# Seer: Probabilistic Scheduling for Hardware Transactional Memory

Nuno Diegues, Paolo Romano, Stoyan Garbatov
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{nmld, paolo.romano, stoyan.garbatov}@tecnico.ulisboa.pt

## ABSTRACT

Scheduling concurrent transactions to minimize contention is a well known technique in the Transactional Memory (TM) literature, which was largely investigated in the context of software TMs. However, the recent advent of Hardware Transactional Memory (HTM), and its inherently restricted nature, pose new technical challenges that prevent the adoption of existing schedulers: unlike software implementations of TM, existing HTMs provide no information on which data item or contending transaction caused abort.

We propose SEER, a scheduler that addresses precisely this restriction of HTM by leveraging on an on-line probabilistic inference technique that identifies the most likely conflict relations, and establishes a dynamic locking scheme to serialize transactions in a fine-grained manner. Our evaluation shows that SEER improves the performance of the Intel TSX HTM by up to 2.5×, and by 62% on average, in TM benchmarks with 8 threads. These performance gains are not only a consequence of the reduced aborts, but also of the reduced activation of the HTM's pessimistic fall-back path.

## Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques - Concurrent Programming

## Keywords

Hardware Transactional Memory; Best-Effort; Scheduling

## 1. INTRODUCTION

***Context.*** Transactional Memory (TM) [16] emerged over the last decade as an attractive alternative to lock-based synchronization. Contrarily to lock-based approaches, in which programmers identify shared data and specify how to synchronize concurrent accesses to it, the TM paradigm requires only to identify which portions of the code have to execute atomically, and not *how* atomicity should be achieved.


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

ACM 978-1-4503-3588-1/15/06$15.00.
DOI: http://dx.doi.org/10.1145/2755573.2755578.

The simplicity and potential of TM has motivated many advances in software prototypes (STMs both in shared memory, e.g. [14, 12, 7, 9] and distributed systems, e.g. [17, 22]) as well as in hardware. Our focus is on Hardware implementations of Transactional Memory (HTM), which have recently entered the realm of mainstream computing since Intel shipped its first HTM — Transactional Synchronization Extensions (TSX) [27] — in its commodity processors.

***Problem.*** Due to the speculative nature of TM, transactions are likely to be restarted and aborted multiple times in conflict prone workloads. This has motivated a large body of research on scheduling techniques, whose key idea is to serialize the execution of transactions that are known to generate frequent aborts. However, most of existing scheduling techniques were designed to operate with software implementations of TM (STM), and rely on specific support provided by the STM to gather knowledge on the conflicts that occurred between transactions. Typically, upon a transaction abort, the STM library can report back to the scheduler which specific memory access and concurrent transaction dictated the abort. This is illustrated in Figure 1, where we depict transaction $T_1$ aborting due to a read-write conflict with a concurrent transaction $T_2$. An STM library is able to report this precise information back to a TM scheduler.

With the adoption of HTMs such as Intel TSX, however, we lose much of this ability. When a hardware transaction is aborted, the feedback is limited and insufficient to pinpoint which transaction caused the abort. As shown in [10], and exemplified in Figure 1, these HTMs merely distinguish between a conflict and other abort causes (e.g., exceeding hardware buffers). For this reason, schedulers for STMs

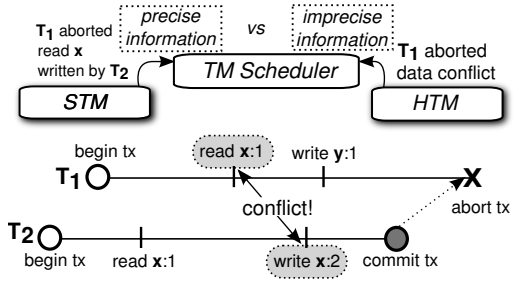


**Figure 1: Two transactions causing a conflict. The information returned by the TM varies depending on its nature: STMs are able to precisely identify the source of the abort, whereas commodity HTMs provide only a coarse categorization of the abort.**

fall short because they rely on precise information, whereas HTMs are only capable of providing *imprecise information*.

***Contributions.*** In this work we introduce SEER, the first scheduler (to the best of our knowledge) to address the HTM restrictions discussed above. The key idea of our proposal is to gather statistics to detect, in a lightweight but possibly imprecise way, the set of concurrently active transactions upon abort and commit events. This information is used as input for an on-line inference technique that uses probabilistic arguments to identify conflict patterns between different atomic blocks of the program in a reliable way, despite the imprecise nature of the input statistics. The final step consists in exploiting probabilistic knowledge on the existence of conflict relations to synthesize a fine-grained, dynamic (i.e., possibly varying over time) locking scheme that serializes "sufficiently" conflict-prone transactions.

A noteworthy feature of SEER is that it relies on reinforcement learning techniques to self-tune the parameters of the probabilistic inference model. To this end, SEER relies on a stochastic hill-climbing technique that explores the configuration space of the model's parameter, while gathering feedback at run-time about the application running and accordingly adjusting the granularity of the locking scheme. Indeed, an appealing characteristic of this dynamically inferred locking scheme is that it does not need to be perfect (e.g., it can suffer of false negatives) in capturing conflicts between atomic blocks of the application, since correctness for transactions is still enforced by the underlying HTM.

SEER includes also an additional novel mechanism that is designed to address another performance pathology of existing HTM systems: when multiple hardware threads are concurrently active on the same physical core, the likelihood of incurring in aborts due to capacity exceptions can grow to such an extent that it cripples performance. This is a direct consequence of the fact that the information used by the HTM concurrency control algorithm is entirely stored in the CPU caches, which may be shared by hardware threads running on the same core. SEER copes with this issue by introducing a simple, yet effective abstraction, the *core lock*, which serializes the execution of hardware threads that share the same core when capacity exceptions are detected.

Besides reducing aborts due to conflicts on data items, SEER achieves also a drastic reduction of the frequency of activation of the pessimistic software fall-back path of the HTM system. In fact, in order to ensure the eventual success of transactions that may fail deterministically using HTM, after a limited number of attempts using hardware transactions, transactions are executed pessimistically using a fall-back path that uses a software-based synchronization mechanism — typically, a single-global lock [27, 18]. By reducing the number of retries necessary to commit a transaction, our proposal also contributes to reducing the frequency of activation of the software fall-back, whose sequential nature is known to hamper HTM performance [27, 18, 15]. Overall, our experimental study shows that, by applying SEER to standard TM benchmarks, one can obtain gains up to 2.5× and average speed-ups of 62% at 8 threads.

This paper is organized as follows. Section 2 surveys the state of the art in TM schedulers and identifies the key factors that make our proposal novel. Then, in Section 3, we provide an overview of our solution. We present the details of our SEER implementation in Section 4. Finally, we evaluate our proposal in Section 5 and conclude in Section 6.

