Conflict Reduction in Hardware Transactions Using Advisory Locks*

Lingxiang Xiang and Michael L. Scott
Computer Science Department, University of Rochester
{lxiang, scott}@cs.rochester.edu

ABSTRACT

Preliminary experience with hardware transactional memory suggests that aborts due to data conflicts are one of the principal obstacles to scale-up. To reduce the incidence of conflict, we propose an automatic, high-level mechanism that uses advisory locks to serialize (just) the portions of the transactions in which conflicting accesses occur. We demonstrate the feasibility of this mechanism, which we refer to as *staggered transactions*, with fully developed compiler and runtime support, running on simulated hardware.

Our compiler identifies and instruments a small subset of the accesses in each transaction, which it determines, statically, are likely to constitute initial accesses to shared locations. At run time, the instrumentation acquires an advisory lock on the accessed datum, if (and only if) prior execution history suggests that the datum—or locations "downstream" of it—are indeed a likely source of conflict. Policy to drive the decision requires one hardware feature not generally found in current commercial offerings: nontransactional loads and stores within transactions. It can also benefit from a mechanism to record the program counter at which a cache line was first accessed in a transaction. Simulation results show that staggered transactions can significantly reduce the frequency of conflict aborts and increase program performance.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.4 [**Programming Languages**]: Processors—*Code generation*

General Terms

Algorithms, Design, Performance

*This work was supported in part by grants from the National Science Foundation (CCF-0963759, CCF-1116055, CNS-1116109, CNS-1319417, CCF-1337224, and CCF-1422649) and by the IBM Canada Centres for Advanced Studies (CAS).

SPAA'15, June 13–15, 2015, Portland, OR, USA.

DOI: http://dx.doi.org/10.1145/2755573.2755577 ...$15.00.

Benchmark	S	%I	W/U	Contention Source	LA	LP
list-hi	1.0	27%	4.92	linked-list	N	Y
tsp	3.6	10%	1.53	priority queue	Y	Y
memcached	2.6	25%	3.11	statistics information	Y	Y
intruder	3.2	32%	4.02	task queue	Y	Y
kmeans	4.6	35%	3.57	arrays	N	Y
vacation	9.7	1%	0.34	red-black trees	N	Y

Table 1: HTM contention in representative benchmarks. S: speedup with 16 threads over sequential run. %I: % of txns forced into irrevocable mode. W/U: ratio of wasted to useful cycles in transactions. LA: locality of contention addresses. LP: locality of contention PCs.

Keywords

Hardware Transactional Memory, Advisory Lock

1. INTRODUCTION

Transactional Memory combines the convenience of atomic blocks as a programming idiom with the potential performance gains of speculation as an implementation. With the recent deployment of hardware support (HTM) on Intel's Haswell [33], IBM's zEC12 [13] and IBM's POWER 8 [4], we anticipate a "virtuous cycle" in which more and more multithreaded programs employ TM, and the performance of the hardware becomes increasingly important.

Preliminary experience with HTM suggests that the raw performance, scalability, and energy consumption of transactional programs—especially those with medium- or large-scale transactions—are all limited by the hardware transaction abort rate [33]. Aborts happen for two main reasons: hardware overflow and conflicts with other transactions. We have addressed the overflow case in prior work [31]; we focus here on conflicts.

Conflicts among transactions lead to poor scalability and energy waste. Table 1 shows several representative TM programs running on a typical 16-core eager HTM system (the experimental methodology is described in Section 6). Repeated aborts limit speedup (column "S") and force many transactions to acquire a global lock and revert to irrevocable mode (column "%I") in order to make forward progress. A high ratio of cycles spent in aborted transactions relative to committed transactions ("W/U" column) correlates strongly with energy waste. As shown in the final, vacation row, wasted work can still be significant even in programs with reasonable speedup.

Several previous projects have proposed hardware mechanisms to reduce the incidence of conflict aborts. Examples include DATM [21], RETCON [3], Wait-n-GoTM [14], and

OmniOrder [20]. While these proposals achieve nontrivial improvements in abort rates and system throughput, they suffer from several limitations. First, most entail significant new hardware complexity—e.g., for the transactional-cycle detector of Wait-n-GoTM or the 8 new coherence protocol states of DATM. Second, they tend to target specific patterns of contention—e.g., the simple conflicting code slices of RETCON or the circular dependences of DATM and Wait-n-GOTM; they may miss other opportunities. Third, they are often specific to particular styles of HTM—e.g., lazy for RETCON, eager for others—and may not apply across the rest of the HTM design spectrum.

As an alternative to all-hardware mechanisms, we introduce the notion of *Staggered Transactions*, an automatic technique to reduce the frequency of aborts. Staggered Transactions serialize the execution of conflict-heavy portions of transactions by means of advisory (optional) locks, implemented using nontransactional loads and stores. Nonconflicting code continues to execute speculatively in parallel, thereby maintaining scalability. Because correctness remains the responsibility of the underlying TM system, Staggered Transactions function correctly even if some transactions neglect to obey the locking protocol. Moreover, the contention reduction achieved is largely independent of other HTM implementation details; in particular, it should be compatible with most conflict resolution techniques.

Effective implementation of Staggered Transactions requires a combination of compile-time and run-time mechanisms to choose *whether* to acquire a lock and, if so, *which* lock to acquire and *where* to acquire it. In our (fully automated) system, the compiler uses Data Structure Analysis [15] to identify and instrument a small subset of the accesses in each transaction, which it determines, statically, are likely to constitute initial accesses to shared locations. These accesses constitute *advisory locking points*. At run time, a locking policy decides which of these locking points to activate, and which lock to acquire at each, based on prior execution history. As we will shown in Section 6, a careful fusion of compile-time and run-time information allows us to avoid not only simple, repetitive contention but also more complex cyclic dependences, in which each transaction in the cycle accesses a (typically dynamically chosen) location on which the next transaction has already performed a conflicting access.

To escape isolation when acquiring locks, Staggered Transactions require nontransactional loads and stores within transactions. They also benefit from a hardware mechanism to identify the PC at which a conflicting location was first accessed. Neither of these features is common on existing machines, but both appear orthogonal to most other HTM features, and neither seems prohibitively difficult to add.

Our principal contributions, discussed in the following sections, include

- A hybrid optimistic / pessimistic execution model for hardware transactions, in which the most commonly conflicting portions of transactions are serialized by advisory locks, built with nontransactional loads and stores.
- Compiler techniques to insert required instrumentation with negligible impact on run-time overhead.
- Run-time techniques to detect contention and enable advisory locks that avoid it, using high-level program knowledge learned by the compiler.

