Seer: Probabilistic Scheduling for Hardware Transactional Memory

NUNO DIEGUES, Feedzai/INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal
PAOLO ROMANO, INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal
STOYAN GARBATOV, OutSystems SA/INESC-ID/Instituto Superior Técnico, University of Lisbon, Portugal

The ubiquity of multicore processors has led programmers to write parallel and concurrent applications to take advantage of the underlying hardware and speed up their executions. In this context, Transactional Memory (TM) has emerged as a simple and effective synchronization paradigm, via the familiar abstraction of atomic transactions.

After many years of intense research, major processor manufacturers (including Intel) have recently released mainstream processors with hardware support for TM (HTM).

In this work, we study a relevant issue with great impact on the performance of HTM. Due to the optimistic and inherently limited nature of HTM, transactions may have to be aborted and restarted numerous times, without any progress guarantee. As a result, it is up to the software library that regulates the HTM usage to ensure progress and optimize performance. Transaction scheduling is probably one of the most well-studied and effective techniques to achieve these goals.

However, these recent mainstream HTMs have some technical limitations that prevent the adoption of known scheduling techniques: unlike software implementations of TM used in the past, existing HTMs provide limited or no information on which memory regions or contending transactions caused the abort.

To address this crucial issue for HTMs, we propose Seer, a software scheduler that addresses precisely this restriction of HTM by leveraging on an online probabilistic inference technique that identifies the most likely conflict relations and establishes a dynamic locking scheme to serialize transactions in a fine-grained manner. The key idea of our solution is to constrain the portions of parallelism that are affecting negatively the whole system. As a result, this not only prevents performance reduction but also in fact unveils further scalability and performance for HTM. Via an extensive evaluation study, we show that Seer improves the performance of the Intel's HTM by up to 3.6×, and by 65% on average across all concurrency degrees and benchmarks on a large processor with 28 cores.

Categories and Subject Descriptors: D.1.3 [**Software**]: Programming Techniques—*Concurrent programming*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Hardware transactional memory, best-effort, scheduling

This work was supported by Portuguese funds through Fundação para a Ciência e Tecnologia via projects UID/CEC/50021/2013 and PTDC/EEISCR/1743/2014.

Authors' addresses: N. Diegues, Feedzai, Avenida D. Joao II, Lote 1.16.01, Piso 11, 1990-083 Lisboa; email: nuno.diegues@feedzai.com; P. Romano and S. Garbatov, INESC-ID Lisboa, Rua Alves Redol 9, 1000-029 Lisbon, Portugal; emails: romano@inesc-id.pt, stoyan.garbatov@ist.utl.pt.

https://doi.org/10.1145/3132036

ACM Reference format:
Nuno Diegues, Paolo Romano, and Stoyan Garbatov. 2017. Seer: Probabilistic Scheduling for Hardware Transactional Memory. *ACM Trans. Comput. Syst.* 35, 3, Article 7 (November 2017), 41 pages.
https://doi.org/10.1145/3132036

1 INTRODUCTION

The evolution of processors changed dramatically in the early 2000s, thanks to the advent of multicore architectures in mainstream computing platforms. These days we find multicore processors in almost every machine and device, making it a ubiquitous technology, with which software is still catching up.

To fully realize the potential of multicore processors, programmers are now obliged to write concurrent applications, and thus to tackle the issue of how to synchronize concurrent accesses to shared data. This challenge of correctly and efficiently synchronizing code is at the heart of many domains of applications, ranging from search engines and source control systems to data stores.

The paradigm of Transactional Memory (TM) (Herlihy and Moss 1993) was proposed exactly to address this challenge. Its premise is twofold: to enable easy development of concurrent applications while at the same time unfolding the potential to explore as much parallelism as possible in the workload and improve performance. The former is possible thanks to the adoption of the familiar abstraction of transactions to delimit portions of code that should execute atomically, that is, appear to the programmer as a single, instantaneously executable instruction (Harris et al. 2008; Michael 2013). The latter is enabled by speculatively executing transactions in parallel and relying on optimistic concurrency control techniques that transparently abort and restart transactions that develop any data conflict (Fraser and Harris 2007; Gramoli and Guerraoui 2014).

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 identifying which portions of the code have to execute atomically, and not *how* atomicity should be achieved. This approach has been shown to significantly ease the development of concurrent applications (Lupei et al. 2010; Pankratius and Adl-Tabatabai 2014; Rossbach et al. 2010).

After a decade of intense research, the relevance of TM in the industry has been recently amplified by the achievement of two main milestones: on the one hand, the standardization of programming constructs for TM in popular programming languages (such as C/C++ (Ni et al. 2008)), and on the other hand, the launch of mainstream processors with hardware support for TM (namely, by Intel (Yoo et al. 2013) and IBM (Adir et al. 2014; Cain et al. 2013; Nakaike et al. 2015; Wang et al. 2012)). These Hardware Transactional Memories (HTMs), which provide minimal support to help implement the TM abstraction, are now available in millions of machines across the globe.

1.1 Implementing the TM Abstraction with Hardware Support

The availability of this hardware support allows a TM library to delegate most of its logic and concerns to a set of new hardware instructions provided by the HTM. Note, however, that the software in the TM library still plays an important role: given the restricted nature of existing HTM implementations, they are no panacea to resolving the whole challenge of synchronization.

HTMs, such as those released by Intel and IBM, are limited in several ways and are, as such, often referred to as *best-effort* HTMs (Damron et al. 2006; Christie et al. 2010; Nakaike et al. 2015). The consequence of that is having no guarantees at all that a hardware transaction succeeds independently of (the lack of) concurrency and contention. This means that the hardware support may simply abort all hardware transactions that are started. Naturally, this is motivated by pragmatical reasons: namely, the hardware keeps track of the transactional reads and writes in the processor

ALGORITHM 1: Intel RTM Usage to Provide a TM Abstraction.

```
 1: TX_START()
 2:  attempts ← MAX_ATTEMPTS
 3: begin:                                  ▷ used to jump to and reattempt with HTM
 4:  htmStatus ← _xbegin()
 5:  if htmStatus = _XBEGIN_STARTED
 6:     if is-locked(sgl)                   ▷ ensure correctness with fall-back
 7:        _xabort()
 8:     else
 9:        return                           ▷ hw transaction enabled, proceed to tx
10:  ▷ hw transaction aborted; decide on what to do next
11:  attempts ← attempts - 1
12:  if attempts = 0                        ▷ decision: give up on HTM, fall back to lock
13:     acquire-lock(sgl)                   ▷ software fall-back with a single lock
14:     return                              ▷ software fall-back path taken, proceed to tx
15:  ▷ decision: try again to execute with hardware support
16:  goto begin

17: TX_END()
18:  if _xtest()                            ▷ returns true if inside an HW transaction
19:     _xend()      ▷ tries to commit the HW transaction; jumps to line 4 when it fails
20:  else
21:     release-lock(sgl)                   ▷ executed with lock-based fall-back
```

caches (via the cache coherency protocol (Nakaike et al. 2015)), which are limited and thus may cause transactions to abort when overflows occur (due to transactionally accessed cache lines being evicted). Also, there are forbidden instructions that cannot happen transactionally (such as for performing I/O, or any system call in general). Finally, there is no contention management to enforce progress between groups of contending transactions, so it is possible that they just live-lock each other indefinitely (Diegues et al. 2014).

Motivated by all these limitations, programmers must use HTMs in cooperation with a so-called software fall-back path: upon each hardware transaction abort, a predicate is queried that decides whether to retry or to fall back to software, in which case that alternative path must ensure also correctness in the presence of concurrent threads running hardware transactions.

In Algorithm 1, we show a typical implementation of the software library that intercepts the calls to start and end transactions (e.g., exposed via the C/C++ constructs for atomic block (Ni et al. 2008)) and uses the canonical single-global lock fall-back path (Yoo et al. 2013). This will allow us to illustrate the challenges that we aim to tackle with this work.

In this pseudo-code, each transaction starts with a budget of attempts in hardware (as in line 2) that is decremented on each abort (in line 11). Hardware transactions[1] started with *_xbegin()* may, at any point of their execution, roll back automatically to the same line where it started. To distinguish between successful starts and aborted transactions, the *_xbegin()* call returns a status code that can be evaluated.

[1]The description of the compiler intrinsics used to manage Intel HTM can be found in https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/X86-transactional-memory-intrinsics.html.

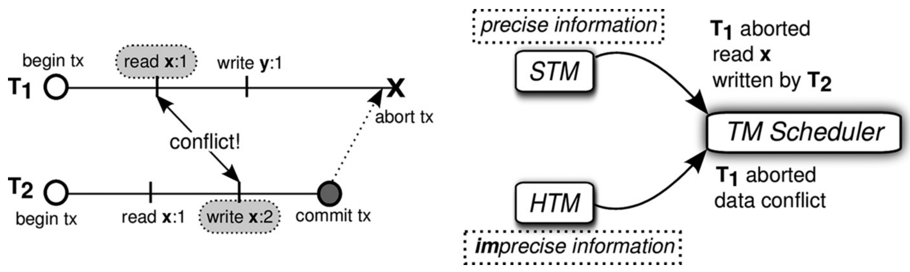


Fig. 1. Two transactions causing a conflict. The information returned by the TM varies depending on its nature: STMs are able to precisely identify the source of the abort, whereas commodity HTMs provide only a coarse categorization of the abort.

When the budget of attempts is exhausted, the execution falls back to the software path and uses a single-global lock that is maintained internally by the TM library (i.e., the programmer is still using only atomic blocks in his or her code). For this lock to be correctly used, hardware transactions verify that it is not locked as soon as they start (in line 6): if the lock changes concurrently after this check, then the hardware transaction gets aborted, because the lock was read transactionally and so the hardware detects any concurrent conflicting (i.e., write) access to it.