## 2. RELATED WORK

Roughly speaking, the objective of a TM scheduler is to decide when it is best to execute a transaction, possibly deciding to serialize concurrent transactions based on their likelihood of contending with each other, with the ultimate goal of maximizing performance (typically throughput).

Most of the existing schedulers target STM systems, which are assumed to be able to provide precise information on the conflicts that caused the abort of a transaction. This is the case for CAR-STM [11] and Steal-On-Abort [3], where there are $N$ serialization queues (one for each thread), and an aborting transaction $T_i$ is placed in the queue of $T_j$ that caused its abort. The idea is that $T_i$ is serialized after $T_j$ because it shall be executed by the thread currently running $T_j$, with which it conflicted. Both these schedulers were proposed in the scope of STMs, which were extended to obtain the required precise information on aborts.

Steal-On-Abort, although initially implemented in software, was later also proposed for an HTM simulator [2]. However, this work assumed hardware extensions to support enqueuing the serialized transactions in each core of the processor. The current expectation is that manufacturers, such as Intel and IBM, will be quite resistant to changes in the hardware due to its complexity [18] . Hence, it is particularly relevant to devise a scheduling solution for *current* HTMs: one that operates in absence of accurate information on the conflict patterns among transactions, like SEER does.

More recently, ProPS [24] followed a similar approach to the ones above but, instead, focused on long running transactions: each abort event is used to accumulate a contention probability between every pair of transaction types (i.e., atomic blocks); whenever a transaction $T$ is about to start, it may have to wait in case there is an atomic block being executed in a concurrent transaction that is expected to conflict with $T$ with high probability. This approach also requires precise information to guide the scheduling decision, which is not the case for HTMs such as Intel's TSX. Shrink [13] acts in a similar way to ProPS, but it is additionally fed with past history of transactions' read- and write-sets: assuming there is some data accesses locality between transactions' restarts, the scheduler uses this information to predict conflicts that would happen if the transaction were allowed to run against current concurrent transactions. Such fine-grained information is not available in HTMs, and could only be made available via additional software instrumentation, yielding considerable overheads.

TxLinux [25] and SER [19] both changed the Linux scheduler to be transaction-aware, the difference being that the former was integrated in a simulated HTM called MetaTM and the latter was fully in software. Similarly to the other works, these proposals also require precise information.

Contrarily the schedulers above, ATS [26] is the only solution that works with imprecise information, i.e., coping with the lack of knowledge on which pairs of transactions conflict during their execution. ATS maintains a contention factor in each thread, updated when transactions abort and commit, such that a single lock is acquired when contention exceeds a specified threshold. This simple approach is agnostic of the atomic blocks being executed, as the whole problem is subsumed by a single contention factor. The positive side is that it works with currently available HTMs. In fact, this is the *de facto* technique used with commodity HTMs due to their best-effort nature: because no transaction is guar-

| Scheduler | SW | HW | Imprecise Information | Fine-Grained |
|---|---|---|---|---|
| ATS [26] | ✓ | ✓ | ✓ | $\chi$ |
| CAR-STM [11] | ✓ | $\chi$ | $\chi$ | ✓ |
| Shrink [13] | ✓ | $\chi$ | $\chi$ | ✓ |
| ProPS [24] | ✓ | $\chi$ | $\chi$ | ✓ |
| SER [19] | ✓ | $\chi$ | $\chi$ | ✓ |
| TxLinux [25] | $\chi$ | ✓ | $\chi$ | ✓ |
| SOA [3, 2] | ✓ | ✓ | $\chi$ | ✓ |
| **Seer** | $\chi$ | ✔ | ✔ | ✔ |

**Table 1: Comparison of TM schedulers in terms of: regulating an STM and/or HTM, working without precise information on which transaction caused the abort, and whether it uses multiple fine-grained locks to schedule transactions' execution. Seer, our proposal, is the only scheduler that provides all the following properties: 1) works with HTM; 2) does not require precise feedback on aborts; and 3) and adopts a fine-grained serialization mechanism.**

anteed to commit, a software fall-back must be provided to ensure progress; the single lock fall-back that is typically used [27, 18, 20, 15] is, in essence, akin to ATS. Since ATS relies on a single contention factor and one lock for serialization, it alternates between serializing all transactions or letting them all execute concurrently; hence, that is why we characterize it as a a coarse-grained scheduler.

We summarize the above state of the art in Table 1. We can see that our contribution Seer is unique by being applicable to commodity HTMs (i.e., it works with imprecise input) and allowing to serialize multiple transactions concurrently in a fine-grained manner (i.e., it does not have a single lock for serialization, as ATS does).

Finally, recent works [8, 4] have investigated the use of online profiling and optimization techniques in a similar spirit to what Seer does, but for a different, complementary purpose: decide the best software fall-back and retry policies.

# 3. OVERVIEW OF THE SOLUTION

Schedulers for TM systems, independently of their software or hardware nature, benefit particularly from the availability of fine-grained precise information about what causes the abort of a transaction. This means that if we are running a transaction for an atomic block of our program, and we know that it aborted due to a concurrent transaction executing another specific atomic block, then it is best to schedule them in a way that prevents their concurrent execution.

Having access to such information is typically trivial in STMs. However, mainstream HTMs provide little to no feedback with respect to this matter. In particular for Intel TSX (and also for IBM's HTMs), upon the abort of a hardware transaction, it is possible to know only a rough categorization: for instance, whether it was a data conflict; or whether the space available for the read- or write-set buffers in the hardware caches was exhausted; or whether there was an interrupt that caused a context switch or a ring transition. As such, no information is given about which transaction was the cause for the abort. This is the challenge that prevents existing schedulers from being effectively applicable to existing HTMs.

The high level idea of our solution is to take a probabilistic approach. While we do not know what exactly causes a transaction $T_i$ to abort, because the HTM provides no such information, we can try to infer the answer by observing enough times which transactions were active when $T_i$ aborted. By repeating this observation over time, we can gather probabilistic knowledge on the likelihood of conflicts between pairs of transactions. This knowledge can then be exploited to decide, when a transaction starts, whether to schedule it or not depending on the conflict probabilities with the currently active transactions.

The probabilistic inference mechanism of Seer is based on three key ideas: (1) we continuously collect on-line information about the transactions concurrently active upon commit and abort events, by means of a lightweight, synchronization-free monitoring mechanism; (2) we periodically analyze this information and estimate probabilities of aborting/committing in the presence of other specific transactions; and (3) this information is used to periodically devise a fine-grained locking-scheme, whose locks are acquired upon the start of a transaction and allow for serializing the execution of conflict prone-pairs of transactions (without blocking other transactions not likely to incur any conflict).

