Techniques to Improve Performance in Requester-Wins Hardware Transactional Memory

ADRIÀ ARMEJACH, Barcelona Supercomputing Center, Universitat Politècnica de Catalunya
RUBEN TITOS-GIL and ANURAG NEGI, Chalmers University of Technology
OSMAN S. UNSAL, Barcelona Supercomputing Center
ADRIÁN CRISTAL, Barcelona Supercomputing Center, Universitat Politècnica de Catalunya, IIIA - Artificial Intelligence Research Institute (CSIC)

The simplicity of requester-wins Hardware Transactional Memory (HTM) makes it easy to incorporate in existing chip multiprocessors. Hence, such systems are expected to be widely available in the near future. Unfortunately, these implementations are prone to suffer severe performance degradation due to transient and persistent livelock conditions. This article shows that existing techniques are unable to mitigate this degradation effectively. It then proposes and evaluates four novel techniques—two software-based that employ information provided by the hardware and two that require simple core-local hardware additions—which have the potential to boost the performance of requester-wins HTM designs substantially.

Categories and Subject Descriptors: C.1.4 [**Processor Architectures**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Contention management, hardware transactional memory, requester-wins conflict resolution

ACM Reference Format:
Armejach, A., Titos-Gil, R., Negi, A., Unsal, O. S., and Cristal, A. 2013. Techniques to improve performance in requester-wins hardware transactional memory. ACM Trans. Architec. Code Optim. 10, 4, Article 42 (December 2013), 25 pages.
DOI: http://dx.doi.org/10.1145/2555289.2555299

1. INTRODUCTION

There is an exigent need for high-productivity approaches that allow control of concurrent accesses to data in shared memory multithreaded applications without severe performance penalties. This has led researchers to look seriously at the concept of Transactional Memory (TM) [Herlihy and Moss 1993; Harris et al. 2010]. TM allows the programmer to demarcate sections of code—called *transactions*—which must be executed atomically and in isolation from other concurrent threads in the system.

This work has been supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2008-02055-E, by the EUROTM Cost Action IC1001, by the Swedish Foundation for Strategic Research under grant RIT10-0033 as well as the HiPEAC Network of Excellence under contract EU FP7/ICT 287759.
Author's addresses: A. Armejach (corresponding author), O. S. Unsal, and A. Cristal, Barcelona Supercomputing Center, Nexus I Building Office 303, Gran Capita 2-4, 08034 Barcelona, Spain; R. Titos-Gil and A. Negi, Chalmers University of Technology, SE-41296 Gothenburg, Sweden; email: adria.armejach@bsc.es.

DOI: http://dx.doi.org/10.1145/2555289.2555299

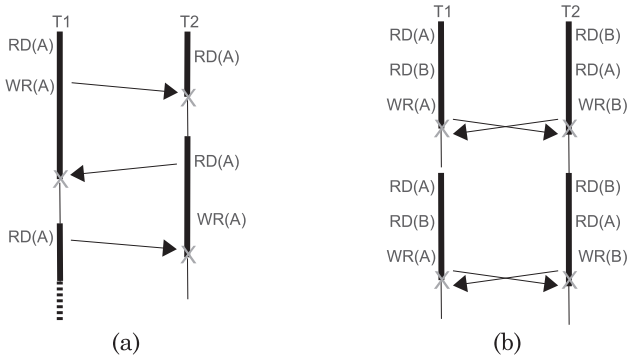

Fig. 1. Two livelock scenarios in requester-wins HTM.

The TM system detects and resolves conflicts, i.e., circumstances when two or more transactions access the same shared data and at least one modifies it.

Although TM has been an active research topic for almost a decade [Hammond et al. 2004; Ananian et al. 2005; Yen et al. 2007; Shriraman et al. 2008; Blundell et al. 2010; Lupon et al. 2010; Jafri et al. 2013], bare-bones support for Hardware Transactional Memory (HTM) is only just appearing. Large-scale adoption of software-only approaches has been hindered for long by severe performance penalties arising out of the need for extensive instrumentation and book-keeping to track transactional accesses and detect conflicts without hardware support. Intel TSX extensions and IBM BlueGene-Q are now testing the waters with hardware TM [Intel Corporation 2012; Wang et al. 2012]. Restricted Transactional Memory (RTM) as described in Intel TSX specifications appears to be a requester-wins HTM where transactions abort if a conflicting remote access is seen while executing a transaction. Transactions may also abort when hardware resource limitations (e.g., cache capacity) or exceptional hardware events (interrupts) are encountered. In this study, we are primarily concerned with the nature of the requester-wins conflict resolution policy and not with conditions arising out of lack of hardware resources or exceptions. The authors do not have access to implementation details of Intel RTM and, thus, results in this article must be seen in the more general context of requester-wins HTM designs.

Requester-wins HTMs are easy to incorporate in existing chip multiprocessors [Chung et al. 2010; Christie et al. 2010]. Conflict detection and resolution mechanisms in such systems do not require any global communication except that which naturally arises from the need to impose cache coherence. Each core tracks accesses made by transactions that run locally. This could be done using cache line annotations indicating lines that have been read or written. Some implementations may choose to employ read/write set bloom filters for the purpose. Either way, the requester-wins policy has no inherent forward progress guarantees since a local transaction aborts whenever it receives a conflicting coherence request for a line in its read or write sets. This susceptibility to livelock is well-known [Bobba et al. 2007]. However, the likelihood of livelocks in such systems and their eventual impact on performance has not been investigated in depth. Livelocks may persist for a while but eventually get broken due to varying delays in real-world systems. When this occurs, they may manifest themselves as degradation in application execution times or system throughput.

Figure 1(a) shows how two transactions may livelock. Both transactions read data that is eventually written by the other. Executions of the two transactions may interleave such that no progress is made at either thread. However, cyclic dependencies between concurrent transactions are not the only sources of livelock. A potentially more

pathological livelock behaviour exists (Figure 1(b)) where multiple read-modify-write transactions may continually abort each other (i.e., friendlyfire [Bobba et al. 2007]). The livelock occurs because the aborted transaction issues a conflicting access upon re-execution, which then aborts the transaction that was allowed to proceed.

The aim of this article is to show that protocols that merely guarantee livelock freedom may not be the most efficient. The performance impact of livelock mitigation and avoidance techniques should be looked at in more depth. HTM systems should incorporate a set of such techniques in a manner that allows resolution of these livelock conditions as soon as possible and with the lowest associated performance cost. This article investigates performance implications of a number of existing strategies like exponential backoff [Moore et al. 2006], serial irrevocability as implemented in GCC `libitm` since version 4.7.0, and hourglass [Liu and Spear 2011]. Our study shows that there is a substantial cost in terms of performance imposed by these strategies. With an aim to minimize this cost, this article proposes some novel techniques, in hardware and software, which are well suited to requester-wins HTM designs. Four new techniques for mitigation of livelocks are presented—two are implemented in software, requiring only simple interfaces for reading information provided by the hardware, and two are implemented in hardware with simple core-local additions.

Our analysis of relative merits of these proposed techniques shows that deficiencies of requester-wins HTMs can be ameliorated effectively for a variety of transactional workloads. One of our aims is to make system programmers using HTM aware of the severity of livelocks and the performance cost imposed by various mitigation and avoidance techniques. This would help them decide what mitigation techniques to choose. This article also aims to convey to processor architects the importance of simple hardware mechanisms to mitigate the impact of livelocks. In summary, it sheds light on the following concerns:

—How severe can livelocks be in requester-wins HTMs?
—What are the performance costs associated with existing livelock mitigation techniques?
—Can new techniques (hardware-only or hybrid) be designed to reduce performance degradation while retaining the simplicity of requester-wins HTM?

The rest of this article is organized as follows: Section 2 shows that livelocks frequently block forward-progress in several transactional workloads running on requester-wins HTMs. It also shows that existing livelock mitigation and avoidance strategies (backoff, serial irrevocability, and hourglass) leave a large performance gap between observed performance and performance achievable by a livelock-free HTM. In Section 3, we describe in detail four new techniques to improve performance by mitigating livelock conditions. Section 4 introduces our experimental methodology, and Section 5 provides experimental evidence that highlights the efficacy of our new techniques. In Section 6, we discuss about related work. Finally, in Section 7, we conclude with final thoughts and a summary of insights gathered from this work.

2. MOTIVATION

Our experiments with a variety of workloads, which include the STAMP benchmark suite [Cao Minh et al. 2008], *water* and *radiosity* from SPLASH2 [Woo et al. 1995] and two microbenchmarks (*deque* and *btree*), show that most of them consistently livelock when running on requester-wins HTM without any livelock mitigation strategy. The data in Table I lists the workloads and their susceptibility to livelock on a variety of scenarios. The results have been gathered on a simulated 8-core machine. A suffix -h after the workload name indicates high contention parameters have been used. A suffix + indicates larger datasets. These livelocks occur due to two or more concurrent threads

Table I. Livelocks in Applications

Application	Live-lock	Operation where livelock occurs	Application	Live-lock	Operation where llivelock occurs
Deque	Yes	Deque access	Water	Yes	Counter increment
Btree	Yes	Btree insertion	Radiosity	Yes	Counter increment
Genome	Yes	Hashtable insertion	Genome+	Yes	Hashtable insertion
Intruder	Yes	Task queue access	Intruder+	Yes	Task queue access
KMeans-h	Yes	Matrix access	KMeans-h+	Yes	Matrix access
Labyrinth	No	–	Labyrinth+	Yes	Vector access
SSCA2	Yes	Vector access	SSCA2+	Yes	Vector access
Vacation-h	Yes	Counter increment	Vacation-h+	Yes	Counter increment
Yada	Yes	Heap removal	Yada+	Yes	Heap removal

entering a pattern of continuous aborts, eventually preventing any forward progress as other threads either wait perpetually at a barrier or enter livelock themselves. We executed each application multiple times with randomized delays added to the main memory access latency ($\pm 5\%$), so as to create different thread interleavings.

