Performance Improvement via Always-Abort HTM *

Joseph Izraelevitz
Computer Science Department
University of Rochester
jhi1@cs.rochester.edu

Lingxiang Xiang
Parallel Computing Lab
Intel Corporation
lingxiang.xiang@intel.com

Michael L. Scott
Computer Science Department
University of Rochester
scott@cs.rochester.edu

Abstract

This work proposes and discusses the implications of adding a new feature to hardware transactional memory, allowing a program to specify that a transaction should *always* abort (even if it executes a commit instruction), and is thus guaranteed to be free of side effects. Perhaps counterintuitively, we believe such a primitive can be useful.

Prior art has already noted that HTM transactions, even in failure, can accelerate the subsequent execution of their contents by warming up the branch predictor and caches. However, traditional HTM requires that the programmer properly coordinate between HTM and other synchronization primitives, otherwise data races can occur. With always-abort HTM (AAHTM), no such synchronization is necessary, because there is no risk of accidentally committing a transaction that has seen inconsistent state. We can therefore use AAHTM in scenarios where traditional HTM would be unsafe. In this paper, we present several designs that use AAHTM, discuss preliminary results, and identify other situations in which the new primitive might be useful.

1. Introduction

The wide commercial availability of hardware transactional memory has given programmers a new and fast synchronization primitive. Hardware transactional memory (or HTM), gives the guarantee that all code executed within a *transaction* will appear as a single atomic action from the perspective of other threads. In practice, the hardware transaction leverages the cache coherence protocol to privatize the transacting thread's changes and detect conflicts between it and other threads. If the transaction cannot complete and commit its changes (due to a conflict with another thread, overflow of speculative hardware state, or use of an instruction that cannot easily be isolated) the transaction aborts, reverting all its changes. In the case of an abort, the thread may either retry the transaction or use a different synchronization technique. In contrast with other such techniques (e.g., locks), HTM achieves atomicity by detecting and recovering from conflicts at run time, as opposed to pessimistically preventing them. As a result, HTM may result in higher concurrency than would otherwise be achievable.

Unfortunately, HTM transactions *must* be used in conjunction with other synchronization techniques. There is no guarantee that a transaction will ever succeed, and without a different fall-back mechanism the transacting thread may retry forever without success (it might, for instance, repeatedly overflow its private caches, or run long enough that it is always interrupted by a context switch). Due to this limitation, HTM transactions are often exported to the application programmer via synchronization libraries that contain a variety of cross compatible synchronization mechanisms ([7, e.g.]).

A common idiom for such synchronization libraries is to use a *fall-back lock*. If a thread experiences several HTM aborts in a row, it acquires this lock to guarantee progress. However, care must be taken to ensure that critical sections executing concurrently under the lock and HTM synchronize properly. No HTM transaction should commit its changes while the lock is held: the transaction might have read invalid state, leading it to exhibit undefined behavior. One common solution to this problem is *early subscription*. Immediately upon entering the transaction, the transacting thread reads the lock and ensures it is unheld (thereby ensuing that the protected data is consistent). If any thread acquires the lock while the transaction progresses, the transaction is immediately aborted due to the detected conflict. However, this solution means that a fall-back lock acquisition can abort a large number of concurrent hardware transactions, *even if no true conflict exists* between those transactions and the lock-protected critical section. An appealing (erroneous) solution to this problem is *lazy subscription* to the fall-back lock [4]. Before entering the transaction, the transacting thread reads the fall back *sequence lock* (a lock whose value increments at every acquire and release). After the read, the thread enters the hardware transaction. At the end of the transaction, just before commit, the thread rereads the fall back lock's value. If it hasn't changed, the transaction commits. The problem with lazy subscription is the uncontrollable nature of a transaction's undefined behavior. During a hardware transaction with lazy subscription, code can read inconsistent state and, due to a resulting erroneous indirect branch or a stray store to a return address on the stack, jump to an arbitrary location *including a commit instruction*, thereby bypassing the final check on the lock. In short, both a fall-back lock and early subscription seem necessary to guarantee the correctness of HTM, barring significant hardware changes [5].

Beyond the expected parallelization improvements of using HTM for more fine-grained concurrency, researchers have noted other benefits. In particular, 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 [6, 12, 14, 22]. If a failed transaction executes sufficiently far before aborting, its subsequent attempts (protected by either HTM or the 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 can act as programmer-requested thread-level speculation.