Figure 2 portraits the life-cycle of transactions within our scheduler. The objective of this life-cycle is to populate a global table that reifies the automatically inferred locking scheme. Seer uses one lock for each transaction in the target application (identified in the columns). Each row $i$ of the table specifies the locks that transaction $T_i$ should acquire, indicating that $T_i$ conflicts often with the transactions associated with these locks, and that these transactions should not be executed concurrently. We associate each atomic block in the application source code to a different transaction $T_i$: this way, we seek to serialize transactions with a fine granularity, contrarily to other approaches that work with HTM and that use a single lock for serialization [26].

To understand how to reach that objective, we begin by describing the life-cycle of Seer. In step ①, a transaction $T_3$ is about to be executed on core $C_1$. Before doing so, it acquires the locks defined by Seer in a global table: in this case, lock $L_1$. Then, it announces that $C_1$ is executing $T_3$ in the list of active transactions in step ②. Step ⑦ shows that transactions are removed from that list when they are finished.

By acquiring lock $L_1$ in the lock table, this means that transaction $T_3$ was deemed to contend with $T_1$. Although instances of $T_1$ do not acquire lock $L_1$, because they do not contend with 'themselves', they do co-operate with contending transactions (such as $T_2$ and $T_3$) by waiting for their completion, before starting executing, if lock $L_1$ is found to be taken. In general:

1. A transaction $T_x$ **waits** for its lock $L_x$ to be free before proceeding, which serves to respect our scheduling policy.

2. It **locks** $L_y$ if it contends with $T_y$ (we allow $x = y$, in which case $T_x$ contends with instances of itself).

We are left with describing how the locking scheme is generated. Step ③ illustrates that, upon a commit or abort of a transaction $T_n$ running on core $C_n$, the active transactions list is scanned and the transactions found there are incremented in two per-thread matrices, namely commitStats and abortStats, which are stored as thread-local variables (step ④). An entry $x, y$ in commitStats (resp. abortStats) tracks

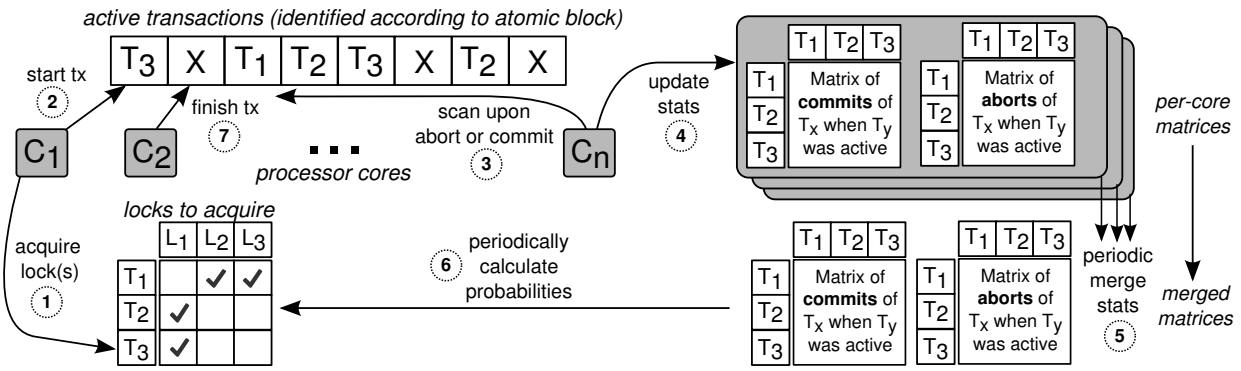


**Figure 2: Overview of Seer. The idea is to assess the probability of conflicts between transactions, without requiring precise information by the HTM. To do so, transactions are announced right before they are executed on a given core, and then this information is scanned upon commits and aborts, to compensate for the lack of feedback from the HTM. While not totally accurate, this information allows to probabilistically infer the relevant conflict patterns among transactions over time, and then to produce a dynamic locking scheme that serves to schedule transactions (by preventing some transactions from running concurrently).**

the frequency of commit (resp. abort) events for transaction $T_x$, in which $T_y$ was found to be running (in the active transactions list), after the commit (resp. abort) of $T_x$ Consider for instance that $T_1$ only conflicts often with $T_3$. Such fact is unknown beforehand and our approach aims to infer it in run-time: as we gather statistics, over time, recurrent events emerge and become identifiable using probabilistic inference.

Periodically, these statistics are merged, across all per-core's matrices, into two global matrices in step ⑤. These are used to calculate and update the locking scheme to reduce aborts of transactions. The intuition is to use the information about how often $T_x$ committed and aborted in the presence of each different transaction. The challenge in doing so is to identify, among all captured conflicts, which ones occur frequently enough to benefit from throttling down concurrency. The ability to extract these decisions using solely the imprecise information provided by commodity HTMs is what makes Seer novel with respect to other schedulers.

As a result, we are able to periodically generate a dynamic locking scheme, as depicted in step ⑥. As explained above, these locks are used to serialize transactions with a fine granularity. This is a key feature that allows Seer to yield substantial performance improvements as we later show in Section 5. Another noteworthy feature of Seer is that it works in a completely transparent fashion to the programmer. We require only minimalist compiler support, by enumerating the atomic blocks in the program, and passing their unique identifier (one per source code atomic block) into the TM library calls. The scheduler itself is implemented in the TM library that regulates the software fall-back management. Further, Seer fully automates the tuning of internal parameters in the probabilistic inference, via a self-optimization mechanism that is driven by the feedback gathered at run-time on the throughput of the TM system.

Finally, Seer introduces the abstraction of *core locks*, i.e., locks that prevent the concurrent execution of multiple hardware threads on the same physical core. The idea of core locks is based on the observation that, in workloads characterized by frequent transactions with non-minimal memory footprints, the likelihood of capacity aborts in the HTM is exacerbated when multiple threads are allowed to execute freely, as they contend for the shared caches of the core.

| Variable | Description |
|---|---|
| thread | Per-thread structure to hold metadata during the execution of a transaction. |
| sgl | Single-global lock used in the software fall-back path of the HTM. |
| activeTxs | Global array where threads announce the transactions they are executing. |
| commitStats | Global matrix where, each line for transaction $T_i$, reports the transactions that were concurrently running whenever $T_i$ committed. This matrix is periodically built by summing the per-core equivalent matrices kept in each thread variable. |
| abortStats | Similar to commitStats, but for abort statistics. |
| executions | Array with total number of executions (commits and aborts) of each transaction. |
| locksToAcquire | Global matrix where each line corresponds to a transaction and the columns define the locks that should be acquired for the transaction according to Seer. |
| txLocks | Global array of locks, one per transaction (i.e., atomic block) of the program. |
| coreLocks | Global array of locks, one per core of the processor. |

**Table 2: Characterization of the data-structures used in Seer. Some of these are visible in the high-level overview in Figure 2, whereas the rest is used in Algorithms 1-5.**

# 4. DETAILED ALGORITHM

We now present the detailed description of SEER. We report the data-structures used by SEER in Table 2, most of which were already presented in the overview.