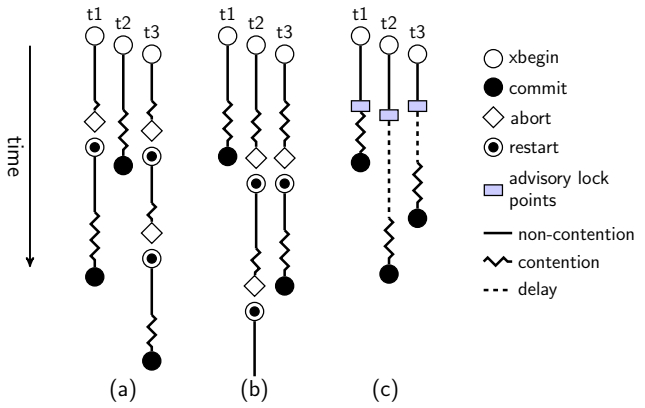

Figure 1: Execution of 3 transactions (t1–t3) (a) on an eager HTM, (b) on a lazy HTM, and (c) with Staggered Transactions optimization.

2. OVERVIEW

Unlike conventional TM, Staggered Transactions force portions of transactions to serialize; the remaining portions run concurrently. Specifically, when conflicts tend to arise in the middle of a given atomic block, our compiler and runtime arrange to acquire an advisory lock at the end of the contention-free prefix, and to hold it until commit time.

Figure 1 illustrates the execution of three transactions with mutually conflicting accesses, shown with a diamond symbol. On a typical eager (Figure 1a) or lazy (Figure 1b) HTM system, only one of the transactions will be able to commit if all three execute concurrently. In a Staggered Transactions system (Figure 1c), execution of the portion of a transaction beginning with the conflicting access is preceded by an *advisory locking point* (ALP). In the diagram, `t1`, `t2`, and `t3` all attempt to acquire the same advisory lock. Transaction `t1` acquires it first; the others wait their turn, and all are able to commit. We say that the conflicting portions of the transactions have been *staggered*. While a transaction could, in principle, acquire multiple advisory locks, we acquire only one per transaction in this paper.

Accurate identification of all and only the contention-prone portions of transactions requires careful coordination of compile-time and run-time techniques. Figure 2 illustrates how the pieces fit together in our system.

The main goal of the compile-time steps (①–③) is to statically insert ALPs in the source code, so that once a frequently conflicting access is found during execution, the runtime can activate the nearest ALP ahead of that access, preventing concurrent accesses to the same object in other transactions. A dedicated compiler pass inspects the IR (intermediate representation) of all atomic blocks (①) and considers load and store instructions as *anchors*—accesses in front of which to insert an ALP (②). To minimize the number of ALPs (and thus the run-time overhead of instrumentation), the compiler uses Data Structure Analysis [15] to select as anchors only those instructions that are likely to be initial accesses to shared objects (data structure nodes) inside a transaction. To help the runtime locate ALPs quickly, the compiler builds an *anchor table* that maps PC ranges to the closest prior ALP in each atomic block (③).

The run-time steps (④–⑧) focus on two decisions concerning advisory locks: (1) whether the runtime should acquire an advisory lock for the current transaction, and if so,

(2) which advisory lock to acquire, at which ALP. The first decision is made by tracking abort history for every atomic block. When an atomic block runs on HTM (④) and a contention abort occurs (⑤), the hardware-triggered handler receives an indication of the conflicting address and (ideally) the program counter at which that address was first accessed (⑥). The runtime appends this information to a per-thread *abort history table* for the current atomic block. Based on the frequency of contention aborts, a software *locking policy* makes decision (1).

Decision (2) is harder. As shown in Table 1 ("LA" and "LP" columns), the program counters associated with initial accesses to conflicting locations are often the same across dynamic transactions, but the accessed data locations often differ: sometimes a common datum is responsible for most aborts; other times, it differs from one transaction instance to another.

The locking policy augments its understanding of the conflict pattern each time a contention abort occurs (⑦). Once a pattern is found, the runtime consults the anchor table to identify an ALP and activate it for future instances of the transaction (⑧). For simplicity, we currently allow only one active ALP for a given atomic block. We also employ a fairly simple policy (more complex possibilities are a subject of future work). Specifically, we activate an ALP only if it corresponds to a PC that has frequently performed the initial access to data that subsequently proved to be the source of conflict. If the addresses of the data in these conflicts were usually the same, the ALP is activated in *precise mode*: it acquires a lock only if the data address in the current transaction instance matches the address of past conflicts. If data addresses have varied in past conflicts, the ALP is activated in *coarse-grain* mode: it acquires a lock regardless of the current data address. In either case, the choice of which lock to acquire is based on the current data address: this will always be the same in precise mode; it may vary in coarse-grain mode.

If conflicts continue to be common despite the activation of a coarse-grain ALP, the runtime uses information gathered by the compiler to activate the *parent* of the ALP instead. The notion of parents again leverages Data Structure Analysis. In a linked data structure, if node B is reached via a pointer from node A, we say that the ALP associated with the initial access to A is the parent of the ALP associated with the initial access to B. In code that traverses a linked list, for example, each node other than the first is accessed through its predecessor; the first is accessed through the head node. In typical traversal code, nodes within the list will share an ALP (embedded in a loop). The parent of that ALP will be the ALP of the head node. Interested readers may consult Lattner's thesis for details [15].

As a simple example, suppose in Figure 2 that q→head is a frequent source of conflicts. After a few aborts, the locking policy will realize that most conflicts happen on the data address q→head, whose initial access in the transaction is usually at the same instruction address, say Addr. The policy categorizes the conflict pattern as *precise* and the runtime activates the ALP right before Addr.

Like the advisory locks of database and file systems, Staggered Transaction advisory locks are purely a performance optimization. Correctness is ensured by the underlying TM system. If the runtime fails to instrument a transaction that participates in a conflict, the only consequence is continued

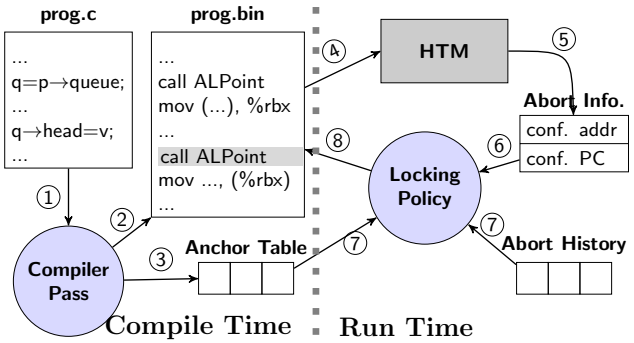

Figure 2: An overview of Staggered Transactions, including compile-time steps ①–③ and run-time steps ④–⑧.

aborts. Likewise, when a transaction *does* attempt to acquire a lock, it need not wait forever—to avoid blocking on a very long transaction (or, on hardware that supports it, a transaction that has been preempted), an ALP can specify a timeout for its acquire operation, and simply proceed when the timeout expires.

