflicts that are likely to occur are between the speculating threads and the straggling threads that have yet to reach the barrier. Second, the design of our barrier, based heavily on the GNU libGOMP implementation, also allows us to avoid false sharing between the arriving thread counter (awaited) and the release signal (gen). Consequently, upon entering the always-abort transaction, we can subscribe to the release signal and abort our transaction immediately once all threads arrive.

```
struct barrier_t {
  int total; // number of threads to wait for
  int gen; // generation counter with flags in low bits
  char[] padding; // padding to avoid false sharing
  int awaited; // number of threads at the barrier
};
void barrier_wait(barrier_t *bar){
  int state = bar→gen;
  if(fai(&bar→awaited,−1) == 0){
    bar→awaited = bar→total;
    state+=1;
    bar→gen = state;
    return;
  }
  int gen = state;
  int tries = 0;
  do{
    if(bar→gen == gen){
      tries++;
      if(enter_aahtm()){
        if(bar→gen≠gen){XABORT();}
        else{return;}
      }else{
        if(tries<NUM_TRIES){continue;}
        else{pause(INTERVAL);}
      }
    }
    gen = bar→gen;
  }while(gen≠state+1);
}
```

Figure 6: Barrier (heavily based on GNU libGOMP)

IV. Evaluation

Our experiments were conducted on a dual-socket, 18-core Intel Xeon E5-2697 v4 (Broadwell) machine running Linux kernel version 3.10.0. The hardware cache prefetcher was turned on. Code was compiled with gcc 5.3.0 using the −O3 optimization flag. Unless otherwise noted, all experiments were performed without hyperthreading, and with each thread pinned to a separate core of a single 18-core processor. Reported results show the average of three runs at each configuration point, and no major performance variation was seen across runs.

A. Locks

Our locks are implemented as a dynamically loaded library that overwrites the pthread synchronization functions. The library is linked in via LD_PRELOAD at run time. Within our library, we implemented several mutex alternatives:

tatas: An exponential back-off test-and-test-and-set lock.

ticket: A FIFO ticket lock with linear back-off proportional to the distance to the lock owner.

htm-1, htm-10: Simple uses of HTM, similar to lock elision, that try the transaction either one or ten times before falling back to a global test-and-test-and-set lock. **htm-1** mimics Intel's Hardware Lock Elision. Following the early subscription convention, the global lock is checked at the beginning of each hardware transaction. Here we take some

precautions to avoid the "lemming effect," [9], [10] where the held lock prevents HTM transactions from completing, which in turn fall back to the lock and prevent more HTM transactions from succeeding. In the case that the lock is already held at the beginning of the transaction, the thread jumps into an infinite loop, waiting for the lock holder to abort it. In htm-10, after an HTM failure, we do exponential back-off before retry.

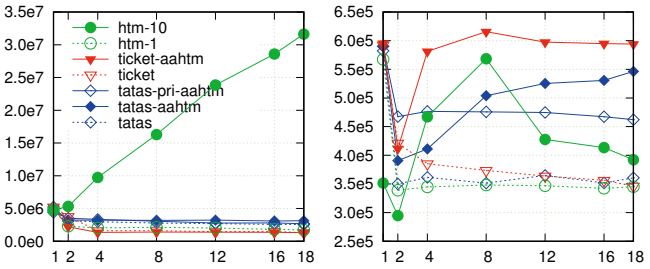
tatas-aahtm: The trivial AAHTM test-and-test-and-set lock of Section III-A. We set NUM_TRIES to 4 based on experimentation for reasonable parameters for generic workloads.

tatas-pri-aahtm: The prioritizing AAHTM test-and-test-and-set lock of Section III-B. We set NUM_TRIES to 4 and MAX_SPECS to 1.

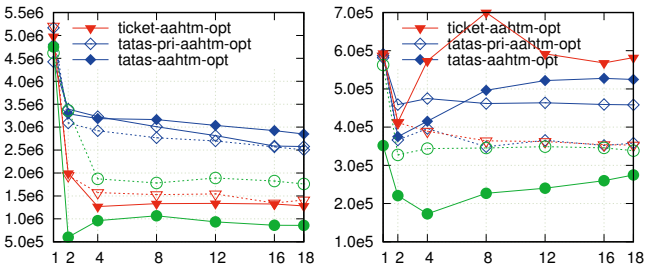
ticket-aahtm: The AAHTM ticket lock of Section III-C. We set NUM_TRIES and MAX_DIST to 4 and MIN_DIST to 2.