***Conventional HTM usage.*** We start by describing the basic software mechanisms that govern HTM transactions and the fall-back path, in which SEER is implanted. We highlight lines associated with the conventional HTM mechanisms with a △ (other lines belong to SEER). We begin with the START procedure, in Alg. 1, where a transaction *txId* is initiated by a given *thread*.

The START procedure implements a retry loop to try to execute a hardware transaction, up to some threshold (`MAX_ATTEMPTS`), resorting to a fall-back path in case the threshold is reached (in line 20). Note that the function to begin a hardware transaction, *_xbegin()* (in line 9), returns a status that normally represents that the transaction has started, i.e., the predicate in line 10 evaluates to true. Otherwise, this status indicates a coarse categorization of the abort. Note that an aborted hardware transaction transparently jumps back, and returns from this function, akin to the *setjmp/longjmp* mechanism used in C/C++.

***Seer Algorithm.*** We now discuss the various mechanisms that augment this conventional procedure: i) transactions are announced to other cores (see line 5), ii) aborts are registered in the per-core statistics (see line 16), and iii) locks are used to induce fine-grained serialization between contending transactions (see lines 8 and 23). We present each part next.

The END procedure is presented in Alg. 2 where we finish the hardware transaction, or release the global lock, depending on the path taken in START. In case the transaction was successfully committed via a hardware transaction, we add this information to our per-core statistics in line 28, and possibly release locks acquired by our scheduler in line 29. Finally, we remove the transaction from the activeTxs list.

The procedures for registering aborts and commits are shown in Alg. 3. The idea is to scan the activeTxs list and to increase the frequency of the transactions found there, in the row corresponding to the transaction that has aborted/committed (identified by txId). This is the mechanism that we use to infer information about conflicts, and to compensate for the lack of feedback from the HTM about the pairs of conflicting transactions. In general, this collection of statistics may not be completely accurate, and could suffer of both false positives and false negatives. SEER copes with this uncertainty using probabilistic inference techniques, whose details we shall discuss shortly.

Notice that the aforementioned statistics are maintained per-core, i.e., in a private fashion. Furthermore, the activeTxs list ends up being a set of single-writer multi-reader registers; we do not place any synchronization when accessing the list, with the intent of keeping it lightweight.

The procedures for lock management, according to our scheduler, are defined in Alg. 4. We use two types of locks:

1. **txLocks:** one per transaction of the application, to serialize contending transactions according to the probabilities (line 48) that we describe later (in Alg. 5). Our scheduler may dictate that a transaction acquires some of these locks only when the transaction has spent most of its attempts in hardware transactions — it has one left — as a last resort measure to obtain progress before triggering the global lock in the fall-back.

2. **coreLocks:** one per physical core of the processor, to reduce capacity aborts, which are amplified due to hardware threads that share the private caches of a physical core. These caches are small and limit the size of hardware transactions, more so if shared among several. Hence, we acquire the coreLock when a capacity abort is detected (line 45).

Furthermore, we also introduce a contention avoidance technique, which imposes waiting before starting a transac-

**Algorithm 1** SEER algorithm.

```
1: START(thread, txId)
2:  thread.core ← current-core()  ▷ thread is bound to core
3:  thread.acquiredTxLocks ← false
4:  thread.acquiredCoreLock ← false
5:  activeTxs[thread.core] ← txId
6△  attempts ← MAX_ATTEMPTS
7△  begin:      ▷ used to jump to and re-attempt with HTM
8:  WAIT-SEER-LOCKS(thread, txId)
9△  htmStatus ← _xbegin()
10△ if htmStatus = _XBEGIN_STARTED
11△   if is-locked(sgl) ▷ ensure correctness with fall-back
12△     _xabort()
13△   else
14△     return     ▷ hw transaction enabled, proceed to tx
15△ ▷ hw transaction aborted, handle before restarting
16: REGISTER-ABORT(thread, txId)
17△ attempts ← attempts - 1
18△ if attempts = 0  ▷ give up on HTM, fall-back to lock
19:   RELEASE-SEER-LOCKS(thread, txId)
20△   acquire-lock(sgl)   ▷ SW fall-back with a single lock
21△   return      ▷ SW fall-back path taken, proceed to tx
22△ ▷ before re-attempting, trigger our scheduler SEER
23: ACQUIRE-SEER-LOCKS(thread, txId, htmStatus)
24△ goto begin
```

**Algorithm 2** SEER algorithm.

```
25: END(thread, txId)
26△ if _xtest()  ▷ returns true if inside a HW transaction
27△   _xend()        ▷ tries to commit the HW transaction
28:   REGISTER-COMMIT(thread, txId)
29:   RELEASE-SEER-LOCKS(thread, txId)
30△ else
31△   release-lock(sgl)  ▷ executed with lock-based fall-back
32: activeTxs[thread.core] ← ⊥
```

**Algorithm 3** SEER algorithm.

```
33: REGISTER-ABORT(thread, txId)
34:  thread.executions[txId]++
35:  for all i = 0 until activeTxs.length
36:    if i ≠ thread.core ∧ activeTxs[i] ≠ ⊥
37:      thread.abortStats[txId][activeTxs[i]]++

38: REGISTER-COMMIT(thread, txId)
39:  thread.executions[txId]++
40:  for all i = 0 until activeTxs.length
41:    if i ≠ thread.core ∧ activeTxs[i] ≠ ⊥
42:      thread.commitStats[txId][activeTxs[i]]++
```

**Algorithm 4** SEER algorithm.

```
43: ACQUIRE-SEER-LOCKS(thread, txId, htmStatus)
44:  if htmStatus & _XABORT_CAPACITY ∧ ¬thread.acquiredCoreLock
45:    acquire-lock(coreLocks[thread.core % PHYSICAL_CORES]) ▷ adapted to the topology of hyper-threads in Intel processors
46:    thread.acquiredCoreLock ← true
47:  if attempts = 1
48:    ACQUIRE-TX-LOCKS(txId)                                ▷ acquire locks specified in row locksToAcquire[txId]
49:    thread.acquiredTxLocks ← true

50: WAIT-SEER-LOCKS(thread, txId)
51:  if is-locked(sgl)                                       ▷ avoid starting hardware transactions if the fall-back is in use
52:    if thread.core = 0                                    ▷ only one thread updates the serialization locks
53:      UPDATE-SEER-LOCKS()                                 ▷ exploit the wait time to run SEER
54:      if enough-samples() then stochastic-hill-climbing(Th1, Th2)   ▷ periodically adapt the parameters used in Alg 5
55:    wait while is-locked(sgl)                             ▷ wait here instead of aborting in line 12
56: ▷ if some other thread is owning these SEER locks, cooperate with it and wait
57:  wait while ¬thread.acquiredTxLocks ∧ is-locked(txLocks[txId])
58:  wait while ¬thread.acquiredCoreLock ∧ is-locked(coreLocks[thread.core])

59: RELEASE-SEER-LOCKS(thread, txId)
60:  if thread.acquiredTxLocks
61:    RELEASE-TX-LOCKS(txId)
62:  if thread.acquiredCoreLock
63:    release-lock(coreLocks[thread.core])
```