Our work attempts to reap these prefetching benefits 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,

* 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.

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. While in this work we only explore the use of our hardware primitive in synchronization primitives such as locks and barriers, always-abort HTM is likely to be useful wherever waiting is required, such as in synchronous communication or requests to hardware accelerators. Its operation is always safe, since an always-abort transaction can never affect semantic state. In contrast to traditional hardware prefetching, AAHTM is significantly more flexible. 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, this hardware primitive is straightforward to implement—all that is required is a slightly different execution mode. Significantly improved hardware prefetching is likely to require a larger larger hardware investment.

The following discussion explores the uses and utility of always-abort HTM. We begin with a motivating microbenchmark which demonstrates the potential gains and pitfalls of our technique. Using the lessons learned there, we propose a number of synchronization primitives that incorporate AAHTM. We continue with preliminary performance results on small and real-world benchmarks with some discussion of our findings, as well as a review of related work. We conclude with some ideas regarding other potential uses of our hardware primitive.

2. 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.

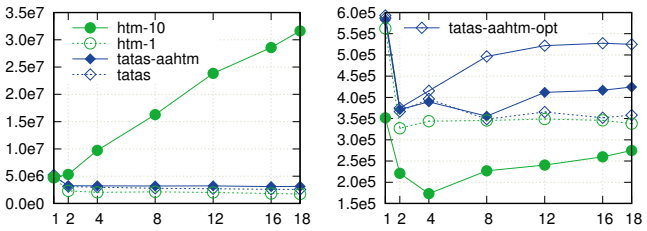
2.1 Performance

To quantify the potential benefits of AAHTM, we begin by exploring a simple microbenchmark—**ArrayBench** (Figure 1)—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 random 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 consider several alternatives. The first is traditional 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 next 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 Figure 2). For our experiments, since AAHTM does not exist, we instead use regular HTM in an unsafe manner, precipitating 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**).



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

Figure 1: ArrayBench throughput in transactions per second for TAS lock and HTM. The X axis is the number of concurrent threads; the Y axis is operations per second.

This experiment also explores an additional advantage of always-abort HTM—namely, that a thread can tell when the mechanism is being used, and can switch 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 accesses (such as highly contended variables or I/O operations), 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. 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 traditional HTM.

2.2 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, such 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.

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 when it begins a new always-abort transaction, 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.

3. 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.

3.1 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 2, including an extension to `pthread_mutex` and a test-and-set lock. Basically, if a thread would normally busy-wait, it enters an always-abort transaction instead. When the transaction eventually fails, it checks to see whether it can exit its busy wait. 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.

3.2 Test-and-Test-and-Set Priority Lock

The test-and-test-and-set lock of Figure 2 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 3) which strictly prioritizes threads that have completed their prefetch and limits the number of concurrently speculating threads. This lock uses two counters: one (`num_spec`) tracks the number of currently active speculative threads; the other (`num_ready`) tracks the number of threads that have completed speculating. If `num_ready` is non-zero, no thread that has not yet speculated can acquire the lock.

3.3 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 [17].

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` 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 4 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

```
1  // we emulate AAHTM using
2  // regular Intel x86 HTM.
3  #define AAHTM_BEGIN \
4   XBEGIN
5  #define AAHTM_TEST \
6   XTEST
7
8  // how we enter AAHTM
9  int enter_aahtm(){
10   if(AAHTM_BEGIN()
11    == HTM_SUCCESSFUL)
12     return 1;
13   // abort
14   return 0;
15 }
16
17 // used for both simple and
18 // prioritized tatas lock
19 struct tatas_lock_t {
20   union{
21     struct{
22       int32_t held;
23       int16_t num_ready;
24       int16_t num_spec;
25     };
26     int64_t all;
27   };
28 };
29
30 // tatas lock with AAHTM
31 // as busy wait-alternative
32 void tatas_lock_aahtm
33  (tatas_lock_t *lk) {
34   if(AAHTM_TEST()){return;}
35   int tries = 0;
36   while(lk→held||
37    tas(&lk→held)){
38     // if lock is held,
39     // start speculating
40     if(enter_aahtm()){return;}
41     else{tries++;}
42     // revert to the lock
43     // if out of tries
44     if(tries≥NUM_TRIES){
45       while(lk→held||
46        tas(&lk→held))
47         pause(INTERVAL);
48       break;
49     }
50   }
51 }
52 void tatas_unlock_aahtm
53  (tatas_lock_t *lk) {
54   if(!AAHTM_TEST())
55     lk→held = 0;
56 }
57
58 // pthreads lock with AAHTM
59 // as busy wait-alternative
60 void pthread_lock_aahtm
61  (pthread_mutex_t *lk) {
62   if(AAHTM_TEST()){return;}
63   int tries = 0;
64   while(pthread_mutex_trylock(lk)
65    ≠0){
66     if(enter_aahtm()){return;}
67     else{tries++;}
68     if(tries≥NUM_TRIES){
69       pthread_mutex_lock(lk);
70       break;
71     }
72   }
73 }
74 void pthread_unlock_aahtm
75  (pthread_mutex_t *lk) {
76   if(!AAHTM_TEST())
77     pthread_mutex_unlock(lk);
78 }
```