1) Micro-benchmarks: We ran our locks on several micro-benchmarks and real-world applications. **ArrayBench** (Figure 7) is the micro-benchmark we introduced in Section II-A. In it, threads contend to access an integer array with one million elements. Each thread generates addresses to touch within the array before contending to enter the critical section. Tests run for approximately five seconds, and we report operations (critical sections) per second (y-axis) as a function of thread count (x-axis). The test has two parameters—size and contention level. The size parameter refers to the number of writes in each critical section; the low size touches ten, the high size touches one hundred. Under low contention, threads write to all their addresses and then leave. Under high contention, all threads first touch the zeroth array element at the beginning of the critical section before touching the rest of their addresses. For this test we also explored an optimized variant of each lock (marked **opt** in Figure 7) which, when executing under AAHTM, employs an alternative code path that elides the high contention write to maximize prefetching gains.

The results of this benchmark are promising for always-abort HTM. An AAHTM primitive is the best option in three of the four configurations, and is only beaten by htm-10 on the low contention, small footprint configuration (Figure 7a). For configurations on which it wins, AAHTM outperforms the nearest alternative by 10 to 200%. The AAHTM ticket lock appears to be the best option for large footprint tests, likely due both to its well-known tendency to mitigate lock contention and to its orderly control of prefetchers. The benefit of the prefetching is clearly seen in the large footprint configurations: the AAHTM ticket lock outperforms its non-prefetching variant by 2 to 3× in these tests. For smaller footprint tests, prefetching becomes less important, and the test is instead limited by the fairness of the lock. The unfairness of the test-and-test-and-set implementations gives them an advantage over the fair ticket locks by allowing



(a) low contention, small footprint. (b) low contention, large footprint.



(c) high contention, small footprint. (d) high contention, large footprint.

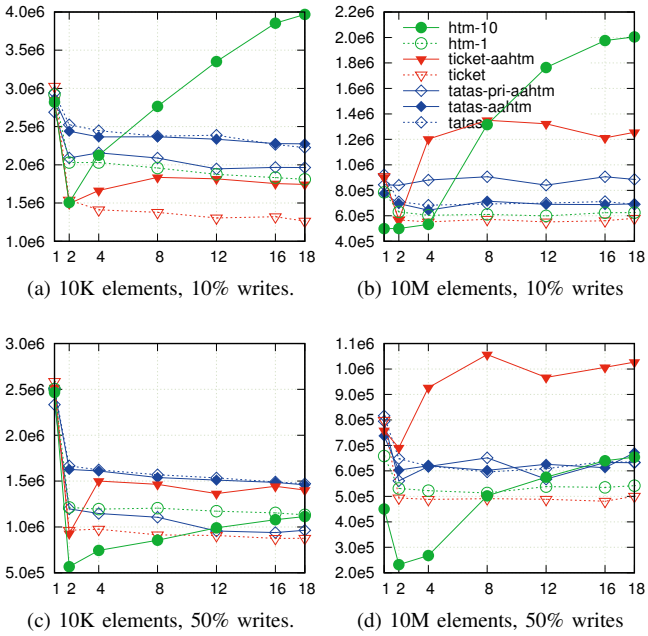
Figure 7: Throughput (critical sections per second) of ArrayBench on Broadwell Xeon.

repeat acquisitions that reduce cache line movement. Finally, note that HTM is the best option only when contention is low and the footprint is small; even so, the retry parameter is extremely important: htm-1 is outperformed by the locks due to spurious aborts.

In **MapBench** (Figure 8) we once again contend on a global data structure—this time, a std::map<int,int> (red-black tree), protected by a global lock. Within its critical section, each thread does a single operation on a randomly chosen key within the key space. The test has two configuration parameters. The first is the size of the key space: either 10K or 10M keys. During initialization, the map is filled with 50% of all possible keys. The second configuration parameter is the ratio of find/insert/delete operations; we test both 80%/10%/10% and 0%/50%/50% configurations. Since the map is half full at initialization, these ratios result in 10% and 50% writes respectively. Each test runs for approximately five seconds, and again we report operations per second (y-axis) as a function of thread count (x-axis).

Once again in this benchmark, we see the benefit of AAHTM, but under more specific conditions. Since the critical sections are so small, and mostly read-dominated, traditional HTM works well in most cases. However, when the write percentage goes up and the tree size grows, HTM becomes less likely to succeed. In such cases, it is useful to prefetch tree state while waiting for the lock, and the AAHTM ticket lock again dominates (Figure 8d).