**Algorithm 5** SEER algorithm.

```
65: UPDATE-SEER-LOCKS()
66:  for all x ∈ A                                           ▷ A is the set of txs in the application source code
67:    η ←avg({P(x aborts | x||y), ∀y∈ A})
68:    σ² ←var({P(x aborts | x||y), ∀y∈ A})
69:    for all y ∈ A                                         ▷ determine if y is likely to contend with x
70:      ▷ 1st condition checks whether abort events of x, in which y is seen running concurrently, are common enough
71:      ▷ 2nd condition checks if y is among the txs that, when executed concurrently with x, most likely contend with x
72:      if (P(x aborts ∩ x||y) > Th1 ∧ P(x aborts | x||y)> Th2-th percentile of a Gaussian N(η,σ²) )
73:        locksToAcquire[x] ← y                             ▷ contending txs take each other's locks when they abort
74:        locksToAcquire[y] ← x                             ▷ recall that a tx also waits for its own tx-lock to be free (line 57)
75: ▷ sort all locks in each row of locksToAcquire, and swap the old matrix by the new one (using an indirection pointer)
```

tion (in line 8). This is presented in WAIT-SEER-LOCKS, in Alg. 4, where there are two main ideas. First, we use a known technique to avoid the lemming effect [6]. The problem is that hardware transactions quickly exhaust their budget of attempts when the fall-back lock is taken and tend to execute mostly in the fall-back as a consequence. To reduce this chance, a transaction waits if the global lock is taken, as otherwise it would likely abort in line 12.

The second idea behind WAIT-SEER-LOCKS is to also wait in case the txLock and/or coreLock are taken by another thread (lines 57 and 58). The intuition is that, even though this thread may not have had aborts that lead it to acquire locks, it is beneficial if it co-operates with concurrent threads that have taken the locks, giving them a chance to complete without conflicting. Doing so is instrumental for the meaningfulness of the locking scheme that we present next while avoiding a transaction to having to pessimistically always acquire the lock of its transaction.

We also opportunistically take the chance to update the locking scheme of SEER in line 53, instead of having the thread waiting idle for the global lock to be released. We specifically do this in one designated thread to avoid synchronization. Furthermore, we have an active transactions list with as many slots as threads in the program, making each entry of the list a single-writer multi-reader register.

The procedure to acquire the transaction locks simply goes over the row *locksToAcquire[txId]* and acquires each lock. All rows are sorted consistently by the periodic update, hence this procedure acquires them in that order to avoid deadlocks. We also optimize this procedure to acquire the locks with a hardware transaction when there are two or more locks, instead of performing multiple compare-and-swap operations (CAS) to acquire all locks. The rationale of this optimization is to batch the synchronization of two or more CASes into a single TSX hardware transaction. If the transaction is not successful, we fall-back to the normal acquisition. Note that this is not lock elision [23]; we are effectively using TSX as a multi-CAS, not eliding the locks acquired.

***Devising the Locking Scheme.*** We are left with the logic for updating the locking scheme for fine-grained serialization of transactions in SEER, which we present in Alg. 5. This procedure, opportunistically invoked by one thread, starts by summing the commit and abort per-core statistics.

For all transactions in the application, we consider a pair $x, y$ at a time, and calculate the conditional probability of $x$ aborting, *given* that $y$ was running concurrently with it, $P(\text{x aborts} \mid x \parallel y)$, and the conjunctive probability of $x$ aborting *and* $y$ running concurrently, $P(\text{x aborts} \cap x \parallel y)$:

$$P(\text{x aborts} \mid x \parallel y) = \frac{a_{x,y}}{c_{x,y} + a_{x,y}}$$

$$\begin{aligned} P(\text{x aborts} \cap x \parallel y) &= P(\text{x aborts} \mid x \parallel y) \times P(x \parallel y)) \\ &= \frac{a_{x,y}}{c_{x,y} + a_{x,y}} \times \frac{c_{x,y} + a_{x,y}}{e_x} = \frac{a_{x,y}}{e_x} \end{aligned}$$

where we abbreviated commitStats[$x$][$y$] to $c_{x,y}$, abortStats[$x$][$y$] to $a_{x,y}$ and executions[$x$] to $e_x$. These two probabilities can be efficiently calculated with the statistics that are at our disposal, and are used to define two thresholds, $\mathcal{T}h_1$ and $\mathcal{T}h_2$, aimed at pursuing different goals.

The threshold $\mathcal{T}h_1$ establishes a lower bound on the probability $P(\text{x aborts} \cap x \parallel y)$, below whose value Seer avoids serializing transactions $x$ and $y$. Low values of this probability imply that the frequency of aborts events of $x$, in which $y$ was found to run concurrently with it, are rare. It is hence beneficial to avoid the cost of restricting concurrency and sparing the costs of additional lock acquisitions.

The threshold $\mathcal{T}h_2$ is instead used to establish a cut-off on the probability distribution of $P(\text{x aborts} \mid x \parallel y)$ (henceforth abbreviated as $\mathcal{P}_{x,y}$), which aims at determining which subset $\mathcal{S}$, of the set of transactions $y$, suspected to conflict $x$, should be prevented from running in parallel with $x$.

More in detail, Seer includes in $\mathcal{S}$ only the transactions y whose probability $\mathcal{P}_{x,y}$ is larger than $\mathcal{T}h_2$-th percentile of a Gaussian distribution $\mathcal{N}(\eta, \sigma^2)$ with mean $\eta$ and variance $\sigma$ equal, respectively, to the mean and variance of the values of $\mathcal{P}_{x,y}$ (for all possible values of $y$). The rationale here is that a transaction $y'$ that is wrongly suspected of conflicting with $x$ (due to false positives while probing the active transactions) will have significantly lower values of $\mathcal{P}_{x,y'}$, with respect to a transaction $y''$ that conflicts with $x$ often. Hence, for such transactions $y''$, $\mathcal{P}_{x,y''}$ will fall in the tail of the cumulative distribution function of probabilities, which we fit with a Gaussian distribution having equivalent mean and variance.

Using the conditional probability $P(\text{x aborts} \mid x \parallel y)$ (differently from the case of $\mathcal{T}h_1$, in which we rely on the conjunctive probability $P(\text{x aborts} \cap x \parallel y)$) is aimed at factoring out, in the inference process, the cases in which $x$ and $y$ are not concurrent. This allows for focusing the analysis solely on the available evidences that support the hypothesis of a cause-effect relation between the concurrent execution of $x$ with $y$ and the abort of $x$, and for separating falsely suspected pairs of transactions from actually conflicting ones more reliably than if one used the conjunctive probability.