3. COMPILER SUPPORT

Our Staggered Transactions system uses a compiler pass for static insertion of advisory locking points (ALPs). Using Data Structure Analysis [15], the compiler identifies and instruments only those loads and stores (anchors) that are likely to constitute initial accesses to shared locations. It also generates an *anchor table* for each atomic block to describe the anchors and non-anchors (uninstrumented loads and stores) and the relationships among them. Anchor tables are subsequently consulted by the runtime to make locking decisions and to locate desired advisory locking points.

Compared to naive instrumentation of every load and store, our technique significantly reduces the number of ALPs, thereby minimizing execution overhead. Moreover, by capturing information about the hierarchical structure of program data, the generated anchor tables allow the runtime to make better locking decisions.

3.1 A Data Structure Approach

We consider program-level objects (data structure nodes) to be an appropriate granularity for the insertion of advisory locking points, based on two observations: First, fields of the same object frequently fit together in just a few cache lines. Since HTM systems typically detect conflict at cache-line granularity, if there is contention on the object, the contended code region will usually start at the first instruction that accesses the data structure node. Second, assuming modular program structure, instructions that access the same object are often concentrated in a short code segment. This implies that the initial access to an object in a given function or a method is likely to be a good anchor candidate. Once these candidates are instrumented, we can skip a number of following loads and stores that access the same objects, without losing the ability to trace back from an abort to an appropriate ALP.

A field-sensitive Data Structure Analysis can be used to identify the objects associated with loads and stores and their alias relationships. We base our compiler pass on Lattner's DSA [15], which we use essentially as a black box. DSA distinguishes pointer-to sets according to the data

structure type and field to which they point. It has previously been used for improved heap allocation [16] and type safety checking. A complete DSA pass has three stages: (1) A *local* stage creates a data structure node (DSNode) for each unique pointer target in a function, and links each pointer access to a DSNode. All pointers linked to the same DSNode may point to the same instance of a data structure. If a pointer field in a data structure points to another data structure (or itself), there will be an outgoing edge from this DSNode to the target DSNode. All DSNodes of a function are organized together as a data structure graph (DSGraph). (2) A *bottom-up* stage merges callees' local DSGraphs into those of their callers. (3) A *top-down* stage merges callers' DSGraphs into those of their callees. We utilize only the result from stage 2, which is more context sensitive than that of stage 3 (to get more accurate alias results, which we do not need, the latter may collapse too many DSNodes).

Given DSA results, our algorithm works in three stages of its own, which we describe in Sections 3.2–3.4.

3.2 Building Local Anchor Tables

A local anchor table keeps information for all loads and stores of a function directly or indirectly called in an atomic block (static transaction). Each load/store is described by a table entry, ATEntry, a 4-field tuple: (instr, isAnchor, parent, pioneer). Field instr indicates the location of the load/store instruction. Field isAnchor is a Boolean flag. We call a load/store an *anchor* if it is the initial access to a DSNode in a possible execution path. (ALPs are placed before anchors in a later stage.) Field parent points to another anchor though which a pointer to the current node was loaded. For example, in the code sequence {B = A→child; ... = B→size;}, if the loads of A→child and B→size are anchors, then the first one is the second one's parent. For a non-anchor access, field pioneer points to the anchor that accesses the same DSNode. For example, in {n = queuePtr→head; ...; queuePtr→tail = m;}, the pioneer of the store to queuePtr→tail is the load of queuePtr→head (assuming that load is an anchor).

Algorithm 1 shows how the compiler constructs an anchor table for a given function. The first stage (lines 3–14) categorizes load and store instructions as anchors or non-anchors through a depth-first traversal of the function's dominator tree. The second stage (lines 15–19) sets up the parent relationship among anchors, using the result of DSA's field-sensitive analysis.

3.3 Building Unified Anchor Tables

Walking from callers to callees, a top-down stage creates one unified anchor table for each atomic block. This stage doesn't change the local tables. Instead, it clones and merges them, taking account of the DSNode mapping from caller to callee at function call sites. Missing parent information for certain anchors, if passed via function arguments, is filled in at this stage. From the construction procedure, we can see that unified anchor tables are context-sensitive: anchors originating from the same instruction may have different parents in different unified anchor tables.

3.4 Instrumentation

Once per-function local anchor tables and per-atomic-block unified anchor tables are available, the insertion of advisory locking points is straightforward: the compiler iterates over all local tables and inserts calls to ALPoint right before each anchor. Every ALP is assigned a unique ID so that the run-time locking policy can locate and activate it. After the binary code has been generated, the compiler knows the real PC of each instruction. It makes the unified anchor tables indexable by PC address, and then emits all unified tables.

Algorithm 1: BuildLocalAnchorTable(func)

```
   Input: a function func
   Output: a local anchor table aTable
1  aTable ← ∅;
2  domTree ← GetDominateTree (func);
3  foreach BasicBlock b in DepthFirstVisit (domTree) do
4      foreach LoadStoreInst inst in b do
5          dsNode ← GetDSNode (inst.pointerOperand);
6          entry ← new ATEntry;
7          entry.inst ← inst;
8          if ∃ m ∈ aTable[dsNode]: m.inst dominates inst
           then
9              entry.isAnchor ← false;
10             entry.pioneer ← m.inst;
11         else
12             entry.isAnchor ← true;
13             entry.parent ← nil;
14         aTable[dsNode].push (entry);
15 foreach DSNode n in aTable do
16     foreach Edge e in n.edges do
17         foreach ATEntry m in aTable[e.toNode] do
18             if m.isAnchor then
19                 m.parent ← n;
```

```
TM_BEGIN(); // genome/sequencer.c:292
...
for (ii = i; ii < ii_stop; ii++) {
  void* segment = vector_at(segmentsContentsPtr, ii);
  TMhashtable_insert(uniqueSegmentsPtr, segment, segment);
}
TM_END();

void* vector_at (vector_t* vectorPtr, long i){ //lib/vector.c:164
  if ((i < 0) || (i ≥vectorPtr→size))          ▶ A 51: Parent 0
    return NULL;
  return (vectorPtr→elements[i]);              ▶ 53: Pioneer 51

// lib/hashtable.c:171
bool_t TMhashtable_insert (hashtable_t* hashtablePtr, ...) {
  long numBucket = hashtablePtr→numBucket;     ▶ A 42: Parent 0
  ...
  ... = TMlist_find(hashtablePtr→buckets[i],
                    &findPair);                ▶ 46: Pioneer 42

void* TMlist_find (list_t* listPtr, ...) { // lib/list.c:588
  list_node_t* nodePtr;
  list_node_t* prevPtr = &(listPtr→head);
  for (nodePtr=(list_node_t*)prevPtr→nextPtr; ▶ A 35: Parent 42
       nodePtr != NULL;
       nodePtr = (list_node_t*)nodePtr→nextPtr) ▶ 38: Pioneer 35
```

Figure 3: An atomic block in genome. ▶ marks an entry with its ID and parent/pioneer field in the unified anchor table.