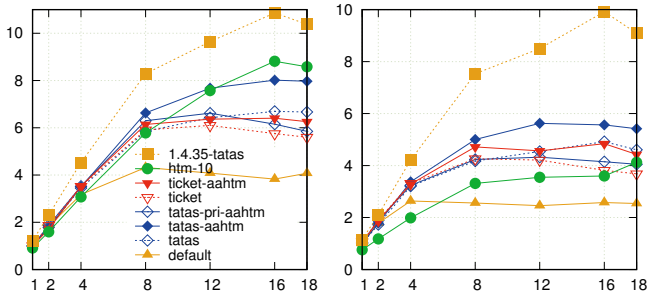
2) Memcached: To investigate the usefulness of our technique on real-world code, we explored the use of our



(a) 10K elements, 10% writes. (b) 10M elements, 10% writes

(c) 10K elements, 50% writes. (d) 10M elements, 50% writes

Figure 8: Throughput (operations per second) of MapBench on Broadwell Xeon.



(a) 90% get, 10% set. (b) 50% get, 50% set

Figure 9: Speedup over single thread on memcached.

locks on **memcached** [11], a key-value store used to cache query results. Memcached is widely used; its clients include Facebook, Wikipedia, and Flickr. The software has been in active development since 2003, and early versions used coarse grain synchronization. Threads contended and bottlenecked on two global locks [12], [13], [14]. The cache_lock protects access to the global hash table, and the stats_lock protects access to usage statistics.

Obviously, AAHTM cannot improve performance on highly performant code which avoids busy-waiting through fine-grained locking, as current versions of memcached do, so, following others, we investigated an earlier version (1.4.5) as a lock benchmark [12], [13].

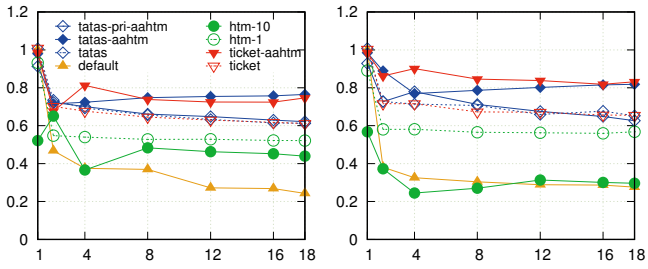
Our setup ran both a memcached server and client on our Intel Broadwell server, each isolated on its own socket. Mirroring our other tests, each server thread was pinned to its own core on the socket. Our tests followed the methodology of Dice et al. [12]. We used the tool memaslap [15] as the client to generate a stream of memcached requests according to a desired distribution. We used 32 client threads, which generated requests with uniformly distributed 8-byte keys and 128-byte values. Tests ran for one minute, and we recorded the average throughput across the interval. As above, our locks were dynamically loaded into the executable, overwriting the traditional pthread mutex. Our results, shown in Figure 9, explore different read to write ratios of client requests, reporting speedup over single-threaded default performance (y-axis) as a function of thread count (x-axis).

Our real-world memcached benchmark results reiterate the lessons learned in the micro-benchmarks. In conditions where HTM is likely to succeed—here the 90% read-dominated workload of Figure 9a—it outperforms all lock-based solutions at high thread counts, where the extracted parallelism is greater than the cost of the HTM transaction. Compared to the uninstrumented memcached (**default** in the figure), HTM is about twice as fast. In conditions where HTM is unlikely to succeed—here the 50% read-write ratio of Figure 9b—locks are more performant. And as memcached stores keys in a large table which cannot fit in last-level cache, each critical section contains multiple random accesses and prefetching via AAHTM is useful. The best lock solution is the AAHTM TATAS lock, which is about 2.5× faster than the default implementation. Noticeably, in the write-dominant case, the AAHTM-accelerated ticket lock gives on-par performance with the test-and-test-and-set lock, while also providing fairness.

For completeness, we also included performance tests of a recent version of memcached (v.1.4.36), which has accelerated the performance from version 1.4.5 by about 3.7× through a rewrite of the synchronization framework using fine-grained locking across seven years of development and over thirty versions. By simply using AAHTM locks on the old version, we are able to achieve 50–60% of this speedup with absolutely no programming effort.