Summarizing: if both conditions in line 72 are met, meaning that $x$ is deemed to abort too often because of $y$, Seer requires that transactions $x$ and $y$ have to acquire each other's lock (recall that we associate on lock per transaction).

Finally, Seer relies on an on-line self-tuning mechanism that automates the identification of the values of the thresholds $\mathcal{T}h_1$ and $\mathcal{T}h_2$, hence sparing users from the burden of identifying statically defined values that may be sub-optimal in heterogeneous, or time varying, workloads. To this end, Seer uses a simple and lightweight bi-dimensional stochastic hill-climbing search, which exploits the feedback of the TM performance (throughput obtained via RTDSC-based measurements) to guide the search in the parameter's space [0,1]×[0,1] for the thresholds $\mathcal{T}h_1$ and $\mathcal{T}h_2$. Our hill-climbing is stochastic in the sense that, with a small probability $p$, it performs random jumps in the parameters' space to avoid getting stuck in local minima. We configured this self-tuning mechanism with standard values that were applied to irregular concurrent applications such as those used with TM [8]. Specifically, we set $p$ to 0.1% and the initial values of $\mathcal{T}h_1 = 0.3$ and $\mathcal{T}h_2 = 0.8$.

# 5. EVALUATION

To evaluate our proposal, we formulate several questions and experiments. First, in Section 5.1, we compare Seer with the available alternatives for HTM. Next, in Section 5.2, we assess how often hardware transactions are successful and to what extent locks are acquired. Finally, in Section 5.3, we seek to understand the merit of each design choice of Seer and its overheads.

All the experimental results were obtained using a TSX-enabled Intel Haswell Xeon E3-1275 processor with 32GB RAM and 8 virtual cores (4 physical, each one running up to 2 hardware threads). We ran our experiments in a dedicated machine running Ubuntu 12.04, and the results reported are the average of 20 runs. Our evaluation uses the standard STAMP suite, a popular set of benchmarks for TM [21], encompassing applications representative of various domains that generate heterogeneous workload. We excluded Bayes given its non-deterministic executions, and Labyrinth as most of its transactions exceed TSX capacity.

## 5.1 How much can we gain with Seer?

To assess the benefits of Seer we consider 3 alternatives:

1. **HLE** where transactions may be retried a small number of times (processor implementation-dependent), but without any scheduling or contention management, which may cause the lemming effect [6] on the elided lock[1].

2. **RTM** where the retry logic is controlled in software and hence we retry a given number of attempts in HTM and always wait before doing so if the single-global lock is taken. As already discussed in Section 2, the usage of a single lock in the fall-back path of these two baseline mechanisms makes them analogous in spirit to the ATS scheduler [26].

3. **SCM** where we implemented the Software-assisted Conflict Management [1] technique. SCM uses an auxiliary lock to serialize transactions that are aborted, thus decreasing the chance of having the lemming effect, where failed hardware transactions keep exhausting the attempts and fall-back to the single-global lock.

We used a budget of 5 attempts for hardware transactions in all approaches (as used by Intel for this set of benchmarks [27]). We present the results for these approaches together with Seer in Figure 3. The speedups are relative to a sequential non-instrumented execution. In general we can see that Seer performs similarly to the best solution up to 3 threads, and better with 4 or more threads. This is a consequence of having more opportunity for our fine-grained scheduling to shine when there is more concurrency.

Figure 3i shows the geometric mean speedup across all the benchmarks, where we can see that Seer yields 62% improvement over RTM and SCM with 8 threads, with peak

[1]STAMP benchmarks are executed as having 1 lock to elide.

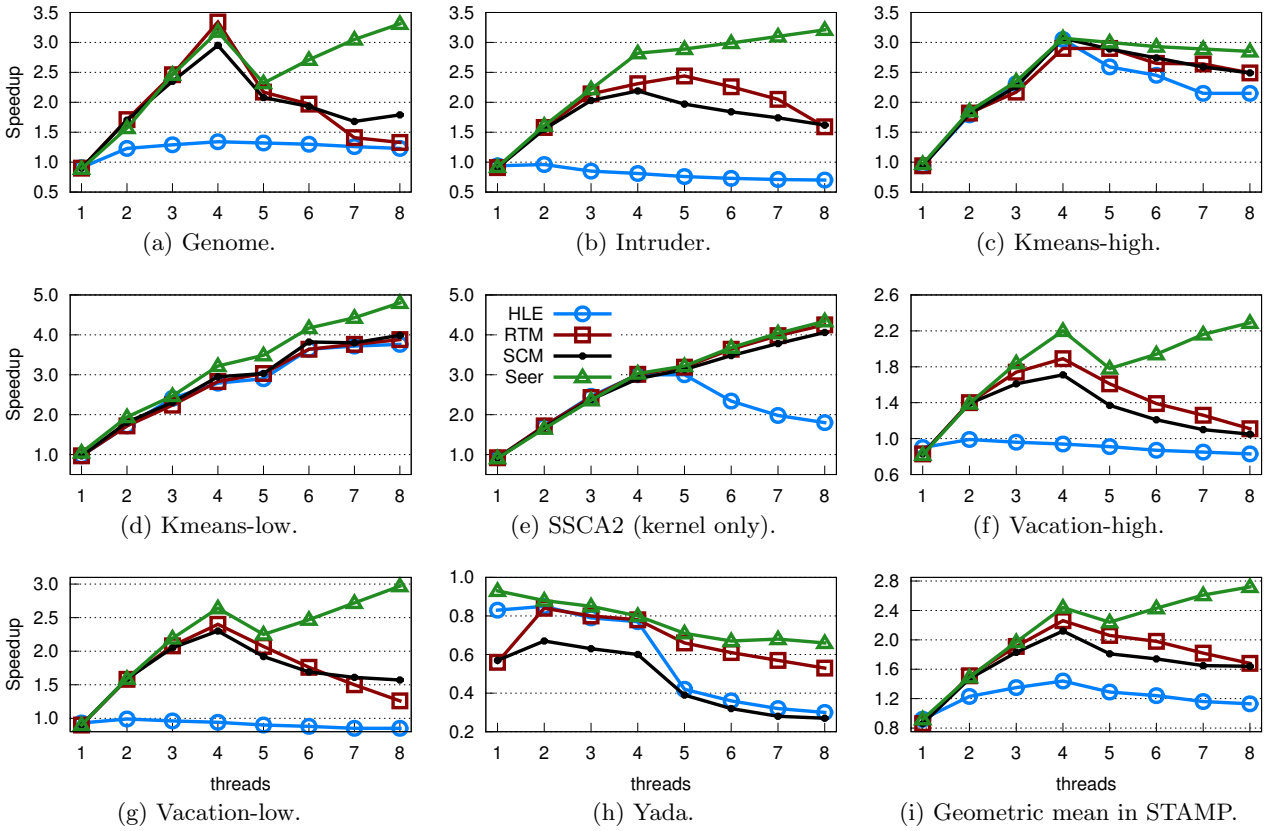


**Figure 3: Speedup of different HTM based approaches across STAMP benchmarks.**

gains of approximately $2-2.5\times$ over the best performing alternative baseline in benchmarks such as Genome, Intruder and Vacation.