1.2 The 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. To enable this, the compiler typically instruments the application also to provide identifiers associated with the call to start and end a transaction, thus mapping them to atomic blocks in the source code.

However, most 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 (Rito and Cachopo 2014; Dragojević et al. 2009b). This happens because that software library has full knowledge of the read and write sets of the transactions and can be easily extended to externalize such information back to the scheduler. 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 TSX by Intel, 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 Diegues et al. (2014) and exemplified in Figure 1, these HTMs merely distinguish between a data conflict and other abort causes (e.g., exceeding hardware buffers). Referring back to Algorithm 1, this means either that the call to *xbegin* reports that the transaction is started or that some error occurred and caused it to abort (namely, capacity overflow of the processor caches, conflict with another undisclosed transaction, or some other problem due to the limitations of HTM).

For this reason, schedulers for STMs, which assume exact knowledge on the conflict patterns among transactions, cannot be straightforwardly employed with HTM, which are instead only capable of providing *imprecise information*. The problem is then how to devise a scheduler that can work with that very limited information and still make accurate decisions. This challenge is exacerbated by the strong impact that schedulers can have on performance: while appropriate scheduling decisions can provide better performance, the contrary is also true, in that erroneous scheduling (e.g., caused by inaccurate/incomplete knowledge) can degrade performance severely by unnecessarily restricting the available parallelism. Indeed, the relevance of correctly adapting HTMs to workloads, preferably automatically and transparently to the programmer, has also been recently acknowledged by Intel researchers (Kleen 2014).

1.3 Contributions

In this work, we propose Seer, the first scheduler (to the best of our knowledge) to address the HTM limitations discussed previously. The key idea of our proposal is to gather statistics at runtime to detect, in a lightweight but possibly imprecise way, the set of concurrently active transactions (i.e., those whose execution time frame overlaps) upon abort and commit events. Note that we say that two transactions, T_1 and T_2, run concurrently if T_2 finishes while T_1 is still active or, vice versa, T_1 finishes while T_2 is still active.

This information is used as input for an online inference technique that uses probabilistic arguments to identify conflict patterns between different atomic blocks of the application in a reliable way, despite the imprecise nature of the input statistics. The final step consists of 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 probabilistic hill-climbing technique that explores the configuration space of the model’s parameter while gathering feedback at runtime about the application’s performance 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 from false negatives) in capturing conflicts between atomic blocks of the application, since correctness for transactions is still enforced by the underlying HTM.

This set of core ideas behind Seer is not intrinsic to HTM—these contributions could as well be used to reduce conflicts through scheduling in STMs. In those cases, however, there are various solutions already established in the state of the art (see Section 2) that take advantage of the flexibility of STM to provide precise information about conflicts.

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 core, the likelihood of incurring aborts due to capacity exceptions can grow to such an extent that it can eventually cripple 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 over the accessed memory regions, Seer achieves also a drastic reduction of the frequency of activation of the software fall-back path of the HTM system, whose sequential nature is known to hamper HTM performance (Yoo et al. 2013; Jacobi et al. 2012; Heber et al. 2012).

This work extends a previous conference paper, where we initially proposed Seer (Diegues et al. 2015), in a number of directions. First, we have evaluated Seer on a wider range of benchmarks and workloads (in Section 6) and, most importantly, on a processor offering a larger degree of parallelism (28 cores now, vs. our original evaluation with eight cores). Moving the focus toward larger parallel architectures has led us to revisit (in Section 4) various important decisions underlying the original design of Seer, which were limiting its scalability. Finally, in Section 5, we introduce a new experimental study, based on simulations, which aims at assessing the accuracy of Seer's statistical inference techniques.

Overall, our experimental study shows that, by applying Seer to standard TM benchmarks, one can obtain gains of up to 3.6× and average speedups of 65% across various degrees of parallelism in a processor with 28 cores and HTM support. We also confirmed that our design yields more accurate probabilities than if we used other alternative calculations that use the same input data.

The rest of this article is organized as follows. In the following section, we survey the state of the art in TM schedulers—which we updated with recent publications that appeared since the publication of the conference version of this article—and identify 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. We then validate its design choices for the probabilistic inference scheme in Section 5. Finally, we evaluate our proposal in Section 6 by comparing it with several alternatives, and conclude in Section 8.

2 RELATED WORK

There is a rich body of work on TM schedulers. Next, we survey the state of the art in this field, and then summarize the characteristics of existing systems, contrasting them with our novel proposal Seer.

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 (Dolev et al. 2008) and Steal-On-Abort (Ansari et al. 2009), 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 precise information on aborts.

Steal-On-Abort, although initially implemented in software, was later also proposed for an HTM simulator (Ansari et al. 2010) by extending LogTM-SE (Yen et al. 2007) and running on a SPARC simulated machine. 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 (Jacobi et al. 2012) and cost of verification (Adir et al. 2014). Hence, it is particularly relevant to devise a scheduling solution for *current* HTMs: one that operates in the absence of accurate information on the conflict patterns among transactions, like Seer does.

More recently, ProPS (Rito and Cachopo 2014) followed a similar approach to the ones previously mentioned, 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 HTM. Shrink (Dragojević et al. 2009b) 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 access 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 (Rossbach et al. 2007) and SER (Maldonado et al. 2010) 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 aforementioned schedulers, ATS (Yoo and Lee 2008) works with imprecise information, that is, coping with the lack of knowledge on which pairs of transactions conflict during their execution. For that, ATS maintains a contention factor that is updated when transactions abort and commit, such that the allowed parallelism is adjusted: if more threads need to run transactions than those allowed, then they are blocked in a single waiting queue, which is equivalent to using a single semaphore. This strategy is thus agnostic of the specific conflict patterns, which is why we characterize it as a coarse-grained scheduler. The positive side is that it works with currently available HTMs.

To overcome the very simplistic approach of ATS, there is the recent proposal of the Octonauts scheduler (Mohamedin et al. 2015), in which there are fine-grained locks (in the form of serialization queues) for different objects in the program (or, conceptually, different memory regions). To achieve this, Octonauts requires a priori static information about the application/workload: to schedule a transaction, it uses static knowledge about the working set of a transaction, so that it can place the transaction in the queues for the objects to which it will read and write. This effectively serializes the transaction with concurrent conflicting transactions. To cope with incorrect static information, Octonauts still uses the HTM support to enforce correctness—similarly to our approach with Seer, in which our devised locking scheme need not be safe. Evidently, this approach suffers from a major drawback in that the programmers must provide this static information about the transactions' working set. This represents a nonnegligible source of complexity for programmers, hence contradicting one of the main motivations of TM, that is, to simplify concurrent programming. Conversely, Seer extracts any necessary information regarding conflict patterns among transactions in a runtime and completely transparent fashion to the programmers.

Finally, the approach of Advisory Locks (Xiang and Scott 2015) proposes to tackle the same issue in HTM by having the compiler identify and instrument potential parts of transactions that generate conflicts. To this end, conflicts are monitored at runtine and, based on the recent past history of conflicts, a decision is taken to grab a lock that protects small areas of shared memory. While the approach seeks to be very precise, by acquiring locks for what is really needed, it has the downside of requiring support for nontransactional operations within a hardware transaction—to allow acquiring the lock at the point where contention is predicted to happen, which is right before performing the potentially dangerous access to memory. Besides that, the proposal assumes additional support from the HTM to identify accesses causing conflicts. For these reasons, we classify this solution, in Table 1, as unable to operate using only imprecise information (unlike Seer).

We summarize the aforementioned state of the art in Table 1. Briefly, we can see that most schedulers cannot cope with imprecise information, as they rely on knowledge about the pairs of transactions that are involved in a data conflict, possibly along with information about the memory address that generates a conflict. To the best of our knowledge, ATS and Octonauts are the only two schedulers that can cope with that limitation inherent to commodity HTMs. On one hand, ATS uses only the number of aborts and commits to provide coarse-grained scheduling. On the other hand, Octonauts assumes static information about transactions' working set. As such, our contribution Seer is unique by being applicable to commodity HTMs (i.e., it works with

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, Whether It Uses Multiple Fine-Grained Locks to Schedule Transactions' Execution, and Whether It Does Not Require Static Information from the Workload

Scheduler	SW	HW	Imprecise Information	Fine Grained	No Static Information
Advisory Locks (Xiang and Scott 2015)	✓	✓	χ	✓	✓
ATS (Yoo and Lee 2008)	✓	✓	✓	χ	✓
CAR-STM (Dolev et al. 2008)	✓	χ	χ	✓	✓
Octonauts (Mohamedin et al. 2015)	✓	✓	✓	✓	χ
ProPS (Rito and Cachopo 2014)	✓	χ	χ	✓	✓
SER (Maldonado et al. 2010)	✓	χ	χ	✓	✓
Shrink (Dragojević et al. 2009b)	✓	χ	χ	✓	✓
SOA (Ansari et al. 2009, 2010)	✓	✓	χ	✓	✓
TxLinux (Rossbach et al. 2007)	χ	✓	χ	✓	✓
Seer	✓	**✔**	**✔**	**✔**	**✔**

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, (3) adopts a fine-grained serialization mechanism, and (4) does not require static information about the workload.

imprecise input) and allowing to serialize multiple transactions concurrently in a fine-grained manner without assuming any a priori knowledge on the application's workload.