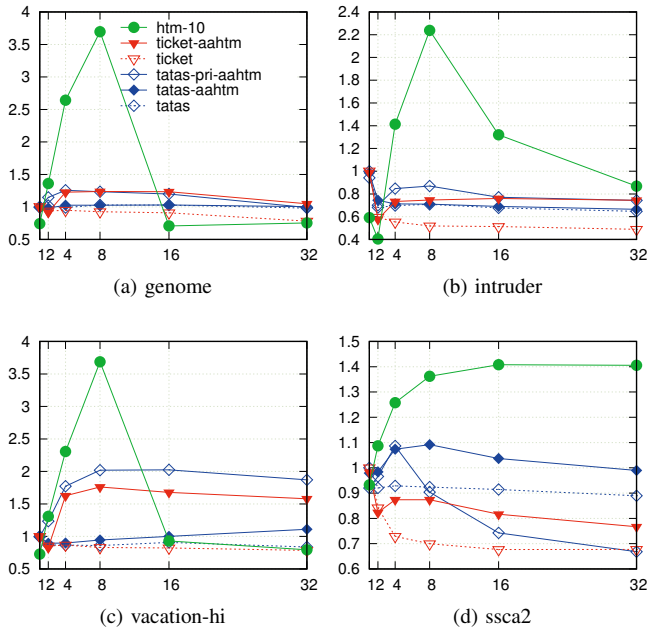
3) Kyoto Cabinet: We further investigated the performance of our primitives on **Kyoto Cabinet** [16], a fast database management library written in C++. The library employs global, coarse-grain synchronization using reader-writer locks. It implements several data structures for use as tables; we used the included in-memory B+ tree database (GrassDB). Kyoto Cabinet also includes several benchmarks with its distribution; we ran the kcgrasstest benchmark with a total of 1M (Figure 10a) and 5M (Figure 10b) random operations respectively.

Since the benchmark we chose is write dominant, the default pthread reader-writer lock cannot take advantage of parallel readers. This **default** is significantly slower than



(a) small database (~1MB) (b) large database (~100MB)

Figure 10: Speedup over single thread on Kyoto Cabinet.



(a) genome (b) intruder

(c) vacation-hi (d) ssca2

Figure 11: Speedup over single thread on STAMP benchmarks.

tatas in all cases. Similarly, both **htm-1** and **htm-10** fail to deliver speedup due to excessive data conflicts. For instance, with 20 threads, each HTM commit in **htm-10** requires 4.3 tries on average, and 91% of all aborts are caused by conflicts. This result suggests that HTM is far from a silver bullet — trivial use of HTM may severely hurt the performance. Moreover, given the lack of good HTM debugging tools, we expect it would take non-trivial effect to transactionalize this library.

In contrast, AAHTM enhances the performance of locking-based code without extra programming effort. With high thread counts ($\geq$4), we see 15%–27% performance gain from **ticket-aahtm** and 4%–25% gain from **tatas-aahtm**, compared to their baseline versions. In general, the larger database (Figure 10b) benefits more from speculation because AAHTM can prefetch more data for a large data set. As in **memcached**, FIFO **ticket-aahtm** outperforms the unfair **tatas**.

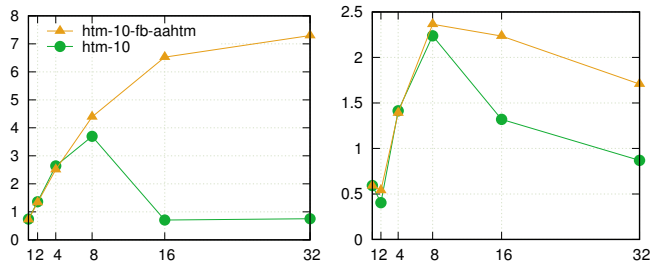
4) STAMP: The Stanford Transactional Applications for Multi-processing (**STAMP**) benchmark suite is a commonly used benchmark in the transactional memory community [17]. The suite contains eight benchmarks built using a transactional style of synchronization, each adapted from some real-world application. Although these benchmarks are optimized for transactional memory, we plugged our AAHTM library into their single-global-lock version to study the potential benefit of AAHTM. In order to experiment with larger thread counts, we used a single socket and assigned each thread to its own core, but also used hyperthreading to obtain levels of concurrency beyond 18 (STAMP thread counts must be a power of two, giving us a max of 32, instead of 36).

In Figure 11, we highlight the STAMP benchmarks in which AAHTM lock performance varied significantly ($>$10%) from the corresponding traditional lock implementation, in each case indicating a nontrivial benefit of AAHTM. The results are shown as speedup over a single thread execution (y-axis) as a function of thread count (x-axis).