Figure 2: Busy-wait alternative

```
80 // tatas priority lock
81 // with AAHTM
82 void tatas_prio_lock_aahtm
83  (tatas_lock_t *lk) {
84   if(AAHTM_TEST()){return;}
85   int tries = 0;
86   tatas_lock_t cp;
87   while(true){
88     cp.all = lk→all;
89     if(cp.num_ready==0){
90       if(!lk→held&&
91        !tas(&lk→held))
92         break;
93     }
94     if(cp.num_spec<MAX_SPECS){
95       int tmp;
96       tmp=fai(&lk→num_spec,1);
97       if(tmp≥MAX_SPECS)
98         fai(&lk→num_spec,-1);
99       else if(enter_aahtm())
100        return;
101      else{
102        fai(&lk→num_spec,-1);
103        fai(&lk→num_ready,1);
104        while
105         (lk→held||
106         tas(&lk→held)){}
107        fai(&lk→num_ready,-1);
108        break;
109      }
110    }
111    pause(INTERVAL);
112  }
113 }
114 void tatas_prio_unlock_aahtm
115  (tatas_lock_t *lk) {
116   tatas_unlock_aahtm(lk);
117 }
```

Figure 3: Test-and-set lock with priority

```
118 struct ticket_lock_t {
119   int next_ticket;
120   int now_serving;
121 };
122 void ticket_lock_aahtm
123  (ticket_lock_t *lk) {
124   if(AAHTM_TEST()){return;}
125   int tries = 0;
126   int my_ticket =
127    fai(&lk→next_ticket, 1);
128   int dist = 0;
129   while((dist=(my_ticket-
130      lk→now_serving))>0){
131     if(dist≤MAX_DIST &&
132      dist≥MIN_DIST &&
133      tries<NUM_TRIES){
134       if(enter_aahtm())
135         return;
136       else{
137         busy_wait();
138         tries++;
139       }
140     }
141     else
142       pause(INTERVAL);
143   }
144 }
145 void ticket_unlock_aahtm
146  (ticket_lock_t *lk) {
147   if(!AAHTM_TEST())
148     lk→now_serving++;
149 }
```

Figure 4: Ticket lock

```
150 struct barrier_t {
151   int total;
152   // number of threads
153   // to wait for
154   int gen;
155   // generation counter
156   // with flags in low bits
157   char[] padding;
158   // padding to avoid
159   // false sharing
160   int awaited;
161   // number of threads
162   // at the barrier
163 };
164 void barrier_wait
165  (barrier_t *bar){
166   int state = bar→gen;
167   if(fai(&bar→awaited,-1)==0){
168     bar→awaited=bar→total;
169     state+=1;
170     bar→gen = state;
171     return;
172   }
173   int gen=state;
174   int tries=0;
175   do{
176     if(bar→gen==gen){
177       tries++;
178       if(enter_aahtm()){
179         if(bar→gen ≠gen)
180           XABORT();
181         else
182           return;
183       }else{
184         if(tries<NUM_TRIES)
185           continue;
186         else
187           pause(INTERVAL);
188       }
189     }
190     gen = bar→gen;
191   }while(gen≠state+1);
192 }
```

Figure 5: Barrier (heavily based on GNU libGOMP)

monitoring the `now_serving` counter. With a similar mechanism, we can also control how many threads are concurrently speculating.

3.4 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.

4. Evaluation

Our experiments were conducted on a 18-core Intel Xeon E5-2697 v4 (Broadwell) machine running Linux kernel version 3.10.0. Code was compiled with `gcc` 5.3.0 using the `-O3` optimization flag. Reported results show the average of three runs at each configuration point, and no major performance variation was seen across runs.