Common to all these works is the capacity to deal with the dynamic arrival of transactions; that is, the scheduling is performed in an online fashion. Then, technically, they differ in terms of three main criteria: (1) SER (Maldonado et al. 2010) and TxLinux (Rossbach et al. 2007) work in cooperation (or by changing) the operating system kernel, to reduce the probability of contention between threads that are scheduled to be concurrently executed; (2) in CAR-STM (Dolev et al. 2008) and SOA (Ansari et al. 2009, 2010), the transaction execution needs to be movable between different threads, which may impose some burden on the programming API, and in Octonauts (Mohamedin et al. 2015), the transaction footprint needs to be known statically; and (3) with ATS (Yoo and Lee 2008), Shrink (Dragojević et al. 2009b), ProPS (Rito and Cachopo 2014), Advisory Locks (Xiang and Scott 2015), and our own proposal of Seer, there are no API changes and, as such, a thread may be blocked when it requests to start a new transaction. We argue that Seer achieves a good compromise: it relies on a nonintrusive approach while still providing considerable gains.

Recent works (Diegues and Romano 2015; Dice et al. 2014; Lev et al. 2007) have investigated the use of online profiling and optimization techniques in a similar spirit to what Seer does, but for a different, complementary purpose: to decide the best software fall-back and retry policies for the HTM. Performing such tuning is highly orthogonal to our contributions: at best, having a better-tuned system could allow more parallelism, which in turn could cause more transaction conflicts, which would allow Seer to provide even more improvements.

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.

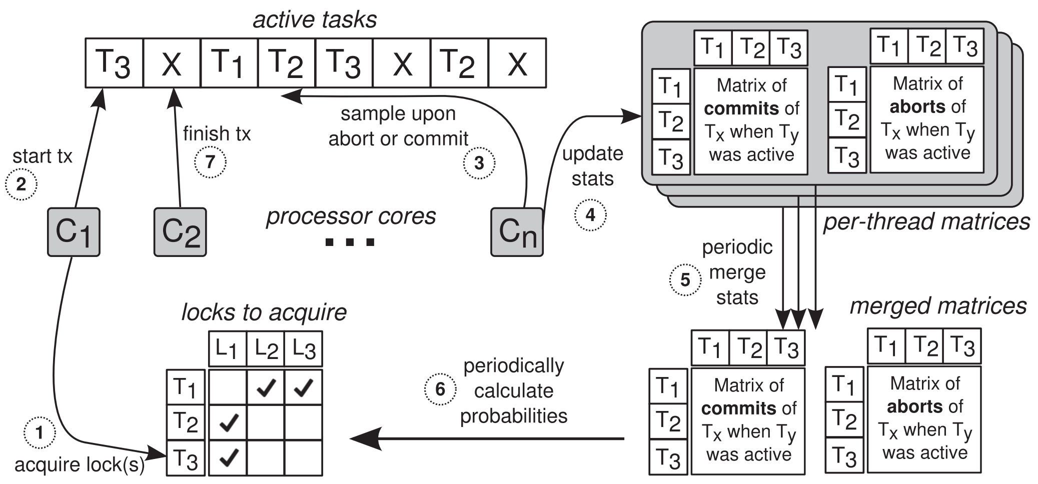

active tasks
T3 X T1 T2 T3 X T2 X
start tx 2
finish tx 7
sample upon abort or commit 3
update stats 4
C1 C2 processor cores ... Cn
Matrix of commits of Tx when Ty was active
Matrix of aborts of Tx when Ty was active
per-thread matrices
periodic merge stats 5
merged matrices
locks to acquire
L1 L2 L3
T1 T2 T3
periodically calculate probabilities 6
1 acquire lock(s)

Fig. 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 sampled upon commits and aborts, to compensate for the lack of feedback from the HTM. While not totally accurate, this information allows one 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).

Having access to such information is typically trivial in STMs. However, as discussed earlier in this article, mainstream HTMs provide little to no feedback with respect to this matter. In particular for Intel's HTM (and also for the various IBM 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, 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. 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 runtime 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 (meant for scheduling), 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 portrays 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, thus obtaining a fine-grained scheme.

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 cooperate 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 sampled[2] 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 (abortStats, respectively) tracks the frequency of commit (abort, respectively) events for transaction T_x in which T_y was found to be running (in the active transactions list) after the commit (abort, respectively) of T_x.

Consider the example of T_1 conflicts often only with T_3. Such fact is unknown beforehand and our approach aims to infer it at runtime: as we gather statistics, over time, recurrent events emerge and become identifiable using probabilistic inference.

Periodically, these statistics are merged, across all per-thread'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 earlier, 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 6. 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

[2]As we shall see, this sampling may range from scanning a full snapshot of the list to obtaining only one uniformly random concurrent transaction.

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 Are Used in Algorithms 2 to 6)

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-thread 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.

in the probabilistic inference, via a self-optimization mechanism that is driven by the feedback gathered at runtime on the throughput of the TM system.

Finally, Seer introduces the abstraction of *core locks*, that is, locks that prevent the concurrent execution of multiple hardware threads on the same core. The idea of core locks is based on the observation that, in workloads characterized by frequent transactions with nonminimal 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.

4 DETAILED ALGORITHM

We now present the detailed description of Seer. We first list the data structures and metadata used by Seer, in Table 2, most of which were already referred to in abstract terms in the overview.

Conventional HTM Usage. In the algorithms explained next, we shall use as a starting point the conventional HTM usage that we described initially in Section 1.1. We highlight lines, which are associated with the conventional HTM mechanisms, with a △ (other lines belong to Seer). Also, we note that now the START and END procedures' interface was augmented to receive the information that a given transaction *txId* is initiated by a given *thread*. This static/source code transaction identifier might be provided by a TM-aware compiler or, ultimately, by the programmer targeting the TM API (as in the current prototype of Seer). While in the following description we assume a closed universe of transactions, for ease of presentation, it is possible to have an unknown number of transactions and set up data structures for the metadata bookkeeping of new transaction identifiers on the fly, as new types of transactions are identified.

Also, we recall that 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 41). Note that the function to begin a hardware transaction, *_xbegin()* (in line 30), returns a status that normally represents that the transaction has started; that is, the predicate in line 31 evaluates to true. Otherwise, this status indicates a coarse categorization of the

ALGORITHM 2: Seer Algorithm.

```
22:  START(thread, txId)
23:    thread.core ← current-core()                              ▷ thread is bound to core
24:    thread.acquiredTxLocks ← false
25:    thread.acquiredCoreLock ← false
26:    activeTxs[thread.core] ← txId
27△    attempts ← MAX_ATTEMPTS
28△  begin:                                     ▷ used to jump to and reattempt with HTM
29:    WAIT-Seer-LOCKS(thread, txId)
30△    htmStatus ← _xbegin()
31△    if htmStatus = _XBEGIN_STARTED
32△      if is-locked(sgl)                             ▷ ensure correctness with fall-back
33△        _xabort()
34△      else
35△        return                                  ▷ hw transaction enabled, proceed to tx
36△    ▷ hw transaction aborted, handle before restarting
37:    REGISTER-ABORT(thread, txId)
38△    attempts ← attempts - 1
39△    if attempts = 0                                ▷ give up on HTM, fall back to lock
40:      RELEASE-Seer-LOCKS(thread, txId)
41△      acquire-lock(sgl)                              ▷ SW fall-back with a single lock
42△      return                                   ▷ SW fall-back path taken, proceed to tx
43△    ▷ before reattempting, trigger our scheduler Seer
44:    ACQUIRE-Seer-LOCKS(thread, txId, htmStatus)
45△    goto begin
```

abort. An aborted hardware transaction transparently jumps back and returns from this function, akin to the *setjmp*/*longjmp* mechanism used in C/C++.

With commodity HTMs, nested transactions are simply flattened in the outer transactional scope. Seer regulates the execution only of top-level transactions and is agnostic to the inclusion of any nested transaction that they may include.

Seer Algorithm. We now discuss the various mechanisms that augment this conventional HTM usage with Seer, briefly: (1) transactions are announced to other cores (see line 26), (2) aborts are registered in the per-thread statistics (see line 37), and (3) locks are used to induce fine-grained serialization between contending transactions (see lines 29 and 44). We present each part next.

The END procedure is presented in Algorithm 3 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-thread statistics in line 49 and possibly release locks acquired by our scheduler in line 50. Finally, we remove the transaction from the *activeTxs* list.

The procedures for registering aborts and commits are shown in Algorithm 4. The idea is to sample the *activeTxs* list and to increase the frequency of a transaction 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 from both false positives and false negatives.

ALGORITHM 3: Seer Algorithm.

```
46:  END(thread, txId)
47△    if _xtest()                                  ▷ returns true if inside a HW transaction
48△      _xend()                                    ▷ tries to commit the HW transaction
49:      REGISTER-COMMIT(thread, txId)
50:      RELEASE-SEER-LOCKS(thread, txId)
51△    else
52△      release-lock(sgl)                          ▷ executed with lock-based fall-back
53:    activeTxs[thread.core] ← ⊥
```

ALGORITHM 4: Seer Algorithm.

```
54:  REGISTER-ABORT(thread, txId)
55:    threadToSample ← getRandomThread()           ▷ following a round-robin scheme
56:    if activeTxs[threadToSample] ≠ ⊥
57:      thread.abortStats[txId][activeTxs[threadToSample]]++

58:  REGISTER-COMMIT(thread, txId)
59:    threadToSample ← getRandomThread()
60:    if activeTxs[threadToSample] ≠ ⊥
61:      thread.commitStats[txId][activeTxs[threadToSample]]++
```

Seer copes with this uncertainty using probabilistic inference techniques, whose details we shall discuss shortly.

Furthermore, notice that we are sampling the active transaction of one thread (following a round-robin scheme) instead of scanning the full list. This is to ensure that the overheads associated with this collection of statistics remain constant and independent of the number of threads used. On top of this, we highlight also that we follow a round-robin scheme; that is, each thread chooses a different thread every time it performs this sampling. This increases the statistical quality of the sampling in the presence of threads that have to retry the same transaction often due to hardware aborts. As we shall see in our evaluation, specifically in Section 6.4, these two choices are of paramount importance particularly given the large number of hardware parallelism available in the machine we used.