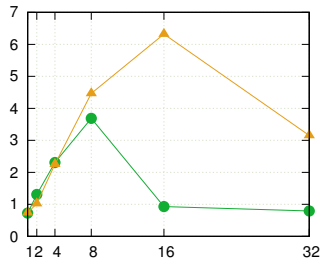
In all benchmarks, traditional HTM scales well at low thread counts due to a relatively low abort rate. However, HTM performance tends to drop off significantly at 16 threads in several benchmarks, suggesting that the lemming effect has started to emerge as a result of high contention. In contrast, global locks are quickly saturated reaching their maximum throughput by four threads. The use of AAHTM, however, generally gives the best global lock performance by reducing the time spent in critical sections through prefetching. This makes sense: the benchmarks **vacationhi**, **genome**, and **intruder** all involve a relatively large number of map operations; however, of these, all but **vacationhi** are read dominated.

5) Avoiding performance collapse of HTM runtime: As observed in earlier studies [9], [10], and shown in our STAMP results (Figure 11), an HTM runtime may suffer from the lemming effect, in which the fall-back lock prevents other HTM transactions from completing and they in turn fall back to the lock, generating a self-perpetuating abort scenario. As a result, a sudden performance collapse may occur at some high thread count even if the application scales well at low thread counts.

Sophisticated contention management in HTM runtimes [10], [18] is a possible solution to the above issue, as it reduces HTM aborts. When serialized execution of the fall-back path is unavoidable, however, due either to repetitive overflow/conflict aborts or to unsupported instructions, AAHTM can improve the fall-back lock of regular HTM and mitigate the lemming effect somewhat. This hybrid approach allows us to build a runtime that obtains the best

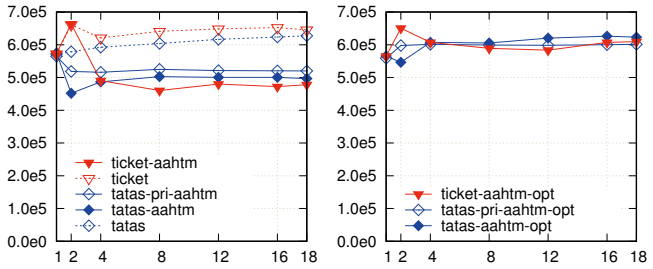


(a) genome (b) intruder



(c) vacation-hi

Figure 12: Speedup over single thread on STAMP benchmarks with AAHTM fall-back lock.



(a) 10 cache lines, 100 writes (b) avoiding cache line stealing

Figure 13: Performance hazards and avoidance on ArrayBench for small array.

of both worlds: improved parallelism when HTM succeeds, and optimized serial execution of the fall-back path when it cannot. The two approaches are further complementary because HTM and the fall-back path cannot execute at the same time; by using AAHTM to reduce the time under lock of the fall-back path we can increase the proportion of the time we spend in parallel executions of HTM transactions.

It is straightforward to enhance an existing HTM runtime with AAHTM. In **htm-10**, for example, when the maximal retry count is reached (10 in this case), the runtime can simply acquires a AAHTM-optimized test-and-test-and-set lock instead of the traditional TATAS lock. Figure 12 shows the drastic improvements of this hybrid approach (**htm-10-fb-aahtm**) on the three benchmarks whose performance collapsed. The use of AAHTM in the fall-back lock extends scalability out to sixteen threads and reduces the fall-off in performance when the benchmark reaches its synchronization bottleneck (the **ssca2** benchmark sees no performance collapse in HTM; its performance is unaffected by the AAHTM fall-back lock).

6) Performance Hazards: While we have shown that AAHTM-based locks can improve performance in many cases, one can imagine scenarios in which they might have a negative impact. In particular, speculators have the potential to "steal" cache lines from the thread that is executing the "real" critical section, thereby slowing down the critical path of the application. We argue that this possibility is highly unlikely to have a significant impact in real-world programs, and can easily be controlled with simple mitigation strategies. (Like all forms of speculation, AAHTM-based locks also impose an energy penalty; we do not assess that here.)

To evaluate the potential for slowdown, we set out to design an adversarial benchmark in which the effect would be as large as possible. We began with the observation that stealing will occur only if the lock holder and speculators perform conflicting accesses to the same cache line, and that the stealing will be a problem only if the lock holder performs a subsequent access to the line (thereby incurring a cache miss). We note, however, that the subsequent access will conflict with the speculator that stole the line, causing its transaction to abort. Multiple misses therefore require either that there are multiple speculators or that a speculator steals multiple lines before the lock holder accesses any of those lines again. The first possibility is easily controlled by limiting speculation. The second possibility is difficult to engineer, even deliberately—particularly if the lock holder and speculators are to execute the same source code.