We have also identified the kind of operation that triggers a livelock condition for each workload. The results in Table I indicate that livelocks are a serious problem for a variety of common operations in different data structures. Without appropriate mitigation strategies, in software or hardware, the use of transactions in such a system may be rendered impractical. This leads us to the next question we attempt to answer: What are the costs of various existing livelock mitigation strategies?

2.1. A Look at Existing Techniques

We now briefly describe existing software techniques for livelock mitigation and avoidance that can improve overall performance in requester-wins HTMs. We will concentrate on three techniques: *exponential backoff*, introduced as an HTM contention manager in LogTM [Moore et al. 2006]; *serial irrevocability*, previously used in software TM proposals [Welc et al. 2008; Felber et al. 2010] and now also used in GCC as the default fallback mechanism upon repeated aborts; and *hourglass*, which provides a more relaxed form of serialization than serial irrevocability [Liu and Spear 2011].

Serial Irrevocability. This is a fallback mode in case a hardware transaction fails to commit after retrying several times. This mode could be chosen because of contention or hardware resource limitations. The number of retries before entering this mode is usually configurable. The mode works by aborting any concurrent transactions in the system through the acquisition of a global multiple-reader-single-writer lock as a writer. This also ensures that no other transaction in the application can begin execution, allowing the irrevocable transaction to be executed without interference from other threads. The algorithm for starting and committing transactions is shown in Algorithm 1. The call to `beginTransaction()` returns success after either starting the transaction in serial irrevocable mode or in the usual hardware-supported TM mode. On a transactional abort the architectural state is restored and execution is resumed from the fallback code path. The `commitTransaction()` routine ensures that the transaction releases the serial lock if it was running irrevocably. Otherwise, it will execute the supported ISA instruction to commit a transaction. This implementation resembles the one that can be found in the new *libitm* library in GCC to provide TM support.

Randomized Exponential Backoff. Exponential backoff has been used in other domains as a collision avoidance strategy wherein backoff duration is chosen randomly from a range of durations that grows exponentially larger as the number of failures increases. Backoff has the potential to reduce chances of repetitive conflicting patterns that occur. However, it does not guarantee forward progress. Exponential backoff has been evaluated in the context of contention management options available in

ALGORITHM 1: Simplified begin and commit transaction function wrappers to implement serial irrevocability.

```
void beginTransaction()
  while true do
      TX_BEGIN(offset to fallback path);                    /* ISA begin instruction */
      if serialLockCanRead() == false then      /* adds serial lock to the read set */
          abortTransaction();        /* there is another thread in irrevocable mode */
      else
          return;                                            /* execute transaction */
      end
      fallback_path:                                       /* fallback path on abort */
      if retryCount < MAX_RETRIES then
          retryCount++;
          if serialLockCanRead() == false then
              waitForSerialLockCanRead(); /* wait for irrevocable thread to finish */
          end
                                                /* retry transactional execution */
      else
          break;                                  /* use serial irrevocable mode */
      end
  end
  acquireSerialLockWriter();                              /* aborts other transaction */
  return;                                  /* execute in serial irrevocable mode */

void commitTransaction()
  if serialLockCanRead( ) == false then
      releaseSerialLockWriter();                /* this was an irrevocable execution */
  else
      TX_COMMIT();                                          /* ISA commit instruction */
  end
  return;                                    /* successfully executed transaction */
```

software transactional memory [Scherer III and Scott 2005]. However, even though backoff strategies have also been evaluated in HTM designs [Moore et al. 2006; Bobba et al. 2007], their impact on performance in HTM systems as prone to livelock as requester-wins remains unclear.

Hourglass Contention Manager. Liu and Spear [2011] define toxic transactions as those that have aborted consecutively a number of times due to conflicts. To deal with these toxic transactions, they propose the *hourglass* contention manager, where such transactions try to grab a global token, preventing new or aborted transactions from starting. This gives the toxic transaction a better chance of committing after acquiring the token, although it is not guaranteed to commit. This mechanism is less drastic than serial irrevocability, as it allows transactions that are already running in the system to proceed when the token is acquired.

2.2. Brief Analysis of Existing Techniques

We now show how these existing techniques perform on a requester-wins HTM with respect to a well-known reference like LogTM [Moore et al. 2006]. This will allow us to estimate the gap in performance between such a requester-wins system and a proposal that implements a more complex strategy. LogTM is a requester-stalls design that uses a scheme for conservative deadlock avoidance. It introduces a timestamp in all coherence messages (thereby prioritizing older transactions in the system) and extends

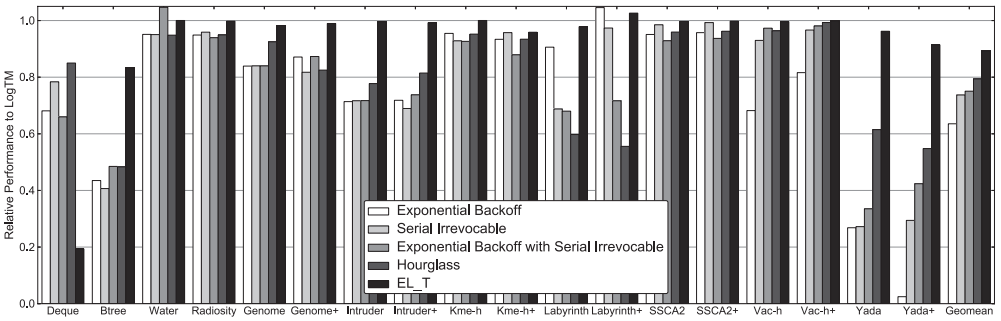

Fig. 2. Relative performance of existing livelock mitigation techniques relative to LogTM.

the coherence protocol with support for *nacks* (negative acknowledgements) that allow transactions to be stalled upon conflict instead of aborting them. The conservative deadlock avoidance scheme works by keeping a *possible cycle* bit in each core, set when a request with an older timestamp is *nacked*. Once this bit is set, the transaction must abort as soon as it receives a *nack* response with an older timestamp.

We have also included a second HTM design point, a lazy-versioning eager-resolution HTM based on the EL_T design described in Bobba et al. [2007]. It uses the L1 caches to buffer speculative updates and resolves conflicts eagerly using timestamp priorities attached to coherence messages—aborting if the remote transaction has higher priority or responding with a *nack* (negative acknowledgment) otherwise. Note that like LogTM, EL_T also requires protocol support for *nacks* and a mechanism for assigning priorities to transaction. Thus, these systems turn out to be more complex than requester-wins designs, where the coherence protocol is not modified.

Figure 2 shows relative performance normalized to LogTM (higher is better), for *exponential backoff*, *serial irrevocability*, a scenario where these two techniques are combined, and *hourglass*. The relative performance of the *EL_T* design is also shown. For reasons mentioned earlier, the LogTM design is more complex than simpler requester-wins HTMs that will soon be widely available and thus should be considered as an upper bound on performance achievable by requester-wins designs. As seen in Figure 2, the EL_T HTM design, which has been included here primarily to add another relevant HTM design point for comparison, performs well under most workloads due to its ability to prioritize transactions, but presents significant degradation in performance under high contention. Overall, it turns out to be around 12% worse than LogTM. Among software techniques, exponential backoff performs 40% worse than the baseline, being inefficient even under mild contention. Serial irrevocability is a good choice for uncontended applications like *SSCA2*. However, when contention is present, its performance drops significantly. Overall, we see that the performance offered by these two techniques and their combination is, on average, 27% worse than LogTM for this set of applications. On the other hand, the hourglass contention manager fares much better, particularly when contention is present by reducing serialization overheads. Overall, it achieves 20% less performance than LogTM. In general, we observe that under high contention there is a marked susceptibility to a much greater degradation in performance. In our opinion, this observed performance gap is large enough to merit a search for solutions to close it. Our solutions presented in the following section, therefore, attempt to do so while retaining the simplicity of requester-wins HTMs.

3. PROPOSED TECHNIQUES TO IMPROVE REQUESTER-WINS HTM DESIGNS

In this section, we describe four novel techniques to improve performance in requester-wins designs by mitigating pathological scenarios like the ones shown in Figure 1.

These techniques attempt to bridge the gap in performance highlighted in the previous section. We first introduce two software-based techniques that are simple to implement in upcoming HTM designs. Later, we look at two additional techniques that require simple core-local hardware support, but retain the requester-wins nature of the HTM.

3.1. Serialize on Conflicting Address (SoA)

It has been shown in prior work [Titos-Gil et al. 2011; Negi et al. 2012a] that conflicts are usually caused by a small number of contended addresses with large fractions of data accessed by typical transactions seeing no contention at all. Thus, a large number of transactions in code are prone to see conflicts only on a few addresses. Moreover, it is not very complicated in hardware to determine this address when a conflict occurs, since, in requester-wins, HTM designs cores abort when they receive coherence requests that carry the address. This information could be passed on to the runtime through interfaces similar to the ones already implemented in production devices. For example, Intel TSX supplies information about the nature of aborts through the *EAX* register, among other things.

Our approach utilizes the additional bits from the *RAX* register to feed the address of the conflicting cache line onto the runtime. Using this additional information, the runtime is able to identify potential hotspots of contended cache lines and rely on locks to execute one transaction after another, with relatively few transactions requiring a fallback to the more drastic form of serialization enforced through serial irrevocability. Algorithm 2 shows the necessary steps to implement this proposal. Note that for the sake of clarity, in this algorithm, we do not include the necessary checks to have serial irrevocability (described in Section 2.1).