3.5 Example

Figure 3 presents a portion of the generated anchor table for an atomic block in the STAMP genome benchmark. The compiler pass first runs DSA for functions called inside the atomic block (vector_at, TMhashtable_insert, and TMlist_find). Function TMlist_find, for example, contains a single DSNode (for nodePtr and prevPtr) with two loads on it. The first load is marked as an anchor (A 35), according to

Algorithm 1; the second is a non-anchor (38) whose pioneer is A 35. A 35's parent field is not filled in until the unified anchor table is constructed.

As we can see, the parent chain between anchors (from hashtablePtr to listPtr) is preserved in the unified anchor table, through which the runtime can make advanced locking decisions. For this code snippet, the compiler will finally instrument three loads (vectorPtr→size, hashtablePtr→numBucket, and prePtr→nextPtr) as advisory locking points.

4. HARDWARE REQUIREMENTS

Staggered Transactions require — or can benefit from — several hardware features. First, they must be able to acquire an advisory lock from within an active hardware transaction. This operation violates the usual isolation requirement, but is compatible with several real and proposed HTM architectures. Second, they must be able to identify the data address that has been the source of a conflict leading to abort. This capability is supported by all current HTM designs. Third, they can benefit from a mechanism to identify the address of the instruction that first accessed an address that was the source of a data conflict. While no such mechanism is available on current hardware, it seems relatively straightforward to provide. Moreover, unlike the hardware features required by systems like DATM [21] and Wait-n-GoTM [14], both intra-transaction lock acquisition and initial access recording appear to be independent of other details of HTM design.

Advisory lock acquisition is most easily performed using *nontransactional loads and (immediate) stores*. Nontransactional loads have appeared in Sun's Rock processor [6] and AMD's ASF proposal [8]. They allow a transaction to see writes that have been performed (by other threads) since the beginning of the current transaction. They do not modify the transaction's speculative state: that is, a store by another thread to a nontransactionally-read location will not cause the transaction to abort (unless the location has also been accessed transactionally). Nontransactional stores similarly allow a transaction to update locations even in the event of an abort, and (in the absence of transactional loads or stores to the same location) will not lead to aborts due to conflicting accesses in other threads. To be of use in acquiring an advisory lock, a nontransactional store must (as in ASF) take place immediately; it cannot (as in Rock or IBM's z series machines [13]) be delayed until the end of the transaction.

On ASF, Rock, and z, nontransactional accesses are performed by special load and store instructions. On IBM's POWER 8 [4], a transaction can *suspend* and *resume* execution. While suspended, its loads and stores are nontransactional, and its stores immediately visible. Significantly, nontransactional accesses have a variety of other uses, including hybrid TM [9, 25]; debugging, diagnostics, and statistics gathering [11]; and ordered speculation (e.g., for speculative parallelization of loops).

Conflict information of some sort is provided on abort by all current HTM designs. All, in particular, provide the address of the location (or cache line) on which a conflict has occurred. For Staggered Transactions, a machine would also, ideally, provide the address of the instruction at which the conflicting datum was *initially* accessed in the transaction. We call this address the *conflicting PC*. Note that it is generally *not* the current PC at the time the conflict is discovered.

```
struct ABContext { // per thread, per atomic block
  int activeAnchor; // ID of the currently active anchor
  int blockAddress; // probable conflicting memory address
  AbortInfo abtHistory[NUM_HISTORY]; // abort history
  const AnchorTable *anchorTable; // pointer to anchor table
};
```

Figure 4: ABContext structure.

In an HTM system implemented in the L1 cache, conflicting PC information could be maintained by adding a PC tag to each cache line, in addition to transaction status bits. This tag would be set whenever a line transitions into speculative mode. Because the tag is inspected only when the line is the source of a data conflict, it need not ever be cleared. As we shall see in Section 6, one can in fact get by with just a subset of the PC (e.g., the 12 low-order bits). This suffices to keep the space overhead under 2.4%.

Software Alternatives to Conflicting PC.

While commercial HTMs have begun to support nontransactional loads and stores, hardware recording of the conflicting PC is still a missing feature. Without it, we need a software alternative to map a conflicting data address to the appropriate anchor in instruction space. A relatively cheap solution is to keep a map M for each thread, indexed by cache line address. At every ALP (with anchor ID I, preceding a load/store of some data address A), the runtime can use nontransactional loads and stores to set M(A) to I, if A was previously absent. If a conflict subsequently occurs on address A, M(A) can be used to identify the ALP that should, perhaps, be activated. While the overhead of this method is nontrivial, it can sometimes be acceptable. Section 6.2 compares this method with hardware-supported conflicting PC tracking.

5. RUNTIME SUPPORT

Our compiler assigns a unique ID to each source code atomic block. For each of these, the runtime maintains an ABContext data structure, as shown in Figure 4, for each executing thread. A pointer to the ABContext for the current atomic block and thread is loaded into a local variable at the beginning of each transaction, and accessed at each ALP. If the transaction aborts, the ABContext may also be accessed by the policy for ALP activation.

5.1 Instrumentation

In our current implementation, each ALP comprises a call to the ALPoint function, shown in Figure 5. The function takes three arguments—a pointer to the appropriate ABContext, the ID of the anchor, and the data address accessed in the following load or store instruction.

The ALPoint function acquires an advisory lock if the current anchor is active (line 2) and either the cache line of the data address matches that of the address in the ABContext, or the ALP is in coarse-grain mode (indicated by c→blockAddress == 0). This disjunction is checked by function IsAddressMatched (line 3). When a lock is acquired, c→activeAnchor is cleared to avoid additional locking attempts within the current transaction (line 4). The advisory lock is released when the transaction commits or aborts. The activeAnchor field is restored the next time the thread begins a transaction for the same atomic block.

```
1  void ALPoint (ABContext *c, int myID, void* addr) {
2      if (c→activeAnchor == myID &&
3        IsAddressMatched(c→blockAddress, addr) {
4          c→activeAnchor = 0;
5          AcquireLockFor(addr);
6      }
7  }
```

Figure 5: ALPoint instrumentation function.