Our adversarial benchmark is an instance of **ArrayBench** in which the array is very small (10 cache lines), the critical section / transaction performs 100 random writes (and thus many repeated writes), and there is no limit on the number of speculators. As shown in Figure 13a, this scenario does indeed lead to slowdown, on the order of 20%. The slowdown goes away if we scatter the writes across a larger number of addresses, reduce the number of writes to any single address, or perform the writes in sequential order, allowing them to be combined in the processor's write buffer.

Even in the adversarial case, the problem disappears if we use a different speculative code path or simply limit speculation. In ArrayBench, we optimized our locks (**opt**) by adding a protective test at the beginning of the critical section: if the test determines that the thread is speculating (using AAHTM_TEST) and the array size is too small, the speculator immediately aborts itself so that the lock holder will never be slowed down. The resulting performance is shown in Figure 13b. When the protective test is not triggered, speculation proceeds as normal and results are as shown in Figure 7.

```
#pragma omp parallel {
  ...
  for (int task = ...) {
    for (int i = ...) {
      int row = perm[i] + base;
      double sum = b[row];
      for (int j = rowptr[row + 1] − 1; j ≥rowptr[row]; −−j)
        sum −= values[j]*y[colidx[j]];
      y[row] = sum;
    } // for each row
    #pragma omp barrier
  } // for each level
} // omp parallel
```

Figure 14: Code skeleton of BackwardSolver.

input	offshore.mtx (75)			inline_1.mtx (288)			thermal2.mtx (991)		
thd #	baseline	aahtm	speedup	baseline	aahtm	speedup	baseline	aahtm	speedup
1	2.28	2.28	0.00%	3.18	3.18	0.00%	3.83	3.83	0.00%
2	3.87	4.02	3.88%	6.16	6.19	0.49%	3.50	3.51	0.29%
4	5.84	6.75	15.58%	11.26	11.40	1.24%	5.92	6.51	9.97%
8	7.14	8.01	12.18%	19.99	20.53	2.70%	10.46	11.66	11.47%
12	6.40	7.09	10.78%	26.55	27.09	2.03%	13.67	14.97	9.51%
16	5.27	5.43	3.04%	31.60	33.42	5.76%	14.77	16.37	10.83%

Table I: Throughput of BackwardSolver (measured as GB/sec) for three input matrices with different degrees of parallelism. **baseline** uses the default GOMP barrier. **aahtm** uses the AAHTM enhanced barrier (see Section III-D).

B. Barrier

We evaluate our AAHTM-based barrier (described in Section III-D) using **BackwardSolver**, an OpenMP implementation of a backward sparse triangular solver based on level-scheduling with barriers [19]. As shown in the code skeleton of Figure 14, in order to properly handle task dependency, there is a barrier between two successive task levels in the main loop. Due to the non-uniform distribution of elements in sparse matrices, tasks processed by different threads have different lengths, resulting in idle threads at each barrier point.

With the AAHTM-based barrier integrated into the OpenMP library, those idle threads are able to speculatively cross the synchronization point to process their tasks in the next loop iteration. Since accesses to the major matrix structures (values) generally have poor spatial locality and are read-dominated, AAHTM transactions can effectively bring in data that would otherwise be cache misses. If the speculating threads have more work than average in the next iteration, overall performance improves (up to 15%).

Table I presents the performance results of BackwardSolver for three input matrices from the University of Florida Sparse Matrix Collection [20]. For offshore.mtx and thermal2.mtx, we see significant performance improvement when using the AAHTM barrier.

V. Related Work

In proposing always-abort HTM we are building upon two important lines of prior research: hardware transactional memory and thread-level speculation.

Transactional memory was proposed by Herlihy and Moss [21] as a hardware mechanism to simplify the construction of concurrent data structures. Subsequent work has explored both hardware and software implementations. In recent years, hardware implementations have appeared in mainstream processors from Intel [22] and IBM [23], [24]. As result of its wide availability, HTM has been incorporated into a variety of synchronization libraries and general applications. *Lock elision* is a common use case for HTM [25], in which ostensibly lock-protected critical sections are run optimistically under HTM to achieve finer-grain conflict detection. More complex runtimes also elide locks with optimistic software synchronization techniques in conjunction with HTM [1]. HTM and software transactional memory have also been used together in hybrid systems [6], [26], [27]. Beyond standard HTM, looser HTM primitives have also been produced. IBM's *rollback only transactions* remove read tracking from transactions, reducing the size and abort rate of transactions at the expense of semantics [24]. Similar relaxations of the read/write set tracking of HTM are proposed elsewhere [28]. Always-abort transactions with such relaxed semantics could perform better than our proposed AAHTM due to a lower conflict rate.