The approach works by trying to acquire a lock from an array of locks using a hashed version of the conflicting address as index. If another thread has already acquired the lock for that address, the current thread waits. This approach allows threads that are likely not to contend with each other to proceed, whereas threads that conflict on the same addresses serialize. We only allow each thread to acquire a single lock to avoid cyclic dependencies. Therefore, the number of locks concurrently in use is small, lower, or equal than the number of executing threads. This approach is able to deal quite effectively with livelock scenarios produced by common read-modify-write transactions, similar to the one shown in Figure 1(b). However, the scenario in Figure 1(a) would still require serial irrevocability to ensure forward progress.

3.2. Serialize on Killer Transaction (SoK)

Our second proposal is a software technique that stalls restarted transactions until the offending transaction (i.e., the transaction whose request caused the abort) completes. As in the previous solution, this is a hardware-assisted software mechanism that requires the identity of the conflicting thread (a.k.a. *killer*) to be passed from the hardware to the runtime at the time of an abort. This scheme is of special interest in requester-wins systems because restarted transactions are likely to abort their killers when restarting.

Algorithm 3 shows how the idea is implemented. Before a transaction begins, its execution, it reads a multiple-reader-single-writer lock from a vector of locks indexed by the thread identifier. This read operation stalls writers if they try to write the lock. When a transaction aborts, before it is allowed to restart, it checks whether it has permissions to write to the killers lock. Note that the killer only releases write permissions on the lock after it has committed the transaction. If it does not have permissions to write to the lock, then the killer is still executing the transaction and the aborted transaction must wait. Cyclic dependencies may arise causing deadlock. The approach avoids this by ensuring that the wait is deadlock free through a check for potential cycles using a vector

ALGORITHM 2: Simplified begin and commit transaction function wrappers to implement serialize on conflicting address (SoA).

```
void beginTransaction()
  while true do
    TX_BEGIN(offset to fallback path);                    /* ISA begin instruction */
    return;                                                 /* execute transaction */
    fallback_path:                                        /* fallback path on abort */
    if thread− >has_lock is valid then                   /* already holding one lock */
      releaseAddressLockWrite(thread− >has_lock);  /* avoid cyclic dependencies */
      thread− >has_lock = invalid
    end
    address = getConflictingAddress();    /* hardware provides conflicting address */
    index = hash(address)
    if address is invalid then
      continue;                    /* abort not related to a data conflict, retry */
    end
    acquireAddressLockWrite(index); /* try to grab lock associated with address */
    thread− >has_lock = index
                                                                        /* retry */
  end

void commitTransaction()
  TX_COMMIT();                                            /* ISA commit instruction */
  if thread− >has_lock is valid then                  /* executed with an acquired lock */
    releaseAddressLockWrite(thread− >has_lock);   /* release, others can proceed */
    thread− >has_lock = invalid
  end
  return;                                        /* successfully executed transaction */
```

(*killers_vector*) that maintains dependencies. Accesses to this vector are protected by a global lock. This guarantees that only one among a group of conflicting transactions is allowed to proceed. Since the lock on this structure serializes accesses to it, when cyclic dependencies exist the design resolves it by allowing the last transaction in a cyclic dependency chain to detect the condition and avoid a potential deadlock by not waiting on its killer. Other transactions in the now cycle-free dependency chain wait. This solution has the advantage of guaranteeing forward progress as long as transactions can execute in hardware, avoiding the use of the serial irrevocable mode in livelock scenarios.

Note that a potential corner case may arise in which a transaction is waiting for a transaction that is not its actual killer, for example, a transaction (*Tx-a*) aborts and before checking whether it has to wait, the killer transaction finishes and a new transaction (*Tx-b*) starts execution. This is likely to be an uncommon scenario, and it does not pose any deadlock or starvation problems. Deadlocks cannot occur because aborted transactions wait on their killers thread identifier, so when the new *Tx-b* finishes, the aborted *Tx-a* will restart. Starvation problems have not been encountered, but could be easily solved by adding fairness to the lock implementation.

3.3. Delayed Requester-Wins (DRW)

Our first hardware-based design makes conflicting requests wait for a bounded length of time before applying the requester-wins policy. This technique can be implemented locally at the core without changing communication protocols or messaging. Basically,

ALGORITHM 3: Simplified begin and commit transaction function wrappers to implement serialize on killer transaction (SoK).

```
void beginTransaction()
  acquireLockArrayRead(my_id); /* acquire local lock for reading, blocks writers */
  while true do
    TX_BEGIN(offset to fallback path);                    /* ISA begin instruction */
    return;                                               /* execute transaction */
    fallback_path:                                       /* fallback path on abort */
    killer_id = getKillerID();          /* hardware provides conflicting thread id */
    if killer_id is invalid then
      continue;                  /* abort not related to a data conflict, retry */
    end
    clearedForDeadlock = false;      /* indicates if has been cleared for deadlock */
    while !lockArrayCanWrite(killer_id) do     /* wait until killer thread is done */
      if !clearedForDeadlock then                /* ensure we will not deadlock */
        acquireGlobalLock();       /* check a vector of adjacencies atomically */
        if isCyclePossible(killer_id, my_id) then /* detects cycles, defined below */
          releaseGlobalLock();                  /* cannot wait, would deadlock */
          break;                                                    /* retry */
        else
          killers_vector[my_id] = killer_id;          /* will wait, update vector */
          clearedForDeadlock = true;    /* do not do the deadlock check again */
        end
        releaseGlobalLock();                              /* deadlock check done */
      end
    end
    acquireGlobalLock();
    killers_vector[my_id] = −1;           /* my killer has finished, update vector */
    releaseGlobalLock();
                                                                  /* retry */
  end

void commitTransaction()
  TX_COMMIT();                                          /* ISA commit instruction */
  releaseLockArrayRead(my_id);                /* release racers waiting on the lock */
  return;                                      /* successfully executed transaction */

bool isCyclePossible(int killer_id, int my_id)
  if killers_vector[killer_id] == −1 then return false;   /* killer not waiting, no cycle */
  if killers_vector[killer_id] == my_id then return true; /* killer waiting for me, cycle */
  return isCyclePossible(killers_vector[killer_id], my_id);             /* recursive call */
```

it attempts to capture the benefits of the requester-stall policy (i.e., resolving conflicts through stalls rather than aborts) while avoiding the complexity introduced by negative acknowledgements (*nacks*) in the coherence protocol. To this end, LogTMs protocol introduces *nacks* as well as a special kind of *unblock* message to inform the directory that a coherence transaction has failed due to conflicts and should be *cancelled*, that is, the coherence state reverted to its original state with no updates to the bit-vector of sharers. As opposed to LogTM, coherence requests in our Delayed Requester-Wins (DRW) design always complete successfully—perhaps with some additional latency—and thus there is no need to extend the protocol with new messages. Delaying coherence

messages has been explored in the past in the context of memory consistency for scalable shared-memory multiprocessors [Gharachorloo et al. 1990].

DRW allows the exclusive owner of a cache line to buffer conflicting requests and thus delay responses until a later point in time. On the requesters side, the cache miss that resulted in a conflict simply appears to be a longer latency miss, and the execution naturally stalls at this point until the memory reference completes. Delayed conflicting requests queued at the exclusive owners cache are considered either when the transaction ends (commits or aborts) or when an associated timeout expires. DRW uses timeouts to conservatively break temporary deadlocks situations that may appear when transactions exhibit circular dependencies. Timeouts are a simple solution to break cycles, and they can be implemented locally at the core level. On the other hand, LogTMs deadlock avoidance mechanism requires the addition a global timestamp (which all threads agree upon) to every coherence request and response, increasing the size of every network message that traverses the communication fabric.

Transactions with buffered conflicting requests are allowed to execute as long as they are able to make forward progress. When a transaction with buffered requests experiences an L1 cache miss, the timer is started. If the cache miss completes within the *timeout latency*, the timer is stopped because the transaction has made forward progress while buffering remote requests, which means that no cycle has been formed yet. The timer is thus reset to its initial value and will be started again in subsequent misses. Otherwise, if the timer expires while a local miss is still pending, the buffered conflicting requests begin to be serviced normally in a requester-wins fashion, triggering an abort and thus breaking any temporary cycle. If the transaction eventually commits, all conflicts are successfully resolved and the requests are serviced with the new committed data.

An important aspect in DRW is the *timeout latency*, that is, the value at which the timer is started on a cache miss. Ideally, the timer should not expire unless a cyclic dependency (transient deadlock) has occurred, and similarly it should expire as soon as the cycle has been formed. In order to set the timer accurately, DRW keeps a table that associates a different timeout latency to each atomic block of code (indexed by the PC of the *begin-tx* instruction). The value used for each atomic block is adaptable, and it moves between a range of values, in our experiments, from 64 to 1,024 cycles. Commits that successfully resolve conflicting requests by delaying the response do not update the value in the latency table. On commits without conflicts, the latency is halved in order to keep the reaction time to potential deadlocks short when contention is low. Upon timeout expiration (i.e., on abort), the latency is doubled. In this way, if conflicts are encountered again after the transaction restarts, a larger window of time is given to remote transactions so that they have a better opportunity to reach commit (i.e., service the buffered conflicting requests) before the local offending transaction aborts due to the timeout.

3.4. WriteBurst: Buffering of Store Misses (WB)

Buffering transactional stores has been shown to be beneficial in both eager and lazy systems [Negi et al. 2011; Titos-Gil et al. 2012]. In the case of a requester-wins HTM, the ability to delay completion of possibly conflicting transactional stores until close to commit time and then releasing them into the coherent cache hierarchy in a burst can improve parallelism by reducing the window of time in which a transactional write to a line may see a conflict. Remote readers can now access lines in a nonconflicting manner and writers that are close to commit have a better chance of acquiring ownership over the write-set before being aborted by a remote reader. If resources are sufficient to buffer all store misses until commit, this technique allows for a form of lazy conflict detection (*committer-wins*) [Hammond et al. 2004], which provides stronger forward progress guarantees and can enhance concurrency by allowing readers to commit

before a conflicting writer [Shriraman and Dwarkadas 2009; Titos et al. 2009; Tomić et al. 2009]. However, unlike the latter lazy systems, in our scheme there is no notion of *committer*, because it is always possible for a transaction to abort after it has reached the commit instruction while it is draining its buffered store misses; but since transactions generally write only a few cache lines, the time required to drain the buffers is generally short and the requirements to buffer all store misses are not excessive.