The actual blocking/waiting is performed in function AcquireLockFor. In our implementation, this function uses a hash of the data address to choose one of a static set of pre-allocated locks, which it then accesses using nontransactional loads and stores. In a system that allows a transaction to remain active when its thread is preempted or blocked, it is important not to force all other transactions to wait if the stalled one holds a lock. With appropriate OS support, the runtime could register the location of any advisory lock it acquires, and the kernel could free this lock when descheduling the thread. Alternatively, a transaction that waits "too long" for an advisory lock could simply time out and proceed without it.

5.2 Locking Policy

The locking policy serves to predict, based on past behavior, which instructions are likely to constitute the first access within a transaction to a location that is likely to be the source of a conflict. Based on this prediction, the runtime activates an appropriate ALP. Many policies are possible. We describe our current choice, which is simple and seems to perform well in practice.

Table 1 suggests that the conflicting data address alone may not be a predictor of contention, due to its poor locality in certain access patterns. Likewise, PC alone is an overpredictor: while the same instruction is very often the initial access to a conflicting location, there are often cases in which that instruction accesses a location that is *not* a source of conflict. These observations suggest that the combination of PC and data address might make an effective predictor. A policy based on this idea appears in Figure 6. The policy works on a per-thread, per-atomic block basis. Function ActivateALPoint is called on an abort. Depending on the frequency with which the current conflicting data address (line 6) and initially-accessing PC (line 7) have appeared in the recent past, the policy chooses one of four behaviors:

Precise Mode. Both the conflicting PC and the data address appear multiple times in the history. This is typical of statistics and bookkeeping information, or of cyclic dependences. In this mode, the appropriate anchor is activated (line 9) with the conflicting address as the target (line 10).

Coarse-grain Mode. In this case the conflicting PC is recurrent, but the data address keeps changing. This is typical of pointer-based structures like lists (Figure 3) and trees, whose nodes are scattered across cache lines. In this mode we activate the anchor with a "wild card" data address (line 14). In the next instance of the transaction, the first accessed DSNode of the structure (usually the root or head node—i.e., the whole data structure) will be locked.

Locking Promotion. If contention persists in coarse-grain mode, the lock is promoted to the parent anchor (line 16) in the hope of avoiding contention there.

```
1  void ActivateALPoint(ABContext *c, AbortInfo *abt) {
2      AEntry *en=SearchByPC(c→anchorTable, abt→confPC);
3      if (!en→isAnchor) // always begin with an anchor
4          en = en→pioneer;
5      AbortInfo *history = c→abtHistory;
6      bool a=CountAddr(history,abt→confAddr)>ADDR_THR;
7      bool p = CountPC(history, en→PC) > PC_THR;
8      if (p && a) {                          ▶ case 1: precise mode
9          c→activeAnchor = en→ID;
10         c→blockAddress = abt→confAddr;
11     } else if (p && !a) {
12         if (retries < PROM_THR) {          ▶ case 2: coarse grain
13             c→activeID = en→ID;
14             c→blockAddr = 0;
15         } else {                           ▶ case 3: locking promotion
16             c→activeID = en→parent;
17             c→blockAddr = 0;
18         }
19     } else { // !p                         ▶ case 4: training mode
20         c→activeID = 0;
21         c→blockAddr = 0;
22     }
23     AppendToHistory(history, en→PC, abt→confAddr);
24 }
```

Figure 6: Pseudocode of a simple locking policy.

Training Mode. When no pattern has (yet) emerged, the policy simply continues to gather statistics.

When a transaction commits while holding an advisory lock, but there was no contention on that lock, an empty entry can be appended to the abort history to shift out the previous records, avoiding over-locking in the case of low contention.

Coarse-grain locking and locking promotion serve to break cycles of conflict among transactions that occur on separate locations. In pointer-based data structures in particular, conflicting addresses may vary across both time and threads. Location-based hardware techniques for conflict avoidance (e.g., Wait-n-GoTM [14]) are generally unable to avoid such conflicts. With the advantage of information from the compiler's Data Structure Analysis, Staggered Transactions handle these conflicts by acquiring locks at a coarser granularity or a higher (more abstract) level of the data structure. Consider, for example, the transaction in Figure 3, which consumes significant time in genome. The transaction inserts segments into several lists in a shared table. A cycle of conflict may easily arise among threads—thread 1 inserts segments to lists A, B, and D; thread 2 to D and C; thread 3 to C and A. With a frequently conflicting PC (Anchor 35) and unstable conflicting addresses, the locking policy will eventually reach the advisory lock for the whole table (Anchor 42), breaking the conflict cycle.

6. EXPERIMENTAL RESULTS

Hardware. Since no existing HTM provides all the hardware features required by Staggered Transactions, we conducted our experiments on MARSSx86 [19], a full-system cycle-accurate x86 simulator with high fidelity HTM support [7]. The HTM simulation is based on a variant of AMD's Advanced Synchronization Facility (ASF) proposal [7, 10]. The ISA uses speculate/commit instructions to mark a transaction region. Read and write sets are maintained in the L1 cache by adding two bits (tx read and tx write) to each cache line. An eager requester-wins conflict resolution policy is implemented on top of a modified MOESI coherence

CPU cores	2.5GHz, 4-wide out-of-order issue/commit
L1 cache	private, 64K D + 64K I, 8-way, write-back, 64-byte line, 2-cycle
L2 cache	private, 1M, 8-way, write-back, 10-cycle
L3 cache	shared, 8M, 8-way, write-back, 30-cycle
Coherence	MOESI
Memory	4 GB, 50ns, 2 memory channels
HTM	2-bit (r/w) per L1 cache line eager requester-wins policy
Stag. Trans.	12-bit PC tag per L1 cache line

Table 2: Configuration of the HTM simulator.

protocol. This HTM, designed to "incur the fewest modifications to the existing cache coherence and core designs" [33], is similar to those employed in Intel's Haswell [33] and IBM's zEC12 [13], with the notable addition of nontransactional loads and stores within transactions. We made the following modifications to the simulator:

- The original ASF proposal treated only annotated load/store instructions (`lock mov`) as transactional operations, and normal load/store as nontransactional. In keeping with later changes in the proposal, we reversed this behavior to match that of other HTM systems.
- As described in Section 4, we added a 12-bit PC tag to every L1 cache line to record bits of the conflicting PC. The space overhead in the L1 cache is less than 2.4%.
- On a contention abort, the hardware places the low 12 bits of the conflicting PC and the low 52 bits of the conflicting data address into the `%rbx` register.

We model a 16-core machine with the configuration shown in Table 2.

Compiler and HTM Runtime. Our compiler support is realized in LLVM 3.4 as an optimization pass, using an existing DSA implementation.