Hardware speculation is ubiquitous in modern processors, which execute instructions across predicted branches, prefetch data into cache based on observed access patterns, and even guess the values to be returned by load misses. Thread-level speculation is a natural extension that seeks to exploit predictable behavior at a somewhat coarser grain. Always-abort HTM strongly resembles the *thread-level* hardware speculation (TLS) of more ambitious processors. In hardware scouting (or runahead execution), for example, a checkpoint is taken on a load miss, and the processor continues speculatively. Within the speculative execution, no state is committed, and processing of instructions continues, bypassing additional load misses as necessary and using the predictors for branches, until the missed data is fetched. At this point all speculative state is wiped and the processor continues, but with the advantage of a warmed-up data cache, instruction cache, and branch predictor [29], [30], [31]. Simultaneous speculative threading expands on this idea to allow some independent state from the hardware scout to commit once the load is fulfilled; this more advanced technique was incorporated into Sun's Rock processor [32]. Our uses of AAHTM resemble scouting across a lock acquisition rather than a load miss. Others have also investigated the overlap between TLS and HTM, proposing repurposing HTM logic to speculatively parallelize sequential code [33], [34].

VI. Conclusion

Given its potential utility and negligible implementation cost, we believe that always-abort HTM is an attractive feature to include in future HTM implementations. For high contention and large critical sections, it is an excellent way to easily improve performance, with additional applications when integrated into barriers or the HTM fall-back lock.

In ongoing work, we are exploring additional ways to use AAHTM. We are particularly interested in using always-abort HTM to elide other types of busy-waiting, such as that incurred when using synchronous communication (e.g., RDMA or MPI) or on-chip hardware accelerators (e.g., as in IBM's PowerEN [35]). We are also interested in using queue- and stack-based locks to prioritize speculating threads.

As AAHTM never impacts correctness, it seems that compilers should be able to exploit the primitive in specialized passes. In particular, compiler assistance and points-to analysis could be used to generate a faster AAHTM code path that avoids possibly contentious memory accesses and elides unnecessary computation, only retaining instructions necessary to generate memory accesses and control flow. It is possible that AAHTM can be used even in sequential code by spawning a AAHTM run-ahead thread. A more ambitious goal is to use compiler analysis to generate a speculative code path that is guaranteed to avoid the lazy subscription problem, thereby allowing the techniques presented here to be used with traditional HTM.

As AAHTM's intended purpose is different from HTM, it can benefit from slightly different hardware implementations. In particular, traditional HTM need not abort on a read eviction from its local cache, so long as it can detect conflicts on the address (e.g. by using a Bloom filter). Such an optimization is counter-productive for AAHTM, as it could evict useful data and waste speculative effort. Other hardware changes, such as removing conflict detection on the transaction's read/write set, would also improve AAHTM.

Acknowledgment

The authors would like to thank Jongsoo Park for his insights on the use of barriers in HPC applications, and Chen Ding, Ryan Yates, and Dong Chen for their insights into future applications of AAHTM.

References

[1] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir, "Adaptive integration of hardware and software lock elision techniques," in *26th ACM Symp. on Parallelism in Algorithms and Architectures*, Prague, Czech Republic, 2014, pp. 188–97.

[2] L. Xiang and M. L. Scott, "Software partitioning of hardware transactions," in *20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, San Francisco, CA, 2015, pp. 76–86.

[3] A. Kleen, "Scaling existing lock-based applications with lock elision," *Comm. of the ACM*, vol. 57, no. 3, pp. 52–6, 2014.

[4] D. Dice, A. Kogan, and Y. Lev, "Refined transactional lock elision," in *10th ACM SIGPLAN Wkshp. on Transactional Computing*, Portland, OR, 2015.

[5] J. Izraelevitz, A. Kogan, and Y. Lev, "Implicit acceleration of critical sections via unsuccessful speculation," in *11th ACM SIGPLAN Wkshp. on Transactional Computing*, Barcelona, Spain, 2016.

[6] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, "Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory," in *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, CA, 2011, pp. 39–52.

[7] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir, "Hardware extensions to make lazy subscription safe," *arXiv:1407.6968*, 2014.