## 5.2 Where are the gains of Seer coming from?

HTM performance is known to be strongly affected by the likelihood with which transactions resort to acquiring the global lock [10]. Furthermore, even in cases in which hardware transactions are used successfully, parallelism may be overly restricted by the usage of the auxiliary lock in the case of SCM, or of fine-grained locks in the case of Seer.

Table 3 provides a breakdown of the usage of locks for each considered approach, shedding lights on the reasons underlying the performance gains achieved by Seer. The reported results are averaged across all benchmarks.

As expected, HLE drastically loses its ability to execute transactions in hardware, as threads increase, because it suffers from the lemming effect and at higher concurrency degrees most transactions use the single-global lock. RTM improves over this scenario but still uses hardware transactions only in 63% of the executions at 8 threads. The SCM approach has significantly lower usage of the fall-back path (up to 5%). However, there are up to 29% hardware transactions that execute under the auxiliary lock. We highlight that this is a single lock, which prevents parallelism among all restarting transactions. This is why SCM is unable to provide noticeable speedups over RTM in practice, as shown in the previous section.

**Table 3: Breakdown of percentage (%) of types of transactions used in average across STAMP.**

| Variant | Transaction Mode | 2t | 4t | 6t | 8t |
|---|---|---|---|---|---|
| HLE | HTM no locks | 75 | 52 | 39 | 23 |
| | SGL fall-back | 25 | 48 | 61 | 77 |
| RTM | HTM no locks | 94 | 82 | 76 | 63 |
| | SGL fall-back | 6 | 18 | 24 | 37 |
| SCM | HTM no locks | 90 | 83 | 77 | 66 |
| | HTM + Aux lock | 8 | 15 | 20 | 29 |
| | SGL fall-back | 2 | 2 | 3 | 5 |
| Seer | HTM no locks | 94 | 94 | 85 | 80 |
| | HTM + Tx Locks | 2 | 2 | 2 | 3 |
| | HTM + Core Locks | 3 | 1 | 4 | 4 |
| | HTM + Tx + Core Locks | 1 | 2 | 8 | 12 |
| | SGL fall-back | 0 | 1 | 1 | 1 |

Finally, Seer is able to improve over all previously described alternatives, exactly because the frequency with which it uses a single-global lock is drastically lower (around 1%), and the other locks that it exploits have a much finer granularity — one per transaction and one per core — that allow to cope with conflict dependencies and cache capacity exceptions (in case multiple hardware threads share the same cache) without serializing *every* active transaction. In fact, in 50% of the cases in which *some* transaction lock is acquired by Seer, the fraction of transaction locks that are actually acquired is lower than 23% of the globally avail-

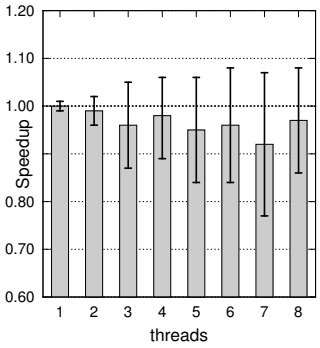


**Figure 4: Overhead of SEER when profiling and calculating locks to acquire.**

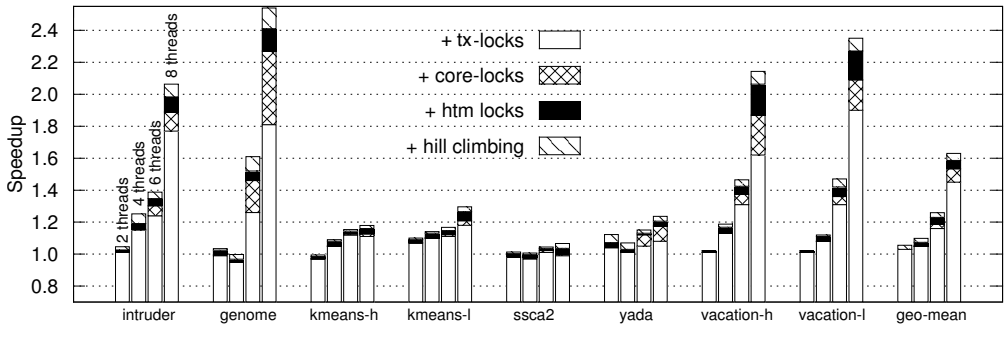


**Figure 5: Cumulative contribution of each technique employed in SEER: speedups are shown relatively to SEER without any lock acquisition (but with all the profiling enabled) and the three variants are incrementally added.**

able transaction locks. This experimental result confirms the ability of the proposed lock inference mechanism to synthesize effective fine-grained locking schemes.

## 5.3 How much does each design choice contribute to Seer?

The design of SEER encompassed: 1) capturing statistics about commits, aborts, and concurrent transactions; 2) acquiring transaction locks when aborts occur; 3) acquire a core lock when a capacity abort happens; 4) acquire transaction locks with a hardware transaction to reduce the overheads of multiple compare-and-swaps; and 5) adapt the thresholds $\mathcal{T}h_1$, $\mathcal{T}h_2$ via a stochastic hill-climbing algorithm.

We first assess the overhead of the monitoring, lock-inference and self-tuning mechanisms of SEER. For this, we ran a variant of SEER that incurs the overheads of all its mechanisms, without however acquiring any lock. In Figure 4, we show the average speed-up of this SEER's variant relatively to RTM (that consistently performed second best in our evaluation in Section 5.1). These results are the geometric mean across the STAMP benchmarks, and we can see that the mean slowdown is less than 5% and varies from negligible to at most 8%. Even challenging scenarios, such as a low contention small hash-map (4k elements and 1k buckets) yielded a maximum of 4% overhead.

In fact, these overheads could be made even lower by reducing the frequency with which SEER samples statistics and updates the locking scheme, at the cost of increasing the latency for identifying an adequate scheduling strategy. Our choice of using relatively aggressive monitoring/optimizations rates is motivated by the need to ensure quick convergence times given that STAMP benchmarks have very short runs (on the order of a few seconds); yet in long running services, where convergence speed is a less critical issue, SEER may be configured to use less frequent sampling/optimization strategies in order to further reduce its overheads.