4.1 Locks

Our lock library is 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 lock 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 the tatas lock.

tatas-aahtm: The trivial AAHTM test-and-test-and-set lock of Section 3.1. We set `NUM_TRIES` to 4 based on experimentation for reasonable parameters for generic workloads.

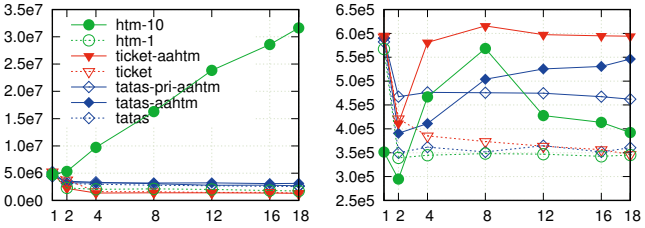
tatas-pri-aahtm: The prioritizing AAHTM test-and-test-and-set lock of Section 3.2. We set `NUM_TRIES` to 4 and `MAX_SPECS` to 1.

ticket-aahtm: The AAHTM ticket lock of Section 3.3. We set `NUM_TRIES` and `MAX_DIST` to 4 and `MIN_DIST` to 2.

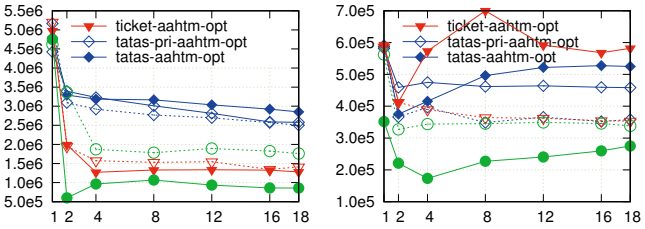
We ran our locks on several microbenchmarks and real-world applications. **ArrayBench** (Figure 6) is the microbenchmark we introduced in Section 2.1. In it, threads contend to access an integer array with one million elements. Each thread generates addresses to touch within the array before contending to enter the critical section. Tests run for approximately five seconds. 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 6) 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 6a). 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 give them an advantage over the fair ticket locks by allowing 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 7) we once again contend on a global data structure—this time, a `std::map<int,int>` (red-black tree). 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

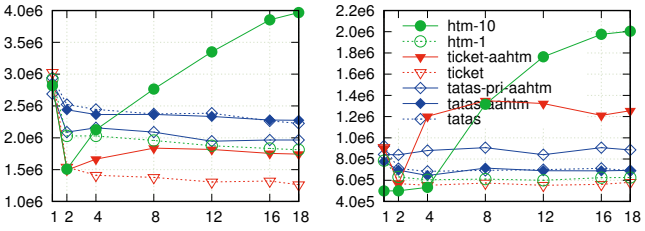


(a) low contention, small footprint. (b) low contention, large footprint.

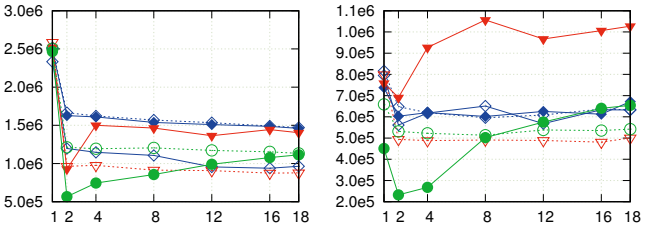


(c) high contention, small footprint (d) high contention, large footprint

Figure 6: Throughput (critical sections per second) of ArrayBench on Broadwell Xeon.



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



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

Figure 7: Throughput (operations per second) of MapBench on Broadwell Xeon.

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.

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 7d).

```
193 #pragma omp parallel {
194   ...
195   for (int task = ...) {
196     for (int i = ...) {
197       int row = perm[i] + base;
198       double sum = b[row];
199       for (int j = rowptr[row + 1] - 1; j ≥rowptr[row]; --j)
200         sum -= values[j]*y[colidx[j]];
201       y[row] = sum;
202     } // for each row
203     #pragma omp barrier
204   } // for each level
205 } // omp parallel
```

Figure 8: 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 1: 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 3.4).