Notice that the aforementioned statistics are maintained per thread, that is, in a private fashion. Furthermore, the *activeTxs* list ends up being a set of single-writer multireader registers; we do not place any synchronization when accessing the list, with the intent of keeping it lightweight. Furthermore, we highlight that this *activeTxs* list can be made dynamic to support an arbitrary, not a priori known, number of threads issuing transactions.

The procedures for lock management, according to our scheduler, are defined in Algorithm 5. We use two types of locks:

(1) **txLocks:** one per transaction of the application, to serialize contending transactions according to the probabilities (line 68) that we describe later (in Algorithm 6). 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.

ALGORITHM 5: Seer Algorithm.

```
62: ACQUIRE-SEER-LOCKS(thread, txId, htmStatus)
63:  if htmStatus & _XABORT_CAPACITY ∧ ¬thread.acquiredCoreLock
64:     ▷ adapted to the topology of hyperthreads in Intel processors
65:     acquire-lock(coreLocks[thread.core % CORES])
66:     thread.acquiredCoreLock ← true
67:  if attempts = 1
68:     ACQUIRE-TX-LOCKS(txId)                    ▷ acquire locks in row locksToAcquire[txId]
69:     thread.acquiredTxLocks ← true

70: WAIT-SEER-LOCKS(thread, txId)
71:  if is-locked(sgl)                 ▷ avoid starting hardware transactions if the fall-back is in use
72:     if thread.core = 0                          ▷ only one thread updates the serialization locks
73:        UPDATE-SEER-LOCKS()                                   ▷ exploit the wait time to run SEER
74:        if enough-samples()
75:           stochastic-hill-climbing($\mathcal{T}h_1$, $\mathcal{T}h_2$)          ▷ periodically adapt the parameters
76:     wait while is-locked(sgl)                         ▷ wait here instead of aborting in line 33
77:  ▷ if some other thread is owning these SEER locks, cooperate with it and wait
78:  wait while ¬thread.acquiredTxLocks ∧ is-locked(txLocks[txId])
79:  wait while ¬thread.acquiredCoreLock ∧ is-locked(coreLocks[thread.core])

80: RELEASE-SEER-LOCKS(thread, txId)
81:  if thread.acquiredTxLocks
82:     RELEASE-TX-LOCKS(txId)
83:  if thread.acquiredCoreLock
84:     release-lock(coreLocks[thread.core])
```

(2) **coreLocks:** one per core of the processor, to reduce capacity aborts, which are amplified due to hardware threads that share the private caches of a 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 65).[3]

Furthermore, we also introduce a contention avoidance technique, which imposes waiting before starting a transaction (in line 29). This is presented in WAIT-Seer-LOCKS, in Algorithm 5, where there are two main ideas. First, we use a known technique to avoid the lemming effect (Dice et al. 2008). 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 33.

The second idea behind WAIT-Seer-LOCKS is to also wait in case the txLock and/or coreLock are taken by another thread (lines 78 and 79). The intuition is that, even though this thread may not have had aborts that led it to acquire locks, it is beneficial if it cooperates with concurrent

[3]Binding threads to cores simplifies the algorithm. However, this is not strictly necessary for correctness, and it could be changed to have threads checking the current core only from time to time (potentially acquiring wrong core locks, but with no harm to correctness).

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 note that the implementation proposed here uses simple mutual exclusion locks. However, we could reduce further the blocking/waiting caused between threads by employing a more complex locking scheme. Namely, when the locking scheme enforces T_1 to acquire a lock to avoid conflicting with T_2, the intent is that instances of T_1 do not run concurrently with instances of T_2. However, nothing is stated about instances of T_1 running concurrently with each other, and the same applies for T_2. So, in essence this could be obtained with a reader lock that would allow multiple instances of T_2.

The locking scheme is updated in line 73. At that point, the thread was waiting for the single-global lock to be released, for which reason executing the logic of Seer instead is not delaying the progress of the thread. 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 multireader 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, and 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 CASs into a single hardware transaction. If the transaction is not successful, we fall back to the normal acquisition. Note that this is not lock elision (Rajwar and Goodman 2001); we are effectively using HTM 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 Algorithm 6. This procedure, opportunistically invoked by the designated thread while waiting for the single-global lock to be released, starts by aggregating the commit and abort statistics gathered on a per-thread basis. The designated thread sweeps over the statistics of the other threads without any synchronization explicitly enforced, thus being subject to racy accesses. We avoid the hazard of "out of the blue" values by placing the data in word-aligned memory positions, which in this Intel 64 architecture guarantees that reads are atomic. Together with proper initialization and publication of the data structures, this avoids the aforementioned hazard and allows at best the thread to collect stale data not reflecting the commit/abort of some recently executed transactions. This design seeks a tradeoff that permits nonharmful data races while sparing from the synchronization overheads that would have to be incurred in order to prevent such racy accesses.

For each pair of transactions, x, y, of the application, we calculate the conjunctive probability of x aborting *and* y running concurrently (i.e., $\mathcal{P}^{conj}_{x,y} = P(\text{x aborts} \cap x \wr\wr y)$) (where $x \wr\wr y$ indicates the transactions x and y are concurrent):

$$\mathcal{P}^{conj}_{x,y} = P(\text{x aborts} \cap x \wr\wr y) = P(\text{x aborts} \mid x \wr\wr y) \times P(x \wr\wr y))$$
$$= \frac{a_{x,y}}{c_{x,y} + a_{x,y}} \times \frac{c_{x,y} + a_{x,y}}{\sum_{k=0}^{|A|} a_{x,k} + c_{x,k}} = \frac{a_{x,y}}{\sum_{k=0}^{|A|} a_{x,k} + c_{x,k}},$$

where we abbreviated the elements of the matrices commitStats$[x][y]$ as $c_{x,y}$ and abortStats$[x][y]$ as $a_{x,y}$. Note that this probability calculation can be efficiently approximated with the statistics that are at our disposal, as shown in the previous algorithms, where we describe how to collect them.

As shown in Algorithm 6, we must then identify the probabilities that are most meaningful, and for which locking should pay off in terms of forbidding some parallelism but avoiding

ALGORITHM 6: Seer Algorithm.

```
85: UPDATE-SEER-LOCKS()
86:  for all x ∈ A                    ▷ A is the set of txs in the application source code
87:    $\eta \leftarrow$ average$\Big(\big\{$P(x aborts ∩ x≀≀y), ∀y∈ A$\big\}\Big)$
88:    $\sigma^2 \leftarrow$ variance$\Big(\big\{$P(x aborts ∩ x≀≀y), ∀y∈ A$\big\}\Big)$
89:    for all y ∈ A                  ▷ determine if y is likely to contend with x
90:      ▷ 1st condition checks whether abort events of x, in which y is seen running concurrently,
         are common enough
91:      ▷ 2nd condition checks if y is among the txs that, when executed concurrently with x, most
         likely contend with x
92:      if ( P(x aborts ∩ x≀≀y) > $\mathcal{T}h_1$ ∧
           P(x aborts ∩ x≀≀y) > $\mathcal{T}h_2$-th percentile of a Gaussian $\mathcal{N}(\eta,\sigma^2)$ ) then
93:          locksToAcquire[x] ← y          ▷ contending txs take each other's locks upon abort
94:          locksToAcquire[y] ← x          ▷ txs also wait for their tx locks to be free (line 78)
95:  ▷ sort all locks in each row of locksToAcquire and swap the old matrix by the new one (using an
     indirection pointer)
```

significant hardware transaction aborts (and thus diminish the reliance on the single-global lock). To achieve this, we resort to two thresholds, $\mathcal{T}h_1$ and $\mathcal{T}h_2$, which are aimed at pursuing different goals.

The threshold $\mathcal{T}h_1$ establishes a lower bound on $\mathcal{P}^{conj}_{x,y}$, below whose value Seer avoids serializing transactions x and y. Low values of this probability imply that the frequency of abort events of x, in which y was found to run concurrently with it, are rare. It is thus 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 cutoff on the probability distribution of $\mathcal{P}^{conj}_{x,y}$, which aims at determining a subset $\mathcal{S}$ among the transactions available in A that are responsible for the most likely conflicts with a given x.

More in detail, Seer includes in $\mathcal{S}$ only each transaction y whose $\mathcal{P}^{conj}_{x,y}$ is larger than the $\mathcal{T}h_2$-th percentile of a Gaussian distribution $\mathcal{N}(\eta,\sigma^2)$ with mean η and variance σ equal, respectively, to the mean and variance of the values of $\mathcal{P}^{conj}_{x,y}$ (for each transaction y with respect to a given x).

In other words, this second threshold aims at identifying the transactions y that have the *relatively* highest probability values of conflict with x. This contrasts with the first threshold, which instead identifies the transactions that have the largest *absolute* values of conflict with x.

Seer can overestimate that *absolute* probability threshold for two main reasons. First, and most importantly, Seer can falsely blame a transaction y for having aborted x, merely because y is spotted as executing right after x gets aborted. Second, in order to minimize monitoring overheads, Seer adopts a lightweight and inherently imprecise mechanism for tracking concurrency among transactions, which also causes falsely blamed transactions.

As we shall show in Section 5, due to this risk of overestimation, if Seer were to use only $\mathcal{T}h_1$, it could overly restrict parallelism for certain workload types (e.g., when there are few transaction types and a high number of concurrent threads). By using also $\mathcal{T}h_2$, though, Seer can reason on the distribution of conflict probabilities and pinpoint in a more accurate way which transactions are actually the most likely causes of conflict, which can be beneficial to reduce false-positive rates of locks.