Our model implements the idea by utilizing the L1 Miss Status Holding Registers (MSHRs) to buffer store misses. Stores to exclusively owned lines are store hits and thus can complete as usual in cache. Our scheme is applied upon stores to shared lines—*upgrade* misses—which result in a message sent to the directory requesting the invalidation of all other privately cached copies. Lines that are absent in the L1 are prefetched nonexclusively if targeted by a speculative store (the L1 cache uses a write-allocate policy). Once the line is allocated in L1, the store is buffered in the MSHR and henceforth treated as an upgrade miss.

Our design leverages the L1 cache entry itself to keep the speculative updates, and the request is buffered in the MSHR. Since the data is present in the L1 cache in shared (S) state, only a minor behavioral change in the cache controller is needed to allow speculative stores that target S state lines to update the cache entry before write permissions are actually acquired. Per-byte dirty bits in cache to track dirty words are not needed since no merging with other versions occurs. An MSHR is allocated for the upgrade miss and the SM bit is set for the entry, but the request for ownership is not immediately sent to the L2. The issue of these upgrade messages to the L2 directory is deferred until (a) the transaction reaches the commit instruction, or (b) all MSHRs are in use. The MSHR keeps track of such entries by maintaining a special *Buffered* bit. Subsequent local loads to lines with buffered MSHR entries simply obtain the data from the cache and add the line to the read set (e.g., set the *speculatively read* bit), as usual. Remote load misses get the non-speculative version from the L2 cache, since the directory remains unaware of the speculative writes at the private cache. If an invalidation is received for a line with a buffered MSHR, then the transaction is aborted and all buffered MSHRs are discarded.

For applications with large write sets, the number of MSHRs is likely to be insufficient to buffer all store misses. When a new store miss finds all MSHR entries occupied, the design triggers a draining process which sequentially issues buffered upgrade request for all entries. To prevent drained speculative writes that have completed in cache to repeatedly expose the transaction against conflicts with restarted readers, our design incorporates a simple Bloom filter [Bloom 1970] called *conflict set signature*. This filter is used to conservatively encode write-set addresses that have seen conflicts with remote transactions. Note that only store hits or drained misses from the MSHRs are added to the write set of the transaction (i.e., set the SM bit in cache). Every time a transaction aborts due to a conflict on a write-set address, the address is added to the conflict set signature. Subsequent restarts of the transaction will most likely fill up all MSHRs again, though in this case the conflict set signature will predict those MSHRs entries whose draining should be avoided for as long as possible. In this way, when MSHRs are insufficient, store misses to thread-local and noncontended data (*contamination* misses [Waliullah and Stenstrom 2012]) are drained first, thus minimizing the aforementioned risk of cross-fire between concurrent writers and readers. The conflict set signature is always cleared on commit, and thus it only records information about previous restarts of the same dynamic transaction instance.

3.5. Overview of Existing and Proposed Techniques

Table II shows all of the described techniques with a summary of their properties and required hardware changes to implement them. Exponential backoff, serial

Table II. Overview of Techniques and Their Characteristics

	Proposal Name	Abbr.	Livelock-free?	Requires hardware?	Hardware Description
Existing	Exponential Backoff	B	No*	No	—
	Serial Irrevocability	S	Yes	No	—
	Hourglass	H	Yes	No	—
	EL_T	T	Yes	Yes	timestamps and nacks in coherence
	LogTM	L	Yes	Yes	timestamps and nacks in coherence, possible cycle detection, priority for older writers
Proposed SW	Serialize on Address	SoA	No*	Minor	provide conflicting address to runtime
	Serialize on Killer	SoK	Yes	Minor	provide killer id to runtime
Proposed HW	Delayed Req-wins	DRW	No*	Yes (local)	timeout counters, buffer for requests
	WriteBurst	WB	No*	Yes (local)	L1 MSHR buffered bit + logic, conflict set sig.

*Serial irrevocability, Serialize on Killer, or Hourglass must be employed to guarantee forward progress.

Table III. Workload Input Parameters and Number of Transactions in Code

Benchmark	Input parameters: Small	Input parameters: Medium (+)	Txs.
Genome	-g256 -s16 -n16384	–g512 -s32 -n32768	5
Intruder	-a10 -l4 -n2048 -s1	-a10 -l16 -n4096 -s1	3
KMeans-h	-m15 -n15 -t0.05 -i n2048-d16-c16	-m15 -n15 -t0.05 -i n16384-d24-c16	3
Labyrinth	-i random-x32-y32-z3-n96	-i random-x48-y48-z3-n64	3
SSCA2	-s13 -i1.0 -u1.0 -l3 -p3	-s14 -i1.0 -u1.0 -l9 -p9	3
Vacation-h	-n4 -q60 -u90 -r16384 -t4096	-n4 -q60 -u90 -r65536 -t4096	1
Yada	-a20 -i 633.2	-a10 -i ttimeu10000.2	6
Deque	100K ops., 1K dummy work		1
Btree	100K ops., 20K preloads, 25% ins.		2
Water	64 molecules		7
Radiosity	-batch		32

irrevocability, and hourglass do not require any kind of hardware additions, our proposed software-based techniques require minor changes to provide core local information to the runtime, while our proposed hardware-based techniques need simple hardware additions that are core-local and retain the requester-wins nature of the HTM. Both LogTM and EL_T require coherence changes that affect communication between cores. Most proposed techniques can experience livelock conditions due to contention, so they should be executed in conjunction with a contention livelock-free technique like serial irrevocability, serialize on killer, or hourglass.

4. SIMULATION ENVIRONMENT AND METHODOLOGY

4.1. Workloads

We use the STAMP benchmark suite as workloads to drive our experiments. These workloads provide significant diversity in behavior and are expected to be good examples of transactional use cases and programming style. We choose to exclude the application *bayes* from our analysis due to large variability in execution times, which are very sensitive to random interleavings. In addition to STAMP, we include four workloads that have been used in a number of TM studies in the past, *water* and *radiosity* from SPLASH2 [Woo et al. 1995], and two microbenchmarks—*deque* and *btree*. Table III lists the command line parameters used in experiments in this article. We simulate both small and medium datasets for STAMP workloads, following the recommended input parameters [Cao Minh et al. 2008].

Table IV. Architectural and System Parameters

Parameter	Values
Cores	8 in-order 2GHz ×86 cores, 1 IPC for non-memory instructions
L1 I&D Caches	64KB 8-way, private, 64B lines, 1-cycle hit latency
L2 Cache	8MB 16-way, shared, 64B lines, 12-cycle uncontended hit latency
L2 Directory	L2-Directory Full-bit vector sharer list; 6-cycle latency
Memory	4GB, 100-cycle latency DRAM lookup latency
Interconnect	2D Mesh, 64-byte links, 1-cycle link latency
Exponential Backoff	Randomized exponential backoff with saturation after n steps. Backoff range [1..2^n] where n is 8; Backoff multiplier factor: 117
Hourglass	4 retries before becoming toxic (best observed results)
Serial Irrevocability	8 maximum number of retries before executing in serial mode
Serialize on Address Serialize on Killer	The rw-lock implementation was stripped from the Linux kernel and is very similar to the one used in GCC to implement serial irrevocability.
Delayed requester-wins	Timeout latencies (min/max): 64/1024 cycles
WriteBurst	Number of MSHR to buffer store miss information: 32

4.2. Simulation Environment

All experiments in this article have been performed using the GEM5 simulator [Binkert et al. 2011]. TM support that had been stripped from Ruby [Martin et al. 2005] upon integration into GEM5 has been plugged back in for the purposes of this article. The setup uses the *timing simple* processor model in GEM5. The memory system is modeled using Ruby. A distributed directory coherence protocol on a mesh-based network-on-chip is simulated. Each node in the mesh corresponds to a processing core with private L1 instruction and data caches and a slice of the shared L2 cache with associated directory entries. Table IV describes key architectural parameters used in the experiments, as well as parameters used in the evaluated livelock avoidance mechanisms. For each workload-configuration pair, we gathered average statistics over 5 randomized runs designed to produce different interleavings between threads. For LogTM, we used the hybrid resolution policy that prioritizes older writers by allowing their write requests to abort younger transactions [Bobba et al. 2007].

To isolate our study from the effects of aborts caused by hardware resource limitations (e.g., cache capacity), our design includes an ideal transactional victim cache which is able to hold any number of speculatively modified cache lines when they are evicted from the L1 data cache while a transaction is executing. This allows transactions with large footprints to commit entirely in hardware, without having to resort to software fallback mechanisms. When a memory reference inside a transaction misses in the L1 cache but hits in the transactional victim cache, a penalty of only one extra cycle over the L1 hit time is applied. The transactional victim cache is flushed on abort and its contents drained to the L2 cache on commit. Evictions of speculatively read lines are also tolerated by our design, which uses perfect read signatures to track read sets. Such lines are not placed in the transactional victim cache and so they need to be fetched back from the L2 if need be.

Table V shows usage of the Victim Cache (VC) for the simulated workloads. We do not show data for workloads that do not make use of the victim cache during their execution. Even though we use an unbounded victim cache, as can be seen in the table, the number of lines that go into the victim cache is very small for all the workloads, with the exception of *labyrinth*. Half of the workloads do not use the victim cache at all, and for those that use it, the maximum occupancy reached by the victim cache stays below 20 cache lines except in *labyrinth*. Moreover, the percentage of transactions that commit without using the victim cache at all is high. Thus, designs that have replacement

Table V. Victim Cache Statistics for Evaluated Workloads on Committed Transactions
Numbers have been averaged over 5 simulated runs with 8 cores using the exponential backoff configuration.