4.2 Barrier

We evaluate our AAHTM-based barrier (described in Section 3.4) using **BackwardSolver**, an implementation of a backward sparse triangular solver based on level-scheduling with barriers [19]. As shown in the code skeleton of Figure 8, in order to properly handle task dependency, there is a barrier between two successive task levels in the main loop (line 203). 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, those waiting 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 (line 200), 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 1 presents the performance results of BackwardSolver for three input matrices from the University of Florida Sparse Matrix Collection. For `offshore.mtx` and `thermal2.mtx`, we see significant performance improvement when using the AAHTM barrier.

5. 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 and Moss [11] 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 [10] and IBM [1, 13]. 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 [20]; it is a technique in which lock-protected critical sections are instead run optimistically under HTM to achieve finer-grain conflict detection. More complex runtimes also elide locks with additional optimistic software synchronization techniques in conjunction with HTM [7]. HTM and software transactional memory have also been used together in hybrid systems [4, 8, 16]. 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 [1]. Similar relaxations of the read/write set tracking of HTM are proposed elsewhere [21]. 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 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 [2, 9, 18]. 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 [3]. Our uses of AAHTM resemble scouting across a lock acquisition rather than a load miss.

6. Conclusion

Given its potential utility and negligible implementation cost, we believe that always-abort HTM would be 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.

In ongoing work, we are exploring additional ways to use AAHTM, and looking for real-world programs that will benefit from its use. 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 [15]). We are also interested in using queue- and stack-based locks to prioritize speculating threads, and in adding AAHTM to other synchronization primitives such as condition variables.

Acknowledgments

We would like to thank Jongsoo Park for his insights on the use of barriers in HPC applications.

References

[1] 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 *Proc. of the 40th Intl. Symp. on Computer Architecture*, ISCA '13, pages 225–236, Tel-Aviv, Israel, 2013.

[2] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-performance throughput computing. *IEEE Micro*, 25(3):32–45, May 2005.

[3] 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 *Proc. of the 36th Intl. Symp. on Computer Architecture*, ISCA '09, pages 484–495, Austin, TX, USA, 2009.

[4] 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 *Proc. of the 16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 39–52, Newport Beach, CA, USA, 2011.

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

[6] D. Dice, A. Kogan, and Y. Lev. Refined transactional lock elision. In *10th ACM SIGPLAN Wkshp. on Transactional Computing*, TRANSACT '15, Portland, OR, USA, 2015.

[7] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir. Adaptive integration of hardware and software lock elision techniques. In *Proc. of the 26th ACM Symp. on Parallelism in Algorithms and Architectures*, SPAA '14, pages 188–197, Prague, Czech Republic, 2014.

[8] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, and P. Romano. ProteusTM: Abstraction meets performance in transactional memory. In *Proc. of the 21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 757–771, Atlanta, GA, USA, 2016.

[9] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proc. of the 11th Intl. Conf. on Supercomputing*, ICS '97, pages 68–75, Vienna, Austria, 1997.

[10] 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*, 34(2):6–20, 2014.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, ISCA '93, pages 289–300, San Diego, CA, USA, 1993.

[12] J. Izraelevitz, A. Kogan, and Y. Lev. Implicit acceleration of critical sections via unsuccessful speculation. In *11th ACM SIGPLAN Wkshp. on Transactional Computing*, TRANSACT '16, Barcelona, Spain, 2016.

[13] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System z. In *Proc. of the 45th ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO 45, pages 25–36, Vancouver, Canada, 2012.

[14] A. Kleen. Scaling existing lock-based applications with lock elision. *Commun. ACM*, 57(3):52–56, Mar. 2014.

[15] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel. Hardware acceleration in the ibm poweren processor: Architecture and performance. In *Proc. of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques*, PACT '12, pages 389–400, Minneapolis, MN, USA, 2012.

[16] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *2nd ACM SIGPLAN Wkshp. on Transactional Computing*, TRANSACT '07, Portland, OR, USA, 2007.

[17] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, Feb. 1991.

[18] 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 *Proc. of the 9th Intl. Symp. on High-Performance Computer Architecture*, pages 129–140, Anaheim, CA, USA, 2003.

[19] J. Park. SpMP (sparse matrix pre-processing) library. `https://github.com/IntelLabs/SpMP/`.

[20] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. of the 34th ACM/IEEE Intl. Symp. on Microarchitecture*, MICRO 34, pages 294–305, Austin, TX, USA, 2001.

[21] 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*, 8(4):31:1–31:20, Jan. 2012.

[22] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *Proc. of the 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '15, pages 76–86, San Francisco, CA, USA, 2015.