Table 3. Simple Example for a Matrix C with Four Types of Transactions Where Their Conflicts Are Known A Priori

-	tx_1	tx_2	tx_3	tx_4
tx_1	**1.00**	0.00	0.00	0.10
tx_2	0.00	0.20	0.00	**0.50**
tx_3	0.00	0.01	0.05	0.00
tx_4	**1.00**	0.00	0.00	0.00

The row and column identifiers are the identifiers of the four types of transactions.

Summarizing: if both conditions in line 92 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 one lock per transaction).

Finally, Seer relies on an online self-tuning mechanism that automates the setting 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 suboptimal 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 to guide the search in the parameter's space [0,1]×[0,1] for the thresholds $\mathcal{T}h_1$ and $\mathcal{T}h_2$. The feedback is based on throughput of transactions per processor cycle, where the latter are obtained through Intel 64 instruction RDTSC, which provides very cheap monotonic measurements over different invocations that are relative to the CPU clocks.

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 typically used with TM (Diegues and Romano 2015). Specifically, we set p to 0.1% and the initial values of $\mathcal{T}h_1 = 0.3$ and $\mathcal{T}h_2 = 0.8$.

The Seer algorithm, together with the benchmarks used in this work, are available as an open-source project at https://github.com/nmldiegues/htm-seer.

5 VALIDATING THE DESIGN CHOICES OF Seer

In order to motivate the choices made while designing Seer, we first validate the proposed probabilistic lock inference scheme by evaluating its accuracy via a simulation study. This choice allows us to assess the accuracy of Seer when faced with a large number of randomly generated, yet known a priori, synthetic workloads. Further, by relying on a simulator, we can exert tight control on the degree of accuracy with which Seer can assess concurrency among transactions. This allows us to study the impact on Seer's locking inference scheme due to imprecise information regarding concurrently executing transactions.

In the validations that we shall conduct in this section, we always synthesize the conflict likelihood between transactions, so that they are known a priori and enforced in the execution by our simulator.

Specifically, in the simulation model, we use a *conflict matrix* C, whose cells $C_{i,j}$ define the probability for transaction Tx$_i$ to be aborted by transaction Tx$_j$, in case they run concurrently (recall the definition provided in Section 1.3). An example of matrix C is shown in Table 3: in this case, Tx$_4$ aborts deterministically (i.e., probability of 100%) if it runs concurrently with transaction Tx$_1$, whereas transaction Tx$_1$ aborts with probability 10% if it executes concurrently with Tx$_4$.

Table 4 summarizes the key parameters used in the simulation model used in this study. Each data point in the following plots corresponds to the average of 100,000 simulations. In each

Table 4. Description of Parameters Used in the Simulations

Parameter of the Configuration	Values Used
Number of simulations per configuration	100,000
Number of rounds per simulation	100,000
Number of threads	Uniformly distributed in $[2; 32]$
Number of transaction types	Uniformly distributed in $[2; 32]$
Chance of running a transaction	75%
Chance of choosing a specific transaction type	Chosen given uniform distribution
Transaction conflicts	Given by C
Zipfian distribution for C	Given by $\mathcal{Z} \in [1.0; 2.5]$

simulation, we vary the number of threads, transaction types, and the conflict matrix as indicated in Table 4. In particular, each row of the conflict matrix is generated using a Zipfian distribution, whose $\mathcal{Z}$ parameter is treated as an independent variable in our study. In the following, we shall vary $\mathcal{Z}$ in the $[1.0; 2.5]$ range, where we recall that $\mathcal{Z}$ values closer to 1.0 generate more uniform distributions, and larger values result in more skewed distributions. This allows one to mimic realistic cases (such as those we test later) where there are large discrepancies between the conflict patterns of different transaction types.

Each simulation consists of the execution of 100,000 *rounds*. In each round, each thread first chooses whether to run a transaction or not (given a likelihood of executing transactional work of 75%). In the positive case, it picks one of the transactions available in the workload with equal probability. In each round, for each pair of threads, it is decided whether a conflict occurs between the transactions that they are executing based on the conflict matrix C. If a transaction Tx_i conflicts, with probabilities according to C, with at least one other transaction active in a different thread, then Tx_i is aborted.

Each thread concludes the round by looking at the *activeTxs* and updating its thread-level statistics accordingly. This lock-step round-based procedure provides exact information on which other transactions were concurrently active during an execution round. Later, in this section, we extend this simulation model to include scenarios in which the sampling of the *activeTxs* array can lead to obtaining erroneous information. This will allow us to simulate more closely Seer, which, we recall, relies on minimal synchronization and possibly inaccurate information.

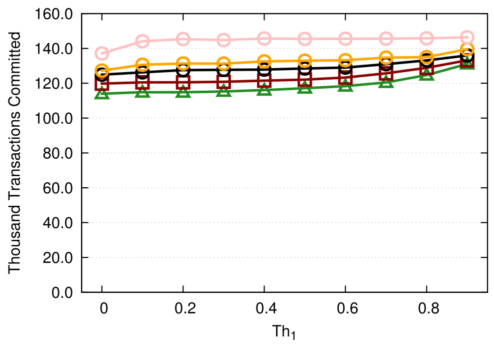
We highlight that this simulation model encompasses a representation also of the fall-back path with a single-global lock: hardware transactions resort to a single-global lock, after attempting to use HTM 5 times, thus forbidding all concurrent transactions from being executed in that round.

It should be noted that our simulation model does not encompass spurious aborts triggered by the HTM. Also, our simulation model does not capture the effects of running multiple hardware threads on the same core. Hence, this study does not aim to evaluate the effectiveness of core locks, for which the reader should refer to Section 6.

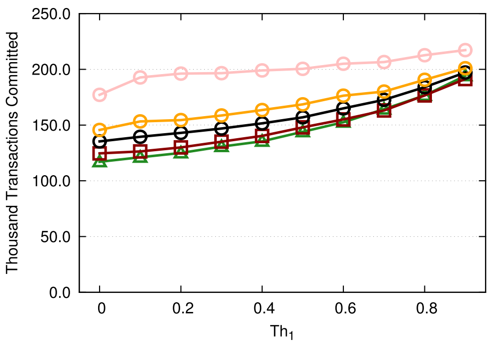
In the simulator, we monitor the execution of the first third of the rounds without acquiring any Seer-induced lock. On the basis of the statistics gathered during this preliminary phase, we use the same logic of Seer to compute $\mathcal{P}_{x,y}^{conj}$.

We then execute the remaining two-thirds of the 100,000 rounds, using the locks chosen by Seer, and evaluate its accuracy as follows:

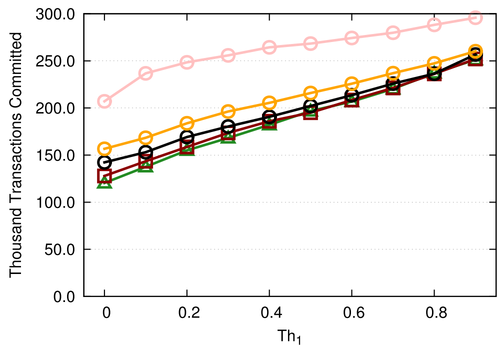
$$accuracy = \frac{truePositives + trueNegatives}{totalEvents}. \quad (1)$$

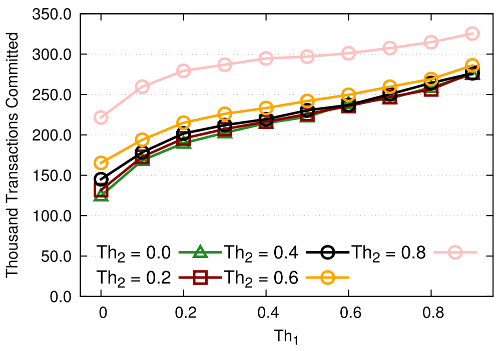


(a) Conflict matrices $\mathcal{C}$ generated for each simulation from Zipfian distribution with $\mathcal{Z} = 1.0$.



(b) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 1.5$.





(c) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 2.0$.

(d) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 2.5$.

Fig. 3. Simulated performance for Seer in four scenarios varying the parameter $\mathcal{Z}$ of the Zipfian distribution that is used to generate the conflict matrices C used as input for each simulation. In each plot, we vary the value of both thresholds that are applied to $\mathcal{P}^{conj}_{x,y}$.

A true positive happens when there would exist a conflict between a pair of transactions that is prevented by a lock devised by Seer. In contrast, a true negative corresponds to a pair of transactions that execute concurrently without conflicting, and for which Seer did not decide to enforce a transaction lock. The total number of events is the number of rounds—in each of which we execute this logic—times the number of pairs of transactions that execute concurrently.

Note that, while Seer relies on a throughput-driven hill-climbing-based self-tuning procedure to set the values of the $\mathcal{T}h_1$ and $\mathcal{T}h_2$ thresholds, in this simulation study we shall treat these thresholds as two independent variables. This choice allows us to gain additional insights on the impact of tuning these two thresholds independently. We will evaluate the gains achievable by using Seer's self-tuning hill-climbing strategy versus static thresholds in Section 6 when presenting the experimental evaluation of our prototype.

We present in Figure 3 the performance in four scenarios with different parameters for the Zipfian distribution that generates the conflict matrices C. In each plot, we show the results for different values of both thresholds that are applied to $\mathcal{P}^{conj}_{x,y}$.

In general, there are two relevant trends that emerge from these data: using the value of 0.0, for either $\mathcal{T}h_1$ or $\mathcal{T}h_2$, results in worse performance than if both thresholds are set to some, properly selected, positive value. As such, this suggests that the proposed approach of combining these thresholds results in better accuracy in terms of locks chosen, which matches our earlier intuitions.