Application	Max. Occupancy	Avg. Occupancy	%Commits without VC
Btree	1	1.00	99.99
Genome+	1	1.00	99.99
Labyrinth	32	6.84	66.20
Labyrinth+	433	378.98	56.00
Vacation-h	1	1.00	99.99
Yada	9	1.58	96.20
Yada+	19	1.65	95.50

policies with some priority for transactional data, or that incorporate transactional bookkeeping in deeper levels of the memory hierarchy (private L2 caches) will likely be able to execute the transactions defined in these workloads entirely in hardware.

4.3. HTM Support in the Coherence Protocol

We have introduced minor changes in one of several coherence protocol implementations available in GEM5. The primary intent is to make a few simple changes that permit buffering of speculative updates in the private L1 cache without maintaining an undo-log. This brings the model as close in function as possible to requester-wins HTM implementations that may soon be available. We extended a typical MESI directory protocol available in the GEM5 release to support silent replacements of lines in *E* (exclusive) state. This is implemented via *yield* response messages that are sent by a former L1 exclusive owner to the L2 directory in response to a forwarded request for a line that is no longer present (after it was silently replaced). Through this feature, the protocol is then able to integrate speculative data versioning in private L1 caches at no extra cost. When a transaction aborts, it simply flush-invalidates all speculatively modified lines in its L1 data cache, which will eventually appear as silent E replacements to the directory. When it commits, it makes such updates globally visible by clearing the Speculatively Modified (SM) bits in L1 cache. To preserve consistent nonspeculative values, transactional writes to M-state lines that find the SM bit not asserted must be written back to the L2 cache. These fresh speculative writes are performed without delay in L1 cache while a consistent copy of the data is simultaneously kept in the MSHR until the writeback is acknowledged (required in case of forwarded requests). Furthermore, transactional exclusive coherence requests (TGETX) must be distinguished from their nontransactional counterparts (GETX) both by L1 cache and L2 directory controllers. For TGETX, the L1 exclusive owner must send the data to both the L1 cache requester and the L2 cache (in order to preserve pretransactional values), whereas for GETX requests it is sufficient with a cache-to-cache transfer, and in these cases the L2 directory expects no writeback.

4.4. Experiments and Metrics

We use execution time breakdowns to identify possible sources of overhead and compare them across the studied mechanisms. Execution times account for memory system effects by allowing the cache hierarchy and locality characteristics of the application to affect the metric. Execution breakdowns are broken down into several components listed in Table VI based on the number of cycles spent performing the corresponding activity in all the cores. Some components are present only in certain configurations. Tables of results also show different statistics depending on the evaluated proposal, and include abort rates which indicate the fraction of transaction executions that result in aborts. This metric, when looked at in conjunction with execution time, provides a better picture of the efficacy of various contention and livelock mitigation techniques

Table VI. Various Components in Execution Time Breakdown Plots

Component	Abbrev.	Description
Non-transactional	non-tx	Time spent execution non-transactional code
Barrier	barrier	Time spent waiting at barriers
Useful-Transactional	useful	Time spent executing transactions that commit
Wasted-Transactional	wasted	Time spent executing transactions that abort
Waiting in serial lock	wait-serial	Time spent waiting for an irrevocable transaction to complete
Waiting in address lock	wait-address	Time spent waiting for a conflicting transaction on the same address to complete
Waiting for killer	wait-killer	Time spent waiting for our killer transaction to complete
Serial irrevocable	serial	Time spent executing an irrevocable transaction
Token usefull	token	Time spent in usefull transactions with the token (hourglass)
Backoff	backoff	Time spent performing exponential backoff
Stall	stall	Time spent waiting for a memory request to complete in LogTM, or by the delayed requester-wins conflict resolution

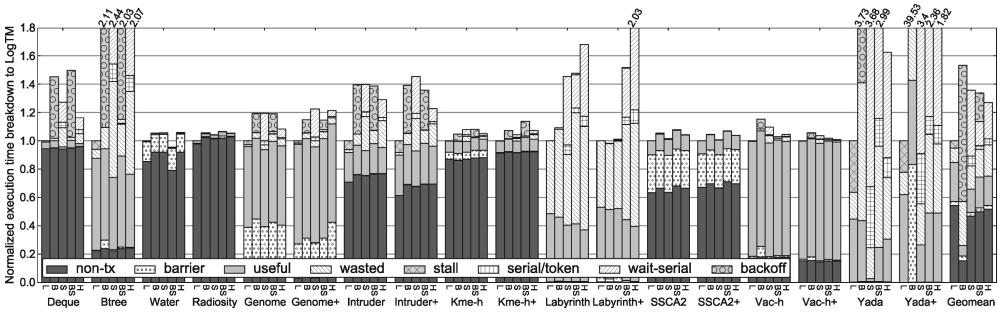

Fig. 3. Relative performance of existing livelock mitigation techniques for 8 core runs (L: LogTM, B: Exponential backoff, S: Serial irrevocable, BS: Exponential backoff and serial irrevocable, H: Hourglass).

evaluated. Finally, we also use execution times for different techniques normalized to single-thread execution time to compare their scalability.

5. EVALUATION

We first evaluate existing techniques in depth to identify possible sources of overhead. Later, we evaluate our proposed software-based and hardware-based techniques. Finally, we conclude with a scalability comparison for the evaluated proposals.

5.1. Evaluation of Existing Techniques

Figure 3 compares the performance (execution times) of the existing techniques: exponential backoff (B), serial irrevocability as implemented in GCC (S), a design that combines both exponential backoff and serial irrevocability (BS), and hourglass (H). LogTM execution times have been used as the basis for normalization, with the breakdown for the configuration shown using the bar marked *L*. In BS, serialization occurs when a transaction fails to commit even after having retried 8 times applying an exponential backoff. For a description of breakdown components, see Table VI.

Serial irrevocability imposes a performance cost because any parallelism among concurrent transactions is precluded. Frequent entries into this mode may result in severe performance degradation. Exponential backoff alone performs badly too. From the figure, it is clear that when contention is present (e.g., in applications like *deque*, *btree*, *genome*, *intruder* and *yada*), just relying on serial irrevocability or exponential backoff can result in performance degradation ranging from 20% to about 40% in

Table VII. Key Metrics for Existing Techniques

	%Saturation	%Irrevocable/Token			%Token aborts	%Abort Rate			
Application	B	S	BS	H	H	B	S	BS	H
Deque	0.12	67.71	0.19	43.45	19.7	36.2	95.2	65.7	78.0
Btree	4.74	11.83	0.62	8.38	20.6	14.7	65.9	25.0	44.5
Water	0.00	1.19	0.00	2.08	6.3	2.2	13.9	2.1	16.8
Radiosity	0.47	0.12	0.00	0.05	26.7	0.4	2.4	0.5	1.2
Genome	2.52	1.39	0.15	0.38	44.8	4.6	16.3	6.4	8.3
Genome+	3.51	0.70	0.06	0.17	34.5	2.5	8.4	3.1	3.5
Intruder	0.61	9.45	0.08	1.31	79.9	14.6	59.8	18.3	44.6
Intruder+	1.05	9.92	0.07	2.32	53.9	10.6	60.9	14.2	38.1
KMeans-h	0.00	8.59	0.00	3.61	29.3	6.0	53.0	7.1	31.2
KMeans-h+	0.18	6.29	0.04	4.52	34.9	7.3	43.6	11.3	38.8
Labyrinth	3.65	18.37	1.74	7.88	36.9	34.5	71.9	50.0	61.9
Labyrinth+	0.95	0.42	2.85	8.89	37.3	30.4	32.0	47.2	63.2
SSCA2	0.00	0.08	0.00	0.12	0.0	0.1	2.2	0.1	1.1
SSCA2+	0.00	0.03	0.00	0.06	0.3	0.1	0.9	0.1	0.6
Vacation-h	8.04	0.87	0.00	0.46	1.1	2.1	12.3	1.7	5.0
Vacation-h+	11.83	0.41	0.01	0.17	0.0	0.8	6.8	1.2	1.8
Yada	33.86	46.42	12.26	13.72	34.8	45.3	90.9	71.9	62.8
Yada+	90.17	29.24	5.40	10.42	32.6	80.9	83.1	52.6	55.7

intruder, 2–2.5× in *btree* and several times (3–4×) in *yada*. Even a small portion of time in serial irrevocable mode results in significant time spent by other threads waiting for the irrevocable execution to finish (wait-serial). This overhead is expected to become worse as thread count increases. Though the combination of exponential backoff and serial irrevocability (BS) performs marginally better, all three livelock mitigation techniques perform comparably. Hourglass contention manager shines here being 6.8% better than BS. However, note that a performance gap of 26.8% can be seen between the baseline (LogTM with conservative deadlock avoidance using timestamp priorities) and the best existing technique (hourglass).