The locking policy keeps 8 recent abort records in each `ABContext`, with `PC_THR=2` and `ADDR_THR=2`. The HTM runtime tries each hardware transaction up to 10 times; it then enters irrevocable mode by acquiring a global lock. Hardware transactions add the global lock to their read set immediately before attempting to commit. Prior to a retry, the runtime spins for an amount of time whose mean value is proportional to the number of retires (as in the "Polite" policy of Scherer & Scott [26]).

Benchmarks. We use the STAMP suite [18] and three other representative TM programs as benchmarks, as summarized in Table 4. STAMP's `yada` and `bayes` are excluded because `yada` has overflow issues and `bayes` has unstable execution time. The `list-hi` microbenchmark is drawn from the RSTM test suite [22]. It comprises a set of threads that search and update a single shared, sorted list. The `tsp` benchmark is our own C++ implementation of a branch-and-bound TSP solver. All candidate tasks are kept in a B+ tree-based [1] priority queue, which supports $O(1)$ `pop` and $O(n \log n)$ `push` operations. We eliminated the tree's `size` field, which tends to be highly contended. The `memcached` benchmark is a modified version of memcached 1.4.9. The network code is elided in order to speed up simulation and to increase the number of working threads. We obtain the input data from `memslap` and inject them directly into the application's command processing functions.

Program	Static Stats		Dynamic Stats (1 thread)			Accuracy (16 thds)
	ld/st instrs	anchs	u-ops per txn	anchs per txn	exec. time inc	
genome	82	19	957	17.6	<1%	100%
intruder	410	56	351	8.5	<1%	97.2%
kmeans	13	6	261	4.5	1.6%	99.1%
labyrinth	418	18	16968	89.4	<1%	100%
ssca2	33	7	86	3.1	<1%	97.9%
vacation	442	76	4621	63.9	<1%	95.3%
list-hi	43	5	391	32.9	5.1%	98.7%
tsp	737	75	2348	9.7	<1%	97.0%
memcached	405	54	2520	80.9	<1%	98.3%

Table 3: Static and dynamic statistics of instrumentation.

All binaries were compiled with `-O2` optimizations, running on Debian 7.0 with a Linux 3.2 kernel. To avoid the potential contention bottleneck in the default `glibc` memory allocator, we use the Lockless Memory Allocator [12] instead. To reduce the impact of the OS scheduler, we pin every worker thread to a specific CPU core during program initialization. Each run was repeated 5 times; the average number is reported.

6.1 Instrumentation Overhead and Accuracy

The "Static Stats" section of Table 3 shows the number of loads and stores analyzed by the compiler and the number of these that were instrumented as anchors ("anchs") at compile time. On average, 13% of loads and stores are instrumented.

The "Dynamic Stats" section reflects the behavior of instrumented code in single-threaded runs. Since the number of anchors executed in each transaction is small compared to the total number of μ-ops, and an inactive ALP is simply a test and a non-taken branch, the execution time change is negligible in most benchmarks. The principal exception

Program	Source	Description and input	ABs	%TM	S	Abts/C	Contention
genome	STAMP	-g1024 -s16 -n16384	5	61%	6.0	0.25	low
intruder		-a10 -l4 -n2038 -s1	3	98%	3.2	5.28	high
kmeans		-m15 -n15 -t0.05 -i random-n2048-d16-c16	3	42%	4.6	4.74	high
labyrinth		-i random-x16-y16-z3-n64, w/ early release	3	91%	1.9	3.47	high
ssca2		-s13 -i1.0 -u1.0 -l3 -p3	10	16%	4.8	0.02	low
vacation		-n4 -q40 -u90 -r16387 -t4096	3	87%	9.7	0.49	med
list-lo	IntSet[22]	64 nodes, 90%/5%/5% lookup/insert/delete	4	86%	3.6	1.11	med
list-hi		64 nodes, 60%/20%/20% lookup/insert/delete	4	83%	1.0	4.05	high
tsp	[1]	travel salesman problem solver, 17 cities	3	90%	3.6	1.74	med
memcached	[17]	in-memory key-value storage	17	85%	2.6	4.77	high

Table 4: Benchmark characteristics. ABs: number of atomic blocks in the source code. %TM: percentage of execution time spent in transactional mode. S: speedup with 16 threads over sequential run on the baseline HTM. Abts/C: aborts per commit on the baseline with 16 threads.

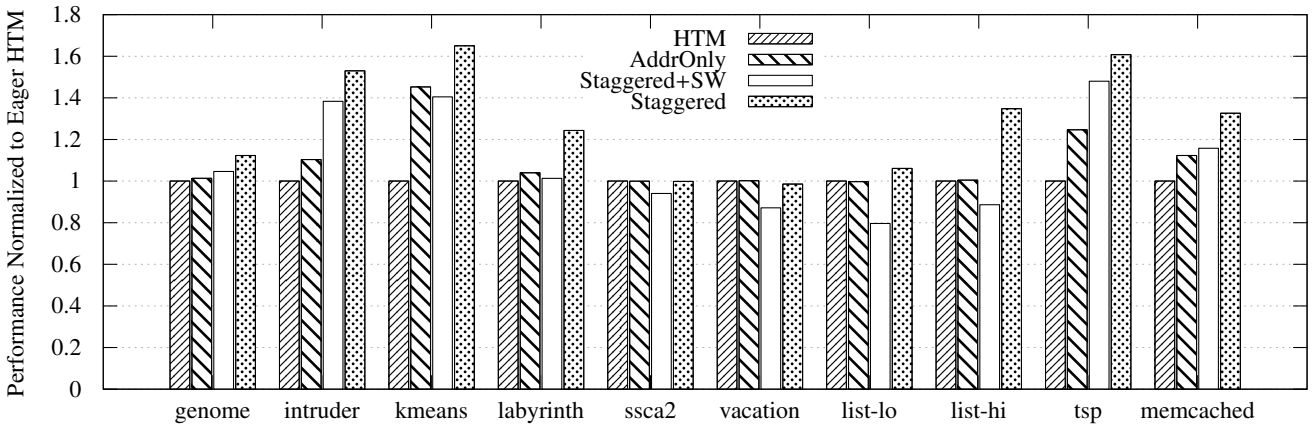

Figure 7: Performance comparison with 16 threads.

is the list microbenchmark, in which the anchors appear in tight loops. Further optimization of such loops may be a fruitful direction for future work.

For comparison, we also constructed a naive implementation in which every load and store was instrumented. This lead to slowdowns in excess of 10% for six of the benchmark programs (labyrinth, kmeans, vacation, list, tsp, and memcached).

The "Accuracy" section of the table shows the percentage of dynamic aborts for which our runtime was able to correctly identify the anchor associated with the initial access of the contended datum. All are above 95%; six out of nine are above 98%.