We also note that the correct settings of the thresholds appear to have a smaller impact for the scenarios in which the conflict matrices are more uniform. In contrast, the relevance of properly setting those thresholds is higher in more skewed workload scenarios.

This can be explained by considering that the more skewed the conflict matrix C is, the smaller the set of transactions that cause most of the aborts for a given transaction Tx. This has the effect of reducing the uncertainty that Seer's lock inference scheme has to cope with. In fact, Seer's design is tailored to identify a small subset of the conflicting transactions—the one that matters the most—exactly because skewed workloads are frequently found in realistic applications (Cooper et al. 2010) (namely, of those that we evaluate also later). That is, normally, not every transaction conflicts often with every other transaction type.

Another important trend is visible across the plots: since the workloads have different characteristics, the best accuracy is achieved when using different combinations of the threshold values. For more uniform workloads (e.g., Figure 3(a)), it is important to use a high $\mathcal{T}h_2$ value, since with uniform workloads the conflict probabilities tend to be very similar. Thus, the use of high values for $\mathcal{T}h_2$ allows for identifying more easily the top conflicting transactions among those that are already above the average for a given transaction. When the workloads are more skewed, in contrast, it is more important to use a sufficiently high $\mathcal{T}h_1$ value that allows one to distinguish the high conflicts from the lower ones, in order to achieve higher accuracy levels.

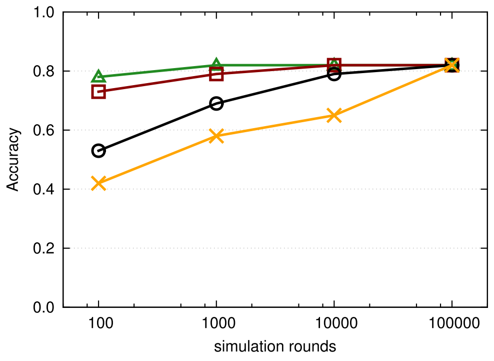
Overall, these results suggest that the optimal settings of the thresholds $\mathcal{T}h_1$ and $\mathcal{T}h_2$ are largely workload dependent: the difference in accuracy can vary from 22% (in Figure 3(a)) to 48% (in Figure 3(d)). These experimental data motivate the use of Seer's hill-climbing-based self-tuning approach, rather than the use of static settings. It should be noted that, although workloads considered in this study may suggest that the use of larger threshold values would always produce the best results, this is not actually the case for all possible workloads. This claim is supported by the experimental results reported in Section 6.1, which show that the use of self-tuning can lead to speedups as large as 50%.

Finally, we analyze also the impact of feeding the lock inference scheme of Seer with approximate/imprecise information regarding the set of concurrently active transactions. Recall that, in the experiments analyzed so far, concurrent transactions have always been inferred correctly, as if there was a global synchronization point when accessing the *activeTxs* list.

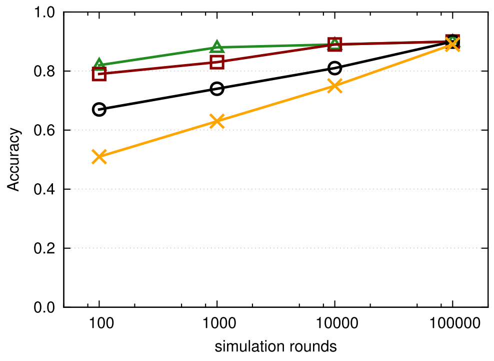
The plots shown in Figure 4, instead, allow us to study how the accuracy of Seer changes when varying the probability of feeding it with imprecise information regarding concurrent transactions. In this case, whenever a transaction Tx samples an entry of the *activeTxs*, it obtains a random transaction type, instead of the correct one, with probability p_{err}. This allows us to simulate the real behavior of Seer, which samples transactions from the *activeTxs* list without any synchronization, and is thus vulnerable to erroneously consider as concurrent transactions that had either already completed when Tx aborted or that had not yet started when Tx aborted.

The plots in Figure 4 consider the same workload settings of Figure 3 and use the values of $\mathcal{T}h_1$ and $\mathcal{T}h_2$ that yielded the best accuracy in the previous experiments. In this case, though, on the horizontal axis, we let the number of simulation rounds vary from 100 to 100,000 and report the performance achieved while varying the probability of sampling a random concurrent transaction instead of the current one (p_err).

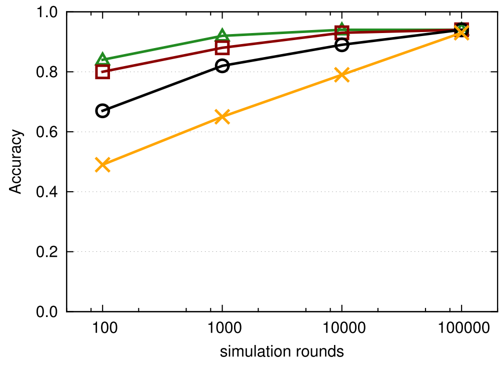
The reason for this is that, by increasing the probability p_{err} of wrongly inferring concurrency, we increase also the noise in the data collected by Seer. As the plots clearly show, this has only



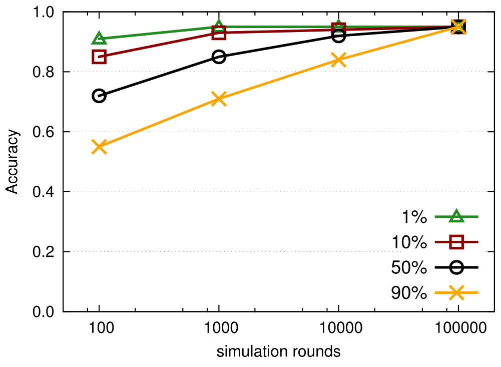
(a) Conflict matrices $\mathcal{C}$ generated for each simulation from Zipfian distribution with $\mathcal{Z} = 1.0$.



(b) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 1.5$.



(c) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 2.0$.



(d) Conflict matrices $\mathcal{C}$ for each execution simulation from Zipfian distribution with $\mathcal{Z} = 2.5$.

Fig. 4. Average accuracy, when varying the number of simulation rounds, to show how the convergence time is affected by the imprecise gathering of statistics when accessing the *activeTxs* list. In each plot, we show four alternatives that simulate different probability values (p_{err}) of sampling a random concurrent transaction instead of the correct one. We show four scenarios, similarly to Figure 3, by varying the parameter $\mathcal{Z}$ of the Zipfian distribution that is used to generate the conflict matrices C used as input for each simulation. In each plot, we use the thresholds for $\mathcal{T}h_1$ and $\mathcal{T}h_2$ that provided the best accuracy in the corresponding scenario in Figure 3.

the effect of delaying the correct inference of the actual conflict probabilities, as with a sufficiently large number of observations the accuracy converges to the values observed in Figure 3.

Furthermore, it is also evident that, when gathering statistics for a low number of rounds, it is more likely to infer concurrent transactions wrongly. However, as we increase the number of rounds, the accuracy quickly grows to expected value. We also highlight that, in TM applications, it is easily the case that throughput rates are well over thousands of transactions per second. This is easy to understand given that transactions are typically small and execute over in-memory data. As such, these numbers of simulated rounds correspond to a very short real time period. Hence, in practice, Seer can infer a highly accurate locking scheme in a robust and timely way.

In general, these results are also important as they attest to the high accuracy that is achievable by Seer in a wide range of workloads, that is, 90% average accuracy across all the workloads with proper threshold tuning.

Table 5. Benchmarks Used in the Evaluation

Reference	Benchmark	Number of Different Transactions
STAMP (Minh et al. 2008)	Genome	5
	Intruder	3
	Kmeans-High	3
	Kmeans-Low	3
	SSCA2	10
	Vacation-High	3
	Vacation-Low	3
	Yada	6
Guerraoui et al. (2007)	STMBench7	45

6 EXPERIMENTAL EVALUATION AND COMPARISON WITH OTHER SYSTEMS

We now report the results of an experimental study based on a fully fledged prototype of Seer, in which we compare the proposed solution with three other state-of-the-art systems.

To evaluate our proposal, we formulate several questions and design a set of experiments aimed to answer them:

— *What are the gains achievable by* Seer *?* In Section 6.1, we compare Seer with three alternative techniques for regulating the execution of HTM transactions.
— *How are those gains obtained?* In Section 6.2, we provide detailed data on how often hardware transactions are aborted and why.
— *How are transactions scheduled?* In Section 6.3, we assess how often hardware transactions are successful and to what extent the various available locks are acquired.
— *What are the overheads of* Seer*?* In Section 6.4, we assess the performance slowdown of running Seer's monitoring and lock inference mechanisms, but without using the locks synthesized by Seer.
— *What are the relative merits of each component of* Seer*'s architecture?* In Section 6.5, we create and compare several variants of Seer that incrementally use the different modules of the proposed system.

All the results in this article were obtained using an Intel Haswell Xeon E5-2683 v3 processor with 28 hardware threads (14 cores, each one running up to two hardware threads with Intel Hyper-Threading). The machine was equipped with 128GB of RAM and ran Ubuntu 14.10. The results reported in this article are the average of 10 runs. The GCC 4.8.2 compiler was used, and the memory allocator was the default *glibc*.

Our evaluation uses the benchmarks reported in Table 5. Among these is the standard STAMP suite (Minh et al. 2008), a popular set of benchmarks for TM, encompassing applications representative of various domains that generate heterogeneous workload. We excluded Bayes given its nondeterministic executions, and Labyrinth as most of its transactions exceed Intel HTM capacity. In addition to these, we have also used the STMBench7 benchmark (Guerraoui et al. 2007), which is particularly interesting for scheduling as it encompasses 45 different transactions types. Due to the limitations of Intel HTM capacity, we have reduced the working set of STMBench7's workloads by roughly 10 times (namely, the number of components per module to one and levels of assemblies to two) so that there is a relatively small number of aborted transactions due to capacity (less than 20%, in contrast with over 50% normally).