[8] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, no. 1, pp. 21–65, 1991.

[9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, 2009, pp. 157–168.

[10] N. Diegues and P. Romano, "Self-tuning Intel restricted transactional memory," *Parallel Computing*, vol. 50, pp. 25–52, 2015.

[11] memcached.org, "memcached – a distributed memory object caching system," http://memcached.org/, accessed: 2017.

[12] D. Dice, V. J. Marathe, and N. Shavit, "Lock cohorting: A general technique for designing NUMA locks," *ACM Trans. on Parallel Computing*, vol. 1, no. 2, p. 13, 2015.

[13] M. Pohlack and S. Diestelhorst, "From lightweight hardware transactional memory to lightweight lock elision," in *6th ACM SIGPLAN Wkshp. on Transactional Computing*, San Jose, CA, 2011.

[14] T. Vyas, Y. Liu, and M. Spear, "Transactionalizing legacy code: An experience report using GCC and memcached," in *8th ACM SIGPLAN Wkshop. on Transactional Computing*, Houston, TX, 2013.

[15] libMemcached.org, "libMemcached," http://www.libMemcached.org, 2011.

[16] FAL Labs, "Kyoto Cabinet: a straightforward implementation of DBM," http://fallabs.com/kyotocabinet/, 2011.

[17] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multiprocessing," in *IEEE Intl. Symp. on Workload Characterization*, Seattle, WA, 2008, pp. 35–46.

[18] L. Xiang and M. L. Scott, "Conflict reduction in hardware transactions using advisory locks," in *27th ACM Symp. on Parallelism in Algorithms and Architectures*, Portland, OR, 2015, pp. 234–243.

[19] J. Park, "SpMP library," https://github.com/IntelLabs/SpMP/.

[20] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. on Mathematical Software*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[21] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *20th Intl. Symp. on Computer Architecture*, San Diego, CA, 1993, pp. 289–300.

[22] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The fourth-generation Intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[23] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for IBM System z," in *45th ACM/IEEE Intl. Symp. on Microarchitecture*, Vancouver, Canada, 2012, pp. 25–36.

[24] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust architectural support for transactional memory in the Power architecture," in *40th Intl. Symp. on Computer Architecture*, Tel-Aviv, Israel, 2013, pp. 225–36.

[25] R. Rajwar and J. R. Goodman, "Speculative lock elision: Enabling highly concurrent multithreaded execution," in *34th ACM/IEEE Intl. Symp. on Microarchitecture*, Austin, TX, 2001, pp. 294–305.

[26] Y. Lev, M. Moir, and D. Nussbaum, "PhTM: Phased transactional memory," in *2nd ACM SIGPLAN Wkshp. on Transactional Computing*, Portland, OR, 2007.

[27] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, and P. Romano, "ProteusTM: Abstraction meets performance in transactional memory," in *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Atlanta, GA, 2016, pp. 757–71.

[28] R. Titos-Gil, M. E. Acacio, J. M. Garcia, T. Harris, A. Cristal, O. Unsal, I. Hur, and M. Valero, "Hardware transactional memory with software-defined conflicts," *ACM Trans. on Architecture and Code Optimization*, vol. 8, no. 4, pp. 31:1–20, 2012.

[29] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *9th Intl. Symp. on High-Performance Computer Architecture*, Anaheim, CA, 2003, pp. 129–40.

[30] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay, "High-performance throughput computing," *IEEE Micro*, vol. 25, no. 3, pp. 32–45, 2005.

[31] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *11th Intl. Conf. on Supercomputing*, Vienna, Austria, 1997, pp. 68–75.

[32] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Simultaneous speculative threading: A novel pipeline architecture implemented in Sun's Rock processor," in *36th Intl. Symp. on Computer Architecture*, Austin, TX, 2009, pp. 484–95.

[33] R. Odaira and T. Nakaike, "Thread-level speculation on off-the-shelf hardware transactional memory," in *IEEE Intl. Symp. on Workload Characterization*, Raleigh, NC, 2014, pp. 212–221.

[34] J. Salamanca, J. N. Amaral, and G. Araujo, "Evaluating and improving thread-level speculation in hardware transactional memories," in *IEEE Intl. Parallel and Distributed Processing Symp.*, Chicago, IL, 2016, pp. 586–595.

[35] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel, "Hardware acceleration in the IBM PowerEN processor: Architecture and performance," in *21st Intl. Conf. on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, 2012, pp. 389–400.