Table VII shows some key metrics for different existing techniques evaluated in this section. The column *%Saturation* indicates the percentage of backoff events where backoff had saturated. Note that we use exponential backoff where the range of possible backoff periods stops growing after a certain number of consecutive aborts. We find that *yada* and *btree* experiences this event often, a sign of contention being persistent, and that larger backoff periods might be beneficial in this particular workloads. The columns under the head *%Irrevocable/Token* indicate the percentage of transactions that ran irrevocably (as a fraction of the total number of committed transactions) when using serial irrevocability as fallback (configurations S and BS), or the percentage of transactions that acquired the global token when using hourglass contention management (configuration H). The last three columns under the head *%Abort Rate* show the percentage of aborts encountered in each configuration as a fraction of the total number of transaction starts (including restarts)–`aborts/(aborts+commits)`. We observe that high-contention workloads running on configuration S (without backoff) enter in irrevocable mode far more often than when using it in conjunction with backoff (configuration BS). This is expected because backoff preempts immediate restart of transactions that are likely to abort their killers. Thus the use of backoff is recommended, specially in contended scenarios. Though hourglass outperforms all other techniques, it is susceptible to performance degradation under contention, particularly if transactions are large. The column *%Token aborts* indicates the abort rate for

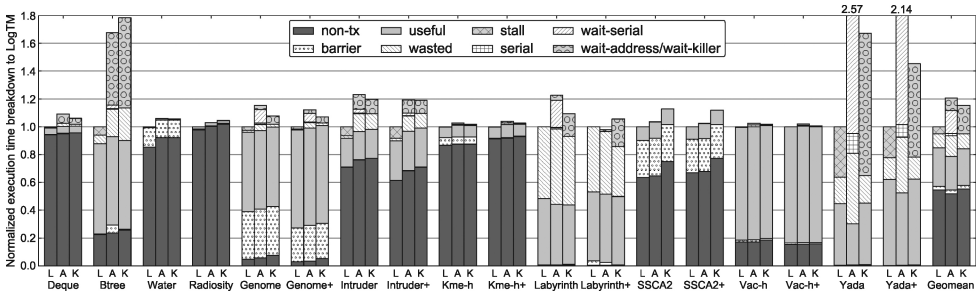

Fig. 4. Relative performance of proposed software-based techniques for 8 core runs (L: LogTM, A: Serialize on conflicting address (SoA), K: Serialize on killer transaction (SoK)).

transactions that executed while holding the hourglass token, a large number of aborts in this mode may cause a penalty similar or even larger than that of serial irrevocability.

5.2. Evaluation of Proposed Techniques

5.2.1. Software-based Techniques. Figure 4 compares the performance of software-based techniques proposed in this article. The numbers are again normalized using LogTM as a baseline. This allows us to compare visually the improvements over techniques discussed in the previous section. Data has been presented for two configurations—Serialize on conflicting Address (SoA) and Serialize on Killer transaction (SoK).

We notice modest improvement in overall performance using either technique over existing techniques (shown in Figure 3). SoK performs the best, reducing the performance gap from the baseline to 15.3%, being slightly better than SoA. However, we see that both these techniques suffer when contention is high and transactions are large. This is evident from the execution times for *btree* and *yada*. In such cases, these techniques turn out to be substantially slower (1.6×–2.5×) than LogTM. Note that in the case of *btree* SoA performs better than SoK, while the opposite trend is seen in the case of *yada*. In *btree,* there is some overlap expected among conflicting addresses because a tree is being accessed. This is, however, not the case in the mesh-refinement algorithm used by *yada*. Previous work [Negi et al. 2012a] has shown that *yada* typically has a very large number of conflicting addresses that do not show much repetition. Moreover, contention in *yada* tends to occur among groups of threads working on the same region of memory. Hence, SoK, with its per-thread locks, fits this case well.

Table VIII shows statistics for the evaluated software-based techniques. From the table, we can see that, in fact, the percentage of transactions executed in serial irrevocable mode is substantially lower in SoA when compared to existing techniques. The column labeled *%Aborts cycle*, shows the percentage of aborted transactions that are allowed to restart without waiting on the killer transaction because a cyclic dependence would occur otherwise. Note that this value stays relatively low for all workloads, keeping additional aborts that might occur due to nonserialized transactions low. In fact, the abort rates for *yada* in SoK are substantially lower than in LogTM; however, LogTM still performs better because, with large transactions, the overheads of serializing grow rapidly.

SoA significantly reduces the number of transactions that run in irrevocable mode when compared to existing techniques, which translates into lower overheads waiting for the serial lock. This is evident upon comparing numbers in Table VIII to those in Table VII. Time waiting on transactions executing on the same conflicting address is generally small (wait-address), although this overhead remains visible in *intruder*

Table VIII. Key Metrics for Software-based Proposed Techniques

	%Irrevocable	%Aborts cycle	%Abort rate		
Application	SoA	SoK	SoA	SoK	LogTM
Deque	0.00	3.98	44.1	31.2	9.5
Btree	0.29	3.08	20.0	22.2	7.9
Water	0.00	4.38	2.1	3.7	0.6
Radiosity	0.01	8.06	0.9	0.7	0.5
Genome	0.45	2.61	8.8	4.0	3.1
Genome+	0.15	2.28	3.7	1.7	1.1
Intruder	0.78	6.96	25.7	23.1	15.4
Intruder+	0.26	6.52	17.5	16.4	11.8
KMeans-h	0.00	11.46	8.7	4.5	0.2
KMeans-h+	0.04	6.03	16.1	6.1	0.2
Labyrinth	0.77	0.27	32.2	26.4	27.9
Labyrinth+	0.00	0.90	28.0	23.6	30.2
SSCA2	0.00	2.55	0.2	0.1	0.0
SSCA2+	0.00	3.39	0.1	0.1	0.0
Vacation-h	0.00	0.00	1.7	0.9	0.6
Vacation-h+	0.00	0.00	0.9	0.5	0.2
Yada	5.13	1.42	50.4	14.8	32.3
Yada+	2.86	1.26	36.5	10.3	18.1

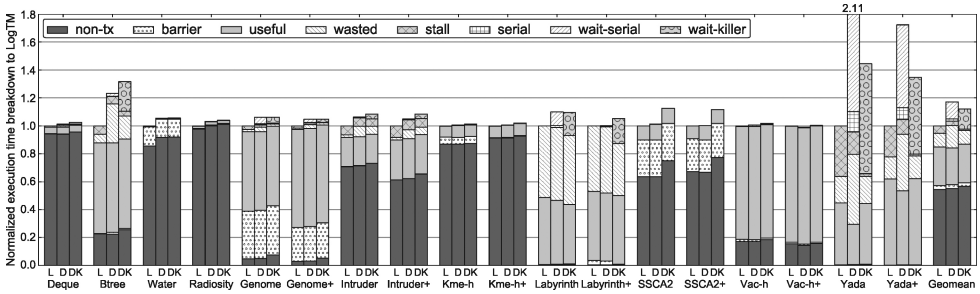

Fig. 5. Relative performance of delayed requester-wins technique for 8 core runs (L: LogTM, D: Delayed req-wins with serial irrevocability, DK: Delayed req-wins with SoK).

and *btree* because these are benchmarks with a larger number of read-modify-write transactions that conflict on a small set of addresses.

5.2.2. Hardware-based Techniques. Figure 5 shows relative performance of two new livelock mitigation techniques based on the Delayed-Requester-Wins (DRW) mechanism. Since DRW does not guarantee forward progress, we must have some form of software fallback to break persistent livelocks. The first DRW-based scheme (bar D in Figure 5) uses serial irrevocability as fallback, while the second scheme uses SoK as fallback (bar DK). A version with hourglass as fallback was evaluated yielding lower performance (results not included); serial irrevocability and SoK are more efficient at bypassing short hotspots of high contention.

Performance differences between the two techniques are most noticeable in applications with large transactions or with moderate to high contention. Notice the large wait time due to serial irrevocability in *yada*. The drastic improvement in performance over serial irrevocability when using SoK is the result of improved parallelism because only those transactions that actually conflict wait. Other contended applications like *intruder* and *genome* obtain the best results seen so far, being only a few percent

Table IX. Key Metrics for Delayed Requester-Wins in Conjunction with Serial Irrevocability and SoK

	%Commits with timeout		%Unexpired timers		%Irrevocable	%Abort Rate	
Application	DRW-S	DRW-SoK	DRW-S	DRW-SoK	DRW-S	DRW-S	DRW-SoK
Deque	32.72	21.49	71.85	74.54	0.09	24.0	13.9
Btree	12.84	9.44	49.67	46.58	0.27	24.7	15.3
Water	0.89	2.67	84.21	91.43	0.00	1.5	0.9
Radiosity	0.30	0.22	96.27	96.71	0.01	0.6	0.4
Genome	1.24	0.71	49.19	47.30	0.28	6.5	3.0
Genome+	0.57	0.29	52.12	48.56	0.07	2.5	1.3
Intruder	12.70	11.25	69.87	71.41	0.16	16.8	10.8
Intruder+	9.10	7.89	75.57	76.54	0.09	12.3	8.0
KMeans-h	4.29	2.72	73.34	65.21	0.00	1.7	1.8
KMeans-h+	4.21	3.66	79.45	78.23	0.00	1.3	1.2
Labyrinth	0.58	0.29	35.29	50.00	1.17	34.7	26.3
Labyrinth+	0.14	0.00	25.00	0.00	0.28	31.8	24.5
SSCA2	0.17	0.10	98.99	100.00	0.00	0.0	0.0
SSCA2+	0.07	0.05	99.69	99.07	0.00	0.0	0.0
Vacation-h	0.60	0.20	84.83	90.91	0.00	1.4	0.9
Vacation-h+	0.22	0.05	72.58	68.75	0.00	0.5	0.3
Yada	11.73	1.71	42.36	26.83	6.12	54.6	13.9
Yada+	6.34	1.37	39.72	30.80	2.89	38.7	10.0

behind LogTM. In *yada*, DRW helps some transactions to commit that would otherwise have to abort, while SoK ensures that aborted transactions do not abort their killers upon restart. *Btree* also benefits substantially from DRW, experiencing a considerable performance boost with respect to previous evaluated techniques. Table IX shows the percentage of commits that had active timeouts, which delayed (buffered) conflicting requests from remote cores instead of aborting the local transaction. *Intruder*, *yada,* and *btree* benefit substantially from this fact. The head *%Unexpired timers* shows that applied timers tend to be cancelled before they expire, allowing the transaction to continue execution. Table IX also shows that abort rates obtained for DRW-SoK, which are considerably lower with respect to other proposals across all workloads.