To quantify the relative relevance of each of the mechanisms integrated in SEER, we conducted a series of experiments, whose results, shown in Figure 5, evaluate the speedup of different variants of SEER. We consider as baseline, the SEER variant previously considered for the plots in Figure 4, which incurs the costs of collecting statistics and updating the locking strategy, without ever acquiring any lock. Then, we consider four improving variants where we cumulatively add the transaction locks acquisition, the core locks acquisition, the hardware transaction acquisition of locks, and the adaptation of the thresholds used.

In general the transaction locks provide the largest boost in performance as they capture the conflicts inherent to each benchmark. Unsurprisingly, the core locks are only beneficial when using 6 or 8 threads, i.e., when we start executing multiple hardware threads on the same core. The hardware lock acquisition also shows improvements with larger concurrency degrees, since these scenarios are the ones that more often trigger the necessity of acquiring locks (and, which hence trigger this optimization more frequently). Also, a similar gain is provided by adapting on-line the thresholds used in the transaction locks probabilities calculations. Finally, we experimented also by enabling only the core-locks and obtained geometric mean speedups of 9% and 22% at 6 and 8 threads. This corroborates the need for both transaction and core locks.

# 6. CONCLUSIONS AND FUTURE WORK

In this work we presented SEER, the first *fine-grained* scheduler designed to cope with specific challenges arising with HTMs. The most innovative feature of our proposal is that it can probabilistically infer conflict patterns among pairs of transactions of a TM program, without relying on the availability of precise information from the underlying TM system. Conversely, SEER relies on lightweight, yet inherently imprecise techniques, to gather information on the set of concurrently active transactions upon the commit and abort events of transactions. Probabilistic techniques are then used to filter out false positives and infer an effective dynamic locking scheme that is used to serialize contention-prone transactions in a fine-grained fashion. We evaluated our solution, SEER, against several alternatives using a mainstream HTM (Intel TSX). As a result, we obtained 62% performance improvements on average in standard benchmarks with 8 threads, with speed-ups peaking up to $2 - 2.5\times$ in complex benchmarks like Genome, Intruder and Vacation.

In the future we plan to extend SEER in several directions. This includes experimenting with probabilistic sampling techniques [5], as well as adopting even more fine-grained locking schemes, which associate locks depending on both the atomic block *and* the identifier of the data structure being manipulated in that atomic block.

***Acknowledgements:*** This work was supported by national funds through Fundação para a Ciência e Tecnologia (FCT), via the projects UID/CEC/50021/2013 and EXPL/EEI-ESS/0361/2013. We are grateful to the anonymous reviewers for their insightful feedback.

# 7. REFERENCES

[1] Y. Afek, A. Levy, and A. Morrison. Software-improved Hardware Lock Elision. In *Proc. Symposium on Principles of Distributed Computing*, PODC, pages 212–221, 2014.

[2] M. Ansari, et al. Improving Performance by Reducing Aborts in Hardware Transactional Memory. In *Proc. High Performance Embedded Architectures and Compilers*, HiPEAC, pages 35–49, 2010.

[3] M. Ansari et al. Steal-on-Abort: Improving Transactional Memory Performance Through Dynamic Transaction Reordering. In *Proc. High Performance Embedded Architectures and Compilers*, HiPEAC, pages 4–18, 2009.

[4] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *Proc. Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 188–197, 2014.

[5] D. Dice, Y. Lev, and M. Moir. Scalable statistics counters. In *Proc. Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 43–52, 2013.

[6] D. Dice et al. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proc. Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 157–168, 2009.

[7] N. Diegues and J. Cachopo. Practical Parallel Nesting for Software Transactional Memory. In *Proc. of Symposium on Distributed Computing*, DISC, pages 149–163, 2013.

[8] N. Diegues and P. Romano. Self-Tuning Intel Transactional Synchronization Extensions. In *Proc. Conference on Autonomic Computing*, ICAC, pages 209–219, 2014.

[9] N. Diegues and P. Romano. Time-warp: lightweight abort minimization in transactional memory. In *Proc. of Principles and Practice of Parallel Programming*, PPoPP, pages 167–178, 2014.

[10] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proc. Conference on Parallel Architectures and Compilation Techniques*, PACT, pages 3–14, 2014.

[11] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: Scheduling-based Collision Avoidance and Resolution for Software Transactional Memory. In *Proc. Symposium on Principles of Distributed Computing*, PODC, pages 125–134, 2008.

[12] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proc. of Conference on Programming Language Design and Implementation*, PLDI, pages 155–165, 2009.

[13] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing Versus Curing: Avoiding Conflicts in Transactional Memories. In *Proc. Symposium on Principles of Distributed Computing*, PODC, pages 7–16, 2009.

[14] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. Principles and Practice of Parallel Programming*, PPoPP, pages 237–246, 2008.

[15] T. Heber, D. Hendler, and A. Suissa. On the Impact of Serializing Contention Management on STM Performance. *Journal of Parallel and Distributed Computing*, 72(6):739–750, June 2012.

[16] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proc. Symposium on Computer Architecture*, ISCA, pages 289–300, 1993.

[17] S. Hirve, R. Palmieri, and B. Ravindran. HiperTM: High Performance, Fault-Tolerant Transactional Memory. In *Proc. Confenrece on Distributed Computing and Networking*, ICDCN, pages 181–196, 2014.

[18] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. In *Proc. Symposium on Microarchitecture*, MICRO, pages 25–36, 2012.

[19] W. Maldonado, et al. Scheduling Support for Transactional Memory Contention Management. In *Proc. Principles and Practice of Parallel Programming*, PPoPP, pages 79–90, 2010.

[20] A. Matveev and N. Shavit. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proc. Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 11–22, 2013.

[21] C. Minh et al. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. Symposium on Workload Characterization*, IISWC, pages 35–46, 2008.

[22] S. Peluso, P. Romano, and F. Quaglia. SCORe: A Scalable One-Copy Serializable Partial Replication Protocol. In *Proc. of Middleware*, pages 456–475, 2012.

[23] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. Symposium on Microarchitecture*, MICRO, pages 294–305, 2001.

[24] H. Rito and J. Cachopo. ProPS: A Progressively Pessimistic Scheduler for Software Transactional Memory. In *Proc. European Conference on Parallel Processing*, Euro-Par, pages 150–161, 2014.

[25] C. Rossbach et al. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *Proc. Symposium on Operating Systems Principles*, SOSP, pages 87–102, 2007.

[26] R. Yoo and H. Lee. Adaptive Transaction Scheduling for Transactional Memory Systems. In *Proc. Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 169–178, 2008.

[27] R. Yoo et al. Performance Evaluation of Intel; Transactional Synchronization Extensions for High-performance Computing. In *Proc. Conference on High Performance Computing, Networking, Storage and Analysis*, SC, pages 1–11, 2013.