6.1 How Much Can We Gain with Seer?

To assess the benefits of Seer, we compare it with three alternatives for the concurrency control:

— **HLE,** where transactions may be retried a small number of times (processor implementation dependent), but without any scheduling or contention management. As pointed out in the literature (Dice et al. 2008), this can cause a lemming effect on the elided lock; that is, failed hardware transactions keep exhausting the attempts and fall back to using the single-global lock.[4]

— **RTM,** where the retry logic is controlled in software: we retry a given number of attempts in HTM and always wait before doing so if the single-global lock is taken. This is the typical mechanism used when relying on hardware transactions (Yoo et al. 2013).

— **SCM,** where we implemented the Software-Assisted Conflict Management (Afek et al. 2014) technique. SCM uses an auxiliary lock to serialize transactions that are aborted, thus decreasing the chance of having the lemming effect.

We used a budget of five attempts for hardware transactions RTM, SCM, and Seer, which is also the value used by Intel researchers for a similar set of benchmarks (Yoo et al. 2013).

We first present the results for this comparison with STAMP in Figure 5. The speedups are relative to a sequential noninstrumented execution. In general, we can see that Seer performs better or similar to the best alternative. This best alternative is sometimes SCM, but other times RTM, with some advantage of the former when there are fewer threads used. Finally, HLE performs significantly worse than the others.

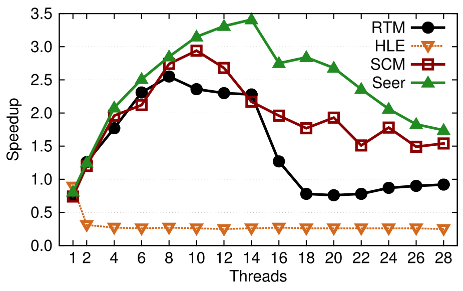
By analyzing these results with a focus on Seer, we can identify three groups of benchmarks with different merits of performance:

— Genome, Intruder, Vacation-high, and Vacation-low: these benchmarks clearly illustrate the advantage of the proposed probabilistic scheduler for hardware transactions. The gains of Seer are consistent and unveil a new scalability ceiling with respect to the baselines using the same hardware parallelism. Furthermore, when exploiting the highest number of threads available, the scheduling devised by Seer minimizes performance degradation and avoids trashing.

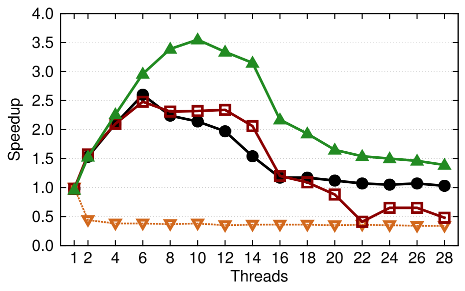
— SSCA2, Kmeans-low, Yada: performance is generally on par with the best of the considered alternatives. As we shall see in more detail in the rest of the evaluation, this happens because (1) the contention level is very limited, meaning there are almost no conflicts to avoid (SSCA2); (2) transactions are small and the workload is not 100% transactional, for which reason reliance on the single-global lock is not very harmful (Kmeans-low); or (3) reducing conflicts unveils more capacity aborts (Yada).

— Kmeans-high: in this case, the performance of Seer is actually decreasing (with respect to the best alternative) as the number of threads increases. This happens because transactions in this workload, although quite conflicting, are still very short. As a result, the locking scheme devised by Seer does not pay off because, when using the baselines alternatives instead, the transactions that abort take the single-global lock for such a short duration that it has only a limited impact on performance. Ideally, Seer would eventually self-tune its thresholds to prevent this situation, as it happens in some of the other benchmarks where its performance is similar to the baselines. However, for this benchmark, we have confirmed that this does not happen because this benchmark is extremely fast (lasting less than a

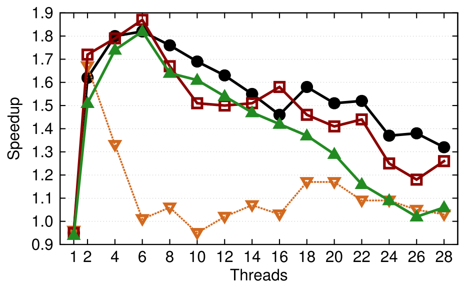
[4]All the benchmarks herein used have a transactional interface, that is, were written using atomic blocks, and, as such, there is only one (global) lock to elide (maintained internally by the TM library).



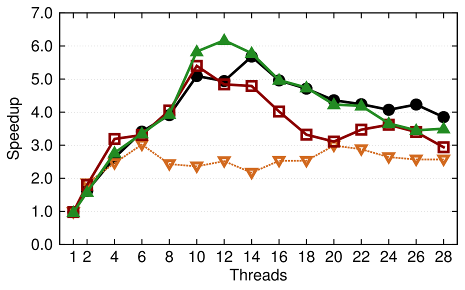
(a) Genome.



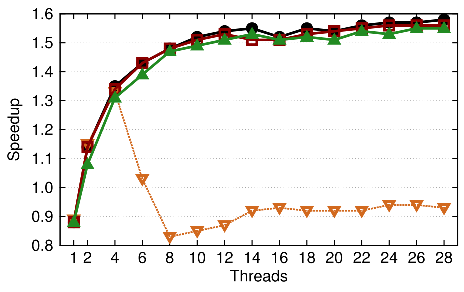
(b) Intruder.



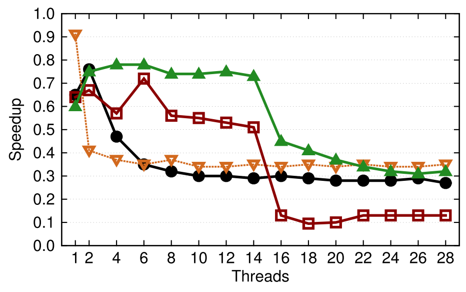
(c) Kmeans-high.



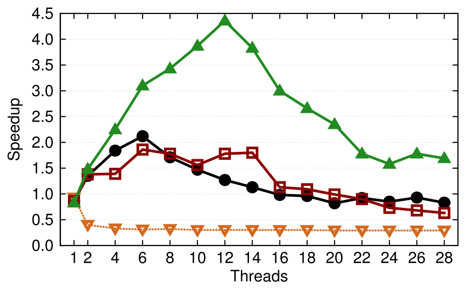
(d) Kmeans-low.



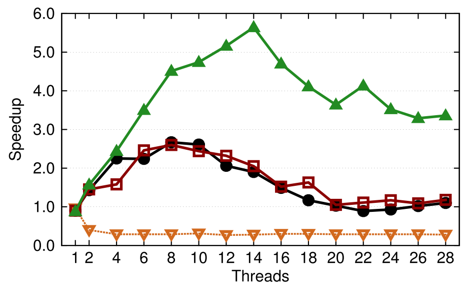
(e) SSCA2.



(f) Yada.

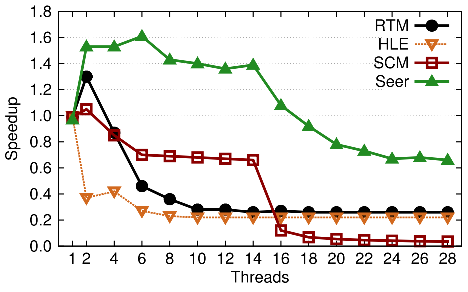


(g) Vacation-high.

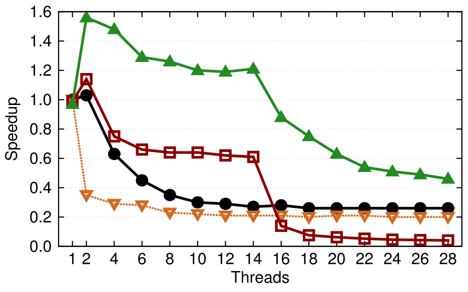


(h) Vacation-low.

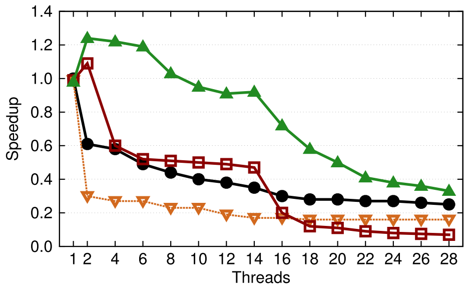
Fig. 5. Speedup of different HTM-based approaches across **STAMP** benchmarks.



(a) Read-dominated.

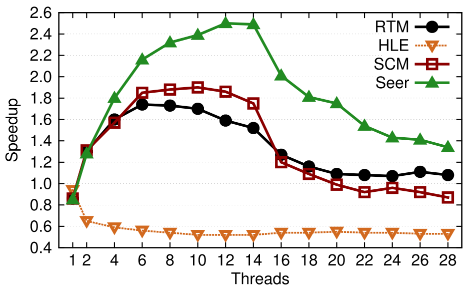


(b) Read-write balanced.

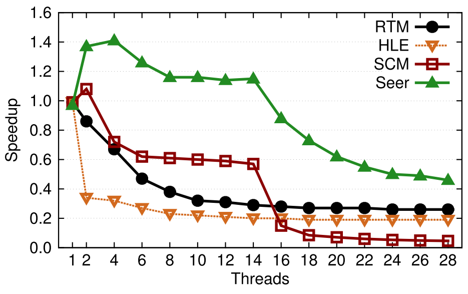


(c) Write-dominated.