The use of timestamp priorities and reductions in wasted execution time due to the possibility to retry conflicting accesses (effectively stalling a lower priority transaction) still allows LogTM to perform significantly better under contention. However, even though DRW does not use additional coherence messages or timestamps, it has an average performance close to that seen in LogTM. Using SoK as fallback, we observe a performance gap of about 12.1%, which can be largely attributed to the results obtained in *yada* and *btree*, as other workloads perform considerably closer to LogTM.

Figure 6 presents an execution time breakdown for the WriteBurst technique. Two versions have been evaluated: One with serial irrevocability (bar W), and one with SoK (bar WK) as fallback mechanism to guarantee forward progress. Again, a version with hourglass as fallback was evaluated (not shown) delivering slower performance.

In workloads where buffering stores can hide conflicts between transactions—by shrinking the window of time in which a transaction is susceptible to abort due to remote readers—using serial irrevocability proves to be slightly better (*btree*, *genome*, and *intruder*). This is due to the fact that transactions can restart immediately as long as they do not reach the threshold to execute in irrevocable mode, and under low contention, this is beneficial. However, if contention is still present, serial irrevocability again imposes a severe performance penalty, see *yada*. When using WriteBurst in conjunction with SoK as a fallback mechanism, there is a slight penalty in applications

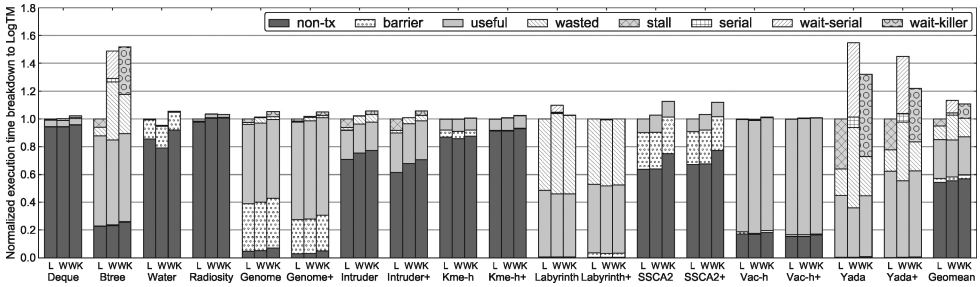

Fig. 6. Relative performance of the WriteBurst technique for 8 core runs (L: LogTM, W: WriteBurst with serial irrevocability, WK: WriteBurst with SoK).

Table X. Key Metrics for the WriteBurst Mechanism in Conjunction with Serial Irrevocability and SoK

	Max. Stores Buffered		Avg. Stores Buffered		%Irrevocable	%Abort Rate	
Application	WB-S	WB-SoK	WB-S	WB-SoK	WB-S	WB-S	WB-SoK
Deque	3	3	2.89	2.89	0.04	27.3	25.8
Btree	11	12	3.42	3.42	2.40	40.5	24.7
Water	2	2	1.86	1.87	0.00	1.9	1.8
Radiosity	18	15	1.10	1.11	0.00	0.8	0.6
Genome	12	11	1.68	1.72	0.02	4.8	3.0
Genome+	12	11	1.51	1.52	0.02	2.1	1.3
Intruder	17	17	2.21	2.19	0.02	13.0	12.0
Intruder+	19	20	1.79	1.78	0.01	10.5	8.5
KMeans-h	2	2	1.69	1.69	0.00	3.9	3.5
KMeans-h+	2	3	1.73	2.38	0.00	2.9	3.1
Labyrinth	32	32	7.17	7.15	0.58	30.7	27.1
Labyrinth+	32	32	13.46	13.56	0.14	26.8	26.9
SSCA2	2	2	1.15	1.14	0.00	0.2	0.1
SSCA2+	2	2	1.10	1.10	0.00	0.1	0.1
Vacation-h	8	8	1.57	1.57	0.00	1.1	0.8
Vacation-h+	5	5	1.48	1.48	0.00	0.4	0.4
Yada	31	32	6.13	6.84	3.13	42.4	14.9
Yada+	32	32	6.26	6.66	1.93	31.5	10.4

where restarted transactions may now not conflict due to the WriteBurst mechanism. However, it proves to be much more effective for large transactions with moderate to high contention (*yada*). Overall, this approach is only 10.5% slower than LogTM.

Table X provides information about the maximum and average number of buffered stores per committed transaction. As can be seen, *labyrinth* and *yada* exhausted the buffer capacity for some transactional executions, and maintain a relatively high average number of buffered stores. *Btree*, *radiosity*, *Intruder,* and *genome* also have a higher number of maximum stores buffered when compared to the rest of the workloads, but their average usage is low. Serial irrevocability, as observed in the breakdown, is only used by *btree* and *yada*, where contention is still an issue.

This high usage of the buffers in *labyrinth* and *yada* may imply that these workloads can benefit from larger buffering capacity and that they would also be sensitive to a lower number of MSHRs. We ran experiments using WB-SoK with 16 and 64 MSHRs and observed that only *labyrinth* and *yada* experienced changes in performance compared to the results gathered using 32 entries. When 16 MSHRs are available, *labyrinth* and *labyrinth+* see performance drops of 7.3% and 5.4%, while *yada* and

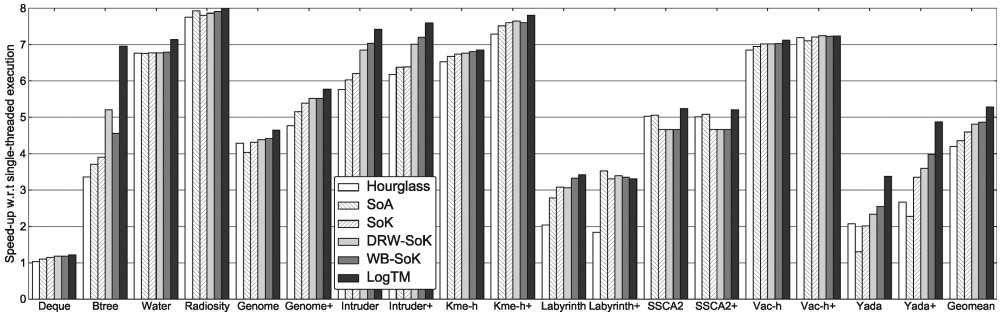

Fig. 7. Relative performance of 8 core runs using existing, software, and hardware techniques with respect to single-threaded execution.

yada+ drop by 9.0% and 5.2%, respectively. On the other hand, when the number of registers is set to 64, *yada* and *yada+* improve by 5.0% and 6.8%, respectively, while both *labyrinth* and *labyrinth+* show roughly the same performance levels of the 32-MSHR configuration. We observed that the substantial improvement seen in *yada* is due to its maximum usage of 60 MSHR, whereas *labyrinth* uses around 40 entries.

5.3. Performance Overview of Proposed Techniques

In this section, we compare relative performance of the introduced techniques in software and hardware. Figure 7 compares scalability for 8-core runs using a subset of the configurations we have discussed earlier.

We show the best performing existing technique, hourglass, which has significant drops in performance under contended scenarios, but can be a good choice when contention is low. Overall, the proposed hardware schemes perform better than their software counterparts, this is specially noticeable in contended applications like *btree*, *intruder,* and *yada*. In applications where contention is mild like in *water*, *radiosity*, *SSCA2*, or *vacation,* SoA and SoK present competitive performance, being on par or even slightly better than hardware proposals (e.g., *SSCA2*). LogTM, plotted as the last bar, performs the best, especially under contention (*btree* and *yada*), where timestamp priorities become more useful, though the proposed schemes can achieve similar performance for most workloads.

This comparison highlights the need for basic livelock mitigation techniques in hardware (specially in contended scenarios), if not full-fledged forward progress guarantees which may be better implemented in software. As long as hardware techniques can effectively limit the need for software intervention, the performance cost associated with providing strong progress guarantees in software would be manageable.

6. RELATED WORK

HTM proposals in the literature have typically provided forward progress guarantees using transaction priorities (e.g., through timestamps) [Moore et al. 2006; Bobba et al. 2007] or lazy contention management [Hammond et al. 2004; Tomić et al. 2009; Negi et al. 2012b]. However, the simplicity with which requester-wins HTMs [Click 2009; Intel Corporation 2012; Chung et al. 2010] can be incorporated in hardware has resulted in such HTMs being the first ones to be widely accessible. As we have shown in this article, such designs tend to be susceptible to performance degradation through transient or persistent livelocks. To the best of our knowledge, prior work has only noted this fact in passing, without presenting in-depth analyses of its performance implications or evaluating solutions that enhance forward-progress properties of requester-wins HTMs.

However, we would like to point out the connection between livelock mitigation and contention management. Extensive research into management of conflicting transactions has been undertaken in both HTM and STM. Designs that manage contention better are also less susceptible to livelock. In the area of STM, a variety of ways in which transactional conflicts could be handled have been evaluated. STM implementations allow great freedom in contention management policy design. Scherer III and Scott [2005] have evaluated a range of such options—Polite, which employs exponential backoff in a manner similar to our implementation; Karma and Eruption, which prioritize transactions based on the amount of work they have done; Kindergarten, where transactions accessing an object of contention take turns; and Polka, where exponential backoff and Karma are used together. Prioritized contention management policies (like Karma), with appropriate instrumentation in code, are relatively simple to implement in software. However, hardware implementations necessitate mechanisms to award and transport such priorities among processing units and, more importantly, mechanisms to notify and respond to decisions based on their use. The key attraction of requester-wins HTM in hardware design is the lack of any such requirement. The cache hierarchy and protocols do not change, changes local to processing units being sufficient to determine and rectify conflicts.