6.2 Parallel Performance

In the "S" column of Table 4, we list the speedup of benchmarks running on the baseline eager HTM at 16 threads. These benchmarks show low to high contention, as indicated in the final column. The worst is list-hi, which stops scaling after 4 threads.

Comparative performance on 16 threads appears in Figure 7, which plots speedup relative to the baseline ("HTM") for "StaggeredTM" (with hardware Conflict PC support) and "StaggerTM w/o CPC" (with software-based anchor tracking, as described in Section 4). Also plotted is a much simpler scheme, "AddrOnly," which places one fixed ALP at the beginning of each atomic block and uses only precise mode to trigger lock acquisition.

Result 1: *Staggered Transactions improve the performance of high-contention applications without slowing down application with low contention.*

We see substantial performance improvement (>30%) in intruder, kmeans, list-hi, tsp, and memcached. The improvement in intruder comes from serializing the modifications to a global queue, especially an enqueue that occurs near the end of a long transaction (TMdecoder_process). In kmeans, most conflicts take place when updating an array of pointers representing the centers of data clusters. Due to the good locality of conflicting PC and data addresses, Staggered Transactions are able to acquire advisory locks on a per-cluster basis (close to what fine-grain locking could achieve). In list-hi, Staggered Transactions avoid repetitive aborts among several conflicting transactions by locking the entire list (case 2 in Figure 6). Note that locking is triggered only when contention actually arises, and transactions that do not contribute to that contention are not blocked. In tsp, Staggered Transactions successfully discover that the head of the priority queue (left-most node of the tree) is the most contended object. Transactions that perform insertions on the same leaf node are also serialized if they repeatedly abort each other. Most conflicts in memcached are due to access to global shared statistics, accessed in the middle of transactions. Staggered Transactions introduce significant serialization, but still allow more concurrency than the baseline with fallback to a global lock.

Moderate performance improvements (6–24%) are obtained in genome, list-lo, and labyrinth. In genome, the most time-consuming transaction inserts a few elements into a fixed-sized hash table, which ends up overloaded and prone to contention, particularly when a conflict chain is established among several transactions. Although the conflicting PC is associated with the list-traversal code used to access buckets of the table, the Staggered Transactions policy can serialize at the level of the table as a whole via locking promotion (Section 5.2), thereby avoiding aborts even in the presence of conflict chains. Two benchmarks (ssca2 and vacation) see no significant improvement in execution time. Even in these, however, Staggered Transactions reduce the frequency of aborts, as shown in Section 6.3.

The harmonic mean of performance improvements across all benchmarks is 24%.

Result 2: *Staggered Transactions benefit from both partial overlap and a flexible blocking policy.*

In intruder, tsp, and memcached, conflicting data addresses are stable across transaction instances. Here the performance improvements stem simply from serializing the conflicting portions of those transactions, and allowing the remainder to execute in parallel. In the other benchmarks, conflicting data addresses vary greatly; for these, coarse-grain locking and locking promotion are essential to conflict reduction.

6.3 Reduced Aborts and Wasted Cycles

Result 3: *Staggered Transactions reduce contention and wasted CPU cycles for most applications.*

For each of our benchmark applications, we compare Staggered Transactions to the baseline HTM with regard to (1) the ratio of aborted to committed transactions and (2) the ratio of wasted to committed *work* (cycles) (Figure 8). Stag-

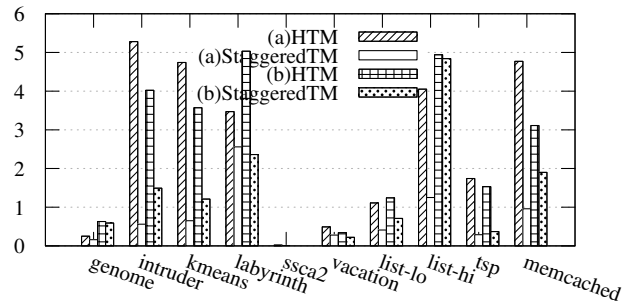

Figure 8: (a) aborts per commit and (b) ratio of wasted CPU cycles over useful cycles with 16 threads.

gered Transactions eliminate up to 89% of the aborts (in intruder) and an average of 64% across the benchmark set (excluding ssca2, which has too few aborts for the numbers to be meaningful). This results in an average savings of 43% of the wasted CPU cycles (a lower number than the savings in abort rate, because aborted transactions typically stop when only part-way through). Assuming that cycles that would have been wasted on aborted transactions are instead devoted either to useful work or to waiting (at relatively low power consumption) for advisory locks, it seems reasonable to expect Staggered Transactions to achieve a significant reduction in energy as well.

7. RELATED WORK

HTM systems can be broadly categorized as *eager* or *lazy*, depending on whether conflicts are discovered "as they occur" or only at commit time. A few systems, such as FlexTM [28] and EazyHTM [30], support mixed policies. Most current commercial systems employ a "requester wins" policy in order to avoid changes to the coherence protocol (IBM's z series machines leverage existing NAK messages to delay aborts, in an attempt to give the victim a chance to commit [13]). A few designs are more sophisticated: LogTM-SE [32], for example, will stall some transactions on conflict; potential deadlock is detected using timestamps in coherence messages. Even a simple conflict manager, however, introduces significant implementation challenges because of the necessary protocol extensions and validation cost [27].

Nonetheless, several ambitious solutions have been proposed for eager HTM. In the dependence-aware transactions (DATM) of Ramadan et al. [21], speculative data may be forwarded from transaction A to transaction B if prior accesses have already dictated that A must commit before B, and B attempts to read something A has written. In the Wait-n-GoTM of Jafri et al. [14], hardware may generate an exception that prompts the runtime to delay a transaction, if prior experience indicates that upcoming instructions are likely to introduce a circular dependence with some other active transaction. Most recently, Qian et al. proposed Omni-Order [20] to support cycle detection and conflict serialization in a directory-based coherence protocol. RETCON [3], which targets lazy HTMs, tries to "rescue" conflicting transactions by re-executing the conflicting code slice at commit time.

While these hardware proposals may achieve significant reductions in conflict rate, most are specific to a particular class of HTM (e.g., eager or lazy), or are applicable only to certain conflict patterns. FlexTM and DATM, for example, require changes to the cache coherence protocol. RETCON can resolve only simple conflicts such as counter increment. Wait-n-GoTM requires the underlying TM to be version-based, and the centralized predictor tends to be a bottleneck.

Staggered Transactions share the "stall before encountering contention" philosophy of systems like LogTM-SE and Wait-n-GoTM. Because Staggered Transactions are implemented principally in software, however, they are not bound to any particular style of HTM or conflict resolution strategy. The required support, we believe, could be added easily to existing hardware. More significantly, Staggered Transactions' use of high-level program knowledge allows them to resolve contention patterns that are unlikely to be captured by a pure hardware solution (e.g., conflicts in a data structure with no stable set of conflicting data addresses).

Contention management has also been a subject of active research in STM systems, where the flexibility of software and the high baseline overhead of instrumentation can justify even very complex policies. While much early work (including our own [26, 29]) served mainly to recover from contention once it happened, several projects have aimed to avoid contention proactively. Multi-version STM, pioneered by Riegel et al. [23], significantly reduces contention by allowing a transaction to "commit in the past" if it has not written any location that was read by a subsequent transaction. Later work by the same authors [24] uses Data Structure Analysis [16] to partition shared data and choose a potentially different STM algorithm or locking policy for each partition. Chakrabarti et al. [5] use a profiling-based *abort information graph* to identify data dependences and optimize STM policy. Given their reliance on software instrumentation, none of these techniques are compatible with existing HTM, and all would be difficult to integrate into future hardware designs.

In a manner less dependent on TM system details, contention can sometimes be avoided by carefully scheduling the threads that run conflicting transactions. Proactive Transaction Scheduling [2] learns from repeated aborts and predicts future contention. The scheduler uses the prediction to serialize entire transactions when they are likely to conflict with one another. In comparison to such techniques, Staggered Transactions avoid the overhead of scheduling decisions, thereby avoiding any negative impact on the performance of short transactions. Also, by serializing only the conflicting portions of transactions, Staggered Transactions can achieve more parallelism.

8. CONCLUSIONS

We have presented an automatic mechanism, Staggered Transactions, to serialize the conflicting portions of hardware transactions, thereby reducing aborts. Our technique employs compile-time Data Structure Analysis to understand program data, allowing us to accommodate a wide variety of conflict patterns. While the choice of which lock to acquire is always based on the address of the data being accessed, the decision as to *whether* to acquire a lock, and at what instruction address, is made adaptively at run time.

From the hardware, Staggered Transactions require the ability to acquire an advisory lock from within an active transaction; they also benefit from a mechanism to recall the program counter of the initial speculative access to a given conflicting location. Experiments on the MARSSx86 ASF simulator demonstrate speedups averaging 24% on a

collection of 9 TM applications. In future work we hope to experiment with a wider range of run-time policies and with compatible physical hardware (e.g., the POWER 8). We also plan to extend our simulations to lazy TM protocols.

References

[1] T. Bingmann. *STX B+ Tree C++ Template Classes*. URL `http://panthema.net/2007/stx-btree`.

[2] G. Blake, R. G. Dreslinski, and T. Mudge. Proactive transaction scheduling for contention management. In *42nd IEEE/ACM Intl. Symp. on Microarchitecture*, MICRO 42, New York, NY, 2009.

[3] C. Blundell, A. Raghavan, and M. M. Martin. RETCON: Transactional repair without replay. In *37th Intl. Symp. on Computer Architecture*, ISCA '10, Saint-Malo, France, 2010.

[4] 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*, ISCA '13, Tel-Aviv, Israel, 2013.

[5] D. R. Chakrabarti, P. Banerjee, H.-J. Boehm, P. G. Joisha, and R. S. Schreiber. The runtime abort graph and its application to software transactional memory optimization. In *9th IEEE/ACM Intl. Symp. on Code Generation and Optimization*, CGO '11, 2011.

[6] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, and S. Yip. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, Mar.–Apr. 2009.

[7] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's Advanced Synchronization Facility within a complete transactional memory stack. In *5th European Conf. on Computer Systems*, EuroSys '10, 2010.

[8] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman. ASF: AMD64 extension for lock-free data structures and transactional memory. In *2010 43rd IEEE/ACM Intl. Symp. on Microarchitecture*, MICRO 43, 2010.

[9] 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*, ASPLOS '11, Newport Beach, CA, 2011.

[10] S. Diestelhorst. *Marss86-ASF*, 2013. URL `http://bitbucket.org/stephand/marss86-asf`.

[11] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: Parallelizing a complex sequential application using transactional memory. In *23rd Intl. Conf. on Supercomputing*, ICS '09, 2009.

[12] L. Inc. *The Lockless Memory Allocator*. URL `http://locklessinc.com`.

[13] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System z. In *2012 45th IEEE/ACM Intl. Symp. on Microarchitecture*, MICRO '12, Vancouver, B.C., Canada, 2012.

[14] S. A. R. Jafri, G. Voskuilen, and T. N. Vijaykumar. Wait-n-GoTM: Improving HTM performance by serializing cyclic dependencies. In *18th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, Houston, Texas, 2013.

[15] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, May 2005.

[16] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '05, Chicago, IL, 2005.

[17] Memcached. URL `http://memcached.org`.

[18] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE Intl. Symp. on Workload Characterization, 2008.*, IISWC '08, 2008.

[19] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conf. 2011 (DAC'11)*, 2011.

[20] X. Qian, B. Sahelices, and J. Torrellas. Omniorder: Directory-based conflict serialization of transactions. In *41st Intl. Symp. on Computer Architecture*, ISCA '14, June 2014.

[21] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *41st IEEE/ACM Intl. Symp. on Microarchitecture*, MICRO 41, Como, Italy, 2008.

[22] Reconfigurable Software Transactional Memory Runtime. URL `http://code.google.com/p/rstm`.

[23] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *20th Intl. Symp. on Distributed Computing*, DISC '06, Stockholm, Sweden, Sept. 2006.

[24] T. Riegel, C. Fetzer, and P. Felber. Automatic data partitioning in software transactional memories. In *20th Symp. on Parallelism in Algorithms and Architectures*, SPAA '08, Munich, Germany, 2008.

[25] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *23rd ACM Symp. on Parallelism in Algorithms and Architectures*, SPAA '11, 2011.

[26] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *24th ACM Symp. on Principles of Distributed Computing*, PODC '05, Las Vegas, NV, 2005.

[27] A. Shriraman and S. Dwarkadas. Refereeing conflicts in hardware transactional memory. In *23rd Intl. Conf. on Supercomputing*, ICS '09, Yorktown Heights, NY, 2009.

[28] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *35th Intl. Symp. on Computer Architecture*, ISCA '08, Beijing, China, 2008.

[29] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '09, Raleigh, NC, 2009.

[30] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: Eager-lazy hardware transactional memory. In *42nd IEEE/ACM Intl. Symp. on Microarchitecture*, MICRO 42, New York, NY, 2009.

[31] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, San Francisco, CA, Feb. 2015.

[32] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *2007 IEEE 13th Intl. Symp. on High Performance Computer Architecture*, HPCA '07, Phoenix, AZ, 2007.

[33] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '13, Denver, CO, 2013.