Entry into the serial irrevocable mode in GCC aborts all concurrent transactions and prevents new ones from starting. Toxic transactions [Liu and Spear 2011] present a less drastic way—hourglass—to allow transactions that repeatedly abort due to conflicts to complete successfully. The mechanism requires such transactions to become toxic, that is, prevent new ones from being scheduled (or re-executed upon abort) by acquiring a token. This gives a chance to concurrent nonconflicting transactions to complete successfully. We have included this strategy in this article, showing that is quite effective, being the best contendant amongst existing techniques.

Dolev et al. [2008] have proposed CAR-STM, which provides, in software, two methods to mitigate the adverse effects of conflicts. It maintains, for each processing core, a queue of transactions to be serialized on that core. An aborted transaction is rescheduled by queueing it on the conflicting core. In addition to this mechanism, a predictive scheduling approach assigns transactions to cores with which they are likely to conflict. However, this mechanism views transactions as tasks to be scheduled and thus imposing scheduling overheads particularly in high contention scenarios. Another predictive approach is used in the Shrink contention manager described in Dragojević et al. [2009].

In the context of HTMs, prior work [Bobba et al. 2007] has identified several pathological conditions that can beset certain contention management policies. Requester-wins systems are inherently eager conflict resolution systems and suffer from pathologies that such systems are susceptible to. However, the absence of transaction priorities swaps starvation problems for increased risk of livelocks. For example, the requester-wins design treats reads and writes at an equal footing, thus avoiding the problem of starving readers/writers. However, the livelock risk, termed friendly fire in Bobba's paper, is present. Our article aims to estimate the likelihood of this risk and presents some new techniques to mitigate or avoid it.

Hybrid approaches have also been investigated. In particular, Hybrid-NOrec [Dalessandro et al. 2011] describes the implementation of a hybrid TM system on best-effort HTM. The design allows software and hardware transactions to coexist, although concurrency among such transactions is restricted rather severly. High-performance variants of this approach require the ability to issue nontransactional loads from within a transactional context.

Further research in HTM has investigated the use of reactive and proactive scheduling strategies [Yoo and Lee 2008; Blake et al. 2009] to enhance parallelism and limit

speculation when it is likely to fail. These proposals track conflicts between transactions and use this information in the future to predict contention and decide whether or not to stall a transaction when a transaction-begin primitive is encountered. Dependence-aware TM [Ramadan et al. 2008] tracks dependencies between concurrent transactions, supplying uncommitted data to dependent transactions and ensuring that commits occur in proper order. Cyclic dependencies are broken by aborting one of the transactions when a cycle is detected. These proposals tend to rely on HTMs that are more sophisticated and significantly more complex than a requester-wins design and, hence, are unlikely to be adopted soon by hardware vendors.

7. CONCLUSIONS

In this section, we summarize key results and insights gathered during this study. These can be categorized under two heads: For Programmers and For Architects.

7.1. For Programmers

Livelocks present a real and rather severe problem in requester-wins best effort HTMs. Even when cyclic dependencies may not arise among transactions, performance degradation due to transient livelocks may still occur because of repeated conflicts between an *aborter* and a restarted *abortee*. Exponential backoff is quite effective at mitigating adverse effects of livelocks. However, it does not guarantee freedom from livelocks. It must be used in conjunction with serial irrevocability to ensure forward progress. However, the TM runtime should not be very eager when deciding to enter serial irrevocability, as this can potentially create pathological situations wherein applications with little contention may show severe performance degradation due to frequent serialization because of the contention created by the serialization mechanism itself. As we show in this article, serialization should be done in stages. Initially, using less severe techniques like Hourglass, SoA, or SoK, permit much greater levels of parallelism before falling back to serial irrevocability.

7.2. For Architects

Bare-bones requester-wins HTM support, while being a good, low-complexity way of introducing practical TM in the real world is not safe from livelocks even in lightly contended scenarios. Although software strategies can prevent livelocks from precluding forward progress, they can also impose a performance penalty which in several cases is rather steep. Simple hardware mitigation strategies are quite useful in this context. By delaying conflict resolution, the architectural simplicity of requester-wins HTM designs can be retained while simultaneously mitigating the possibility of livelock and overheads associated with it. As we have shown in this study, this can be easily done by deferring processing of conflicting coherence requests (DRW) or delaying when writes are injected into the memory hierarchy (by buffering store misses). While such schemes may not guarantee freedom from livelock, they prove to be quite effective in avoiding them in many transactional use cases.

ACKNOWLEDGMENTS

The authors would like to thank Per Stenström and all reviewers for their comments and valuable feedback.

REFERENCES

Ananian, C. S., Asanović, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S. 2005. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News*.

Blake, G., Dreslinski, R. G., and Mudge, T. 2009. Proactive transaction scheduling for contention management. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*.

Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 422–426.

Blundell, C., Raghavan, A., and Martin, M. M. K. 2010. RetCon: Transactional repair without replay. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*.

Bobba, J., Moore, K. E., Yen, L., Volos, H., Hill, M. D., Swift, M. M., and Wood, D. A. 2007. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*.

Cao Minh, C., Chung, J., Kozyrakis, C., and Olukotun, K. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*.

Christie, D., Chung, J.-W., Diestelhorst, S., Hohmuth, M., Pohlack, M., Fetzer, C., Nowack, M., Riegel, T., Felber, P., Marlier, P., and Rivière, E. 2010. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European Conference on Computer Systems*.

Chung, J., Yen, L., Diestelhorst, S., Pohlack, M., Hohmuth, M., Christie, D., and Grossman, D. 2010. ASF: AMD64 extension for lock-free data structures and transactional memory. In *Proceedings of the 43th International Symposium on Microarchitecture*.

Click, C. 2009. Azul's experiences with hardware transactional memory. HP Labs, Bay Area Workshop on Transactional Memory.

Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M. L., and Spear, M. F. 2011. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Dolev, S., Hendler, D., and Suissa, A. 2008. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the 27th Symposium on Principles of Distributed Computing*.

Dragojević, A., Guerraoui, R., Singh, A. V., and Singh, V. 2009. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*.

Felber, P., Fetzer, C., Marlier, P., and Riegel, T. 2010. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems 21,* 12.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*.

Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. 2004. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*.

Harris, T., Larus, J., and Rajwar, R. 2010. *Transactional Memory* 2nd Ed. Morgan and Claypool Publishers.

Herlihy, M. and Moss, J. E. B. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*.

Intel Corporation. 2012. Transaction Synchronization Extensions (TSX). In *Intel Architecture Instruction Set Extensions Programming Reference*. 506–529. Retrieved from http://software.intel.com/file/41604.

Jafri, S. A. R., Voskuilen, G., and Vijaykumar, T. N. 2013. Wait-n-GoTM: Improving HTM performance by serializing cyclic dependencies. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*.

Liu, Y. and Spear, M. 2011. Toxic transactions. In *Proceedings of the 6th Workshop on Transactional Computing*.

Lupon, M., Magklis, G., and González, A. 2010. A dynamically adaptable hardware transactional memory. In *Proceedings of 43rd Annual IEEE/ACM International Symposium on Microarchitecture*.

Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., and Wood, D. A. 2005. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*.

Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A. 2006. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*.

Negi, A., Armejach, A., Cristal, A., Unsal, O. S., and Stenström, P. 2012a. Transactional prefetching: narrowing the window of contention in hardware transactional memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*.

Negi, A., Titos-Gil, R., Acacio, M. E., Garcia, J. M., and Stenstrom, P. 2011. Eager Meets Lazy: The impact of write-buffering on hardware transactional memory. *Proceedings of the International Conference on Parallel Processing*.

Negi, A., Titos-Gil, R., Acacio, M. E., Garcia, J. M., and Stenstrom, P. 2012b. Pi-TM: Pessimistic invalidation for scalable lazy hardware transactional memory. In *Proceedings of the 18th International Symposium on High Performance Computer Architecture*.

Ramadan, H. E., Rossbach, C. J., and Witchel, E. 2008. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*.

Scherer III, W. N. and Scott, M. L. 2005. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*.

Shriraman, A. and Dwarkadas, S. 2009. Refereeing conflicts in hardware transactional memory. In *Proceedings of the 23rd International Conference on Supercomputing*.

Shriraman, A., Dwarkadas, S., and Scott, M. L. 2008. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*.

Titos, R., Acacio, M. E., and Garcia, J. M. 2009. Speculation-based conflict resolution in hardware transactional memory. In *Proceedings 23rd International Parallel and Distributed Processing Symposium*.

Titos-Gil, R., Negi, A., Acacio, M. E., García, J. M., and Stenstrom, P. 2011. ZEBRA: A data-centric, hybrid-policy hardware transactional memory design. In *Proceedings of the International Conference on Supercomputing*.

Titos-Gil, R., Negi, A., Acacio, M. E., Garcia, J. M., and Stenstrom, P. 2012. Eager beats lazy: Improving store management in eager hardware transactional memory. *IEEE Transactions on Parallel and Distributed Systems 99*, 2192–2201.

Tomić, S., Perfumo, C., Kulkarni, C., Armejach, A., Cristal, A., Unsal, O., Harris, T., and Valero, M. 2009. EazyHTM: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*.

Waliullah, M. and Stenstrom, P. 2012. Removal of conflicts in hardware transactional emory systems. *International Journal of Parallel Programming*. http://link.springer.com/article/10.1007%2Fs10766-012-0210-0.

Wang, A., Gaudet, M., Wu, P., Amaral, J. N., Ohmacht, M., Barton, C., Silvera, R., and Michael, M. 2012. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*.

Welc, A., Saha, B., and Adl-Tabatabai, A.-R. 2008. Irrevocable transactions and their applications. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. 1995. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual International Symposium on Computer Architecture*.

Yen, L., Bobba, J., Marty, M. M., Moore, K. E., Volos, H., Hill, M. D., Swift, M. M., and Wood, D. A. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*.

Yoo, R. M. and Lee, H.-H. S. 2008. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*.

Received June 2013; revised September 2013; accepted November 2013