Fig. 6. Speedup of different HTM-based approaches across **STMBench7** workloads.



(a) Geometric mean for STAMP.



(b) Geometric mean for STMBench7.

Fig. 7. Geometric mean speedup for STAMP and STMBench7.

second for most of the parallel executions). To verify this, we have experimentally executed the benchmark in a closed loop, while preserving the statistics collected by Seer, and verified that it converges to the performance of the other baselines (namely, of RTM) in three to five runs.

Finally, with respect to STAMP, we analyze also the geometric mean speedup across all benchmarks that is shown in Figure 7(a). This data evidences that, despite some workloads where performance remains on par with some of the alternatives, the gains are quite substantial for most of the considered number of threads: namely, 64% over RTM and 42% over SCM at 14 threads (i.e., the peak of average speedup), and up to 77% over SCM at 20 threads. The gains are also up to 4.8× over HLE.

Moving on to STMBench7 (Guerraoui et al. 2007), we present results for three workloads, in which we vary the percentage of read-only transactions. This is a particularly challenging setting for a TM system because the 45 different transaction types[5] have very heterogeneous characteristics, ranging from very small to huge transactions, small and large read working sets, and varying contention degrees. As such, this opens more opportunities in general for scheduling, and results also in a more complex test for Seer.

It is also noteworthy that these workloads are quite contended, as it is visible by the rather small scalability potential and speedup values; this stems not only from the nature of the benchmark, as evidenced by other evaluations with Software TMs (Fernandes and Cachopo 2011; Dragojević et al. 2009a; Rito 2015; Rito and Cachopo 2014), but also from the working-set reduction that we performed to make it adequate to the capacity of hardware transactions.

Although the absolute speedups vary with the workload changes, the relative merits of each solution remain approximately the same. As such, we focus on the geometric mean across the three workloads, which is shown in Figure 7(b). There, we observe an improvement of Seer of at least 2× at four threads (i.e., the peak of average speedup) and up to 3.7× over RTM and 5.3× over SCM at 14 threads.

The trend across all benchmarks is quite visible and solid: Seer enables more scalability up to some degree of parallelism and then avoids drastic performance plunges when the contention becomes very high with dozens of concurrent threads.

To better appreciate the reasoning behind Seer, we present in Table 6 some of the locking schemes that it automatically induced in these experiments. The idea of each matrix is similar to that presented in the description of Figure 2: each row indicates the locks that transactions of that type must acquire; each cell (i, j) of the matrix specifies whether Seer allows concurrency between transactions of type T_i and T_j.

We seek to illustrate contrasting examples with the matrices shown.

The reported matrices report examples corresponding to diverse workloads, which allow for demonstrating Seer's ability to devise heterogeneous scheduling polices. For example, KMeans-low has a reasonably low amount of transaction aborts, mainly due to the fact that its transactions are short, and more than half of the time is spent in nontransactional code. Therefore, even when a large number of threads (namely, 28) are active, our system imposes only one transaction lock (to forbid the concurrent update of a single queue). Seer is able to recognize that the other transaction types can proceed concurrently with little conflict and thus avoids acquiring the corresponding locks. For the case of Genome, there are more data structures updated in different transaction types, but still the learned scheduling imposes only a fine-grained blocking for two of the transactions (specifically, forbidding their parallel execution only while allowing them to run concurrently with other transaction types). This is due to the fact that T_1 and T_2 manipulate data structures (hashtable-1 and hashtable-2), which are not accessed by any other transaction type.

Finally, we focus on Vacation by presenting two contrasting workloads: a low-conflict workload with a low number of threads, and a high-conflict one with a high amount of threads. In the former, Seer identified that deleting customers can cause some avoidable conflicts with the travel agency reservations, since deleting a customer in T_2 also deletes the reservations and availability of resources, which are accessed by T_1. Since the update of resources, performed by transaction T_3, is a rare transaction in the low-conflict workload, no locks are devised for it. In contrast, in the

[5]We note that STMBench7 source code has a single atomic block, which would prevent Seer from differentiating the transaction types. Therefore, we modified the benchmark to use the TM API exposed by Seer and pass the transaction type identifier upon the transaction beginning. To automate this, one could rely on the call stacks to understand the true source code origin for a runtime transaction (Dice et al. 2014).

Table 6. Examples of Devised Locking Matrices That Define the Fine-Grained Scheduling Imposed by Seer

KMeans-low 28threads	tx_1	tx_2	tx_3
tx_1 (update clusters)	χ	χ	χ
tx_2 (update queue)	χ	✓	χ
tx_3 (update delta)	χ	χ	χ

Genome 8threads	tx_1	tx_2	tx_3	tx_4	tx_5
tx_1 (hashtable-1 insertions)	✓	χ	χ	χ	χ
tx_2 (iterate array, write one slot)	χ	χ	χ	χ	χ
tx_3 (hashtable-2 insertion)	χ	χ	✓	χ	χ
tx_4 (hashtable-3 insertion)	χ	χ	χ	χ	χ
tx_5 (update segments read from tables)	χ	χ	χ	χ	χ

Vacation-low 2threads	tx_1	tx_2	tx_3
tx_1 (make reservation)	χ	✓	χ
tx_2 (delete customer)	✓	χ	χ
tx_3 (update resources)	χ	χ	χ

Vacation-high 28threads	tx_1	tx_2	tx_3
tx_1 (make reservation)	✓	✓	✓
tx_2 (delete customer)	✓	✓	✓
tx_3 (update resources)	✓	✓	✓

Each table denotes the benchmark used and the number of threads that executed, together with a brief description for each of the transaction types. For each transaction type (in each row), we then denote whether it is forced to acquire the n^{th} lock or not (where n refers to the lock corresponding to tx_n, thus indicating a relevant conflict probability between the transaction of the row and tx_n). A ✓(χ, respectively) in position (i, j) of the matrix denotes whether transaction T_i is required (not required, respectively), to acquire T_j's lock.

high-conflict workload, Seer imposes a very constrained scheduling with all locks under usage. In such case it could be argued that one would be better off using only the single-global lock. However, we note that there is no noticeable penalty of using the additional locks in this scenario, since they are acquired in parallel with the execution of the single transaction that is in the critical path under the single global lock. As an optimization, it would be possible to detect scenarios in which transactions have to acquire a large number of fine-grained locks and acquire, instead, a smaller number of coarse-grained locks (similarly to the ideas in the hierarchical locking of TinySTM (Felber et al. 2008)).

Furthermore, we highlight that each thread maintains only two matrices with $N \times N$ slots (where N is the number of transaction types). Hence, even for STMBench7 with 45 transaction types and 28 threads, this sums up to about half a megabyte. Adding to this, there is some centralized metadata (where the statistics are aggregated), but that is also negligible compared to the half megabyte already. Therefore, the memory footprint is residual.

6.2 Where Are the Gains of Seer Coming From?

The approach taken by Seer is to bridle parallelism so that transactions can make a more efficient use of HTM.

In the previous subsection, we already quantified the performance benefits globally achievable by Seer. We now focus our experimental study to shed light on the origins of the speedups that it achieves. In particular, we aim to answer the following two key questions: (1) to what extent aborts of hardware transactions are reduced and (2) to evaluate the consequent reduction of activation of the fall-back path.

Once again, we show results for both the STAMP (in Figure 8) and STMBench7 (in Figure 9) benchmarks. In those plots, we can see, on the one hand, the percentage of hardware transaction aborts (specified by type) and, on the other hand, the percentage of processor cycles spent waiting for the single-global lock to become available. For SCM, we also include in this processor cycle count the cycles spent spinning on the auxiliary lock. For Seer we also report the cumulative percentage of cycles spent waiting for transactions and core locks. This data is shown for each workload and while varying the number of threads for both Seer and for the three considered baselines.

We distinguish three types of aborts for hardware transactions:

— *conflict*: aborts due to concurrent threads issuing contending data accesses. Note that the subscription of the single-global lock may generate aborts that are reported as conflicts: when a hardware transaction starts, it verifies that the single-global lock is free and continues to execute the transaction, so that if the lock gets acquired, the hardware transaction shall abort.
— *capacity*: aborts triggered when a hardware transaction accesses more cache lines than those that fit the hardware limits (namely, of the processor caches).
— *sgl*: explicit abort requests triggered in case a hardware transaction finds the fall-back lock busy upon having subscribed it.
— *other*: aborts triggered due to other reasons, for example, page faults and interrupts.

In general, more than half of the aborts are due to conflicts. These are the ones that feed Seer's statistics, and that the transaction locks try to reduce. Then, most of the other aborts left are due to capacity, which Seer attempts to limit via the core locks.

By analyzing each benchmark, we can see that Seer does reduce the total aborts in several cases (namely, in the Genome, Intruder, Vacation, and STMBench7 benchmarks) when compared to the approach with RTM. There are two noteworthy remarks with respect to the other two alternatives. First, recall that HLE insists very little on hardware transactions, and so it ends up aborting less in total because it is the only approach that does not have a budget of five retries per transaction (as experimental evaluation seems to indicate that it retries one or up to two times at most). Second, SCM does have the same budget of five retries as Seer, but since it serializes aborted transactions via the auxiliary lock, it restricts parallelism upfront and in a coarse fashion; recall that SCM relies on a single auxiliary lock, whereas Seer's locks are fine grained.

As such, these two alternatives manage to reduce transaction aborts. Yet, this comes at the cost of severely hindering parallelism. Evidence of this claim can be clearly obtained by analyzing the data regarding the processor cycles spent waiting for the fall-back lock (and, for SCM, also for the auxiliary lock) to be free. For the case of Seer we show not just the cycles spent waiting for the global lock, but also for Seer's transaction and core locks. When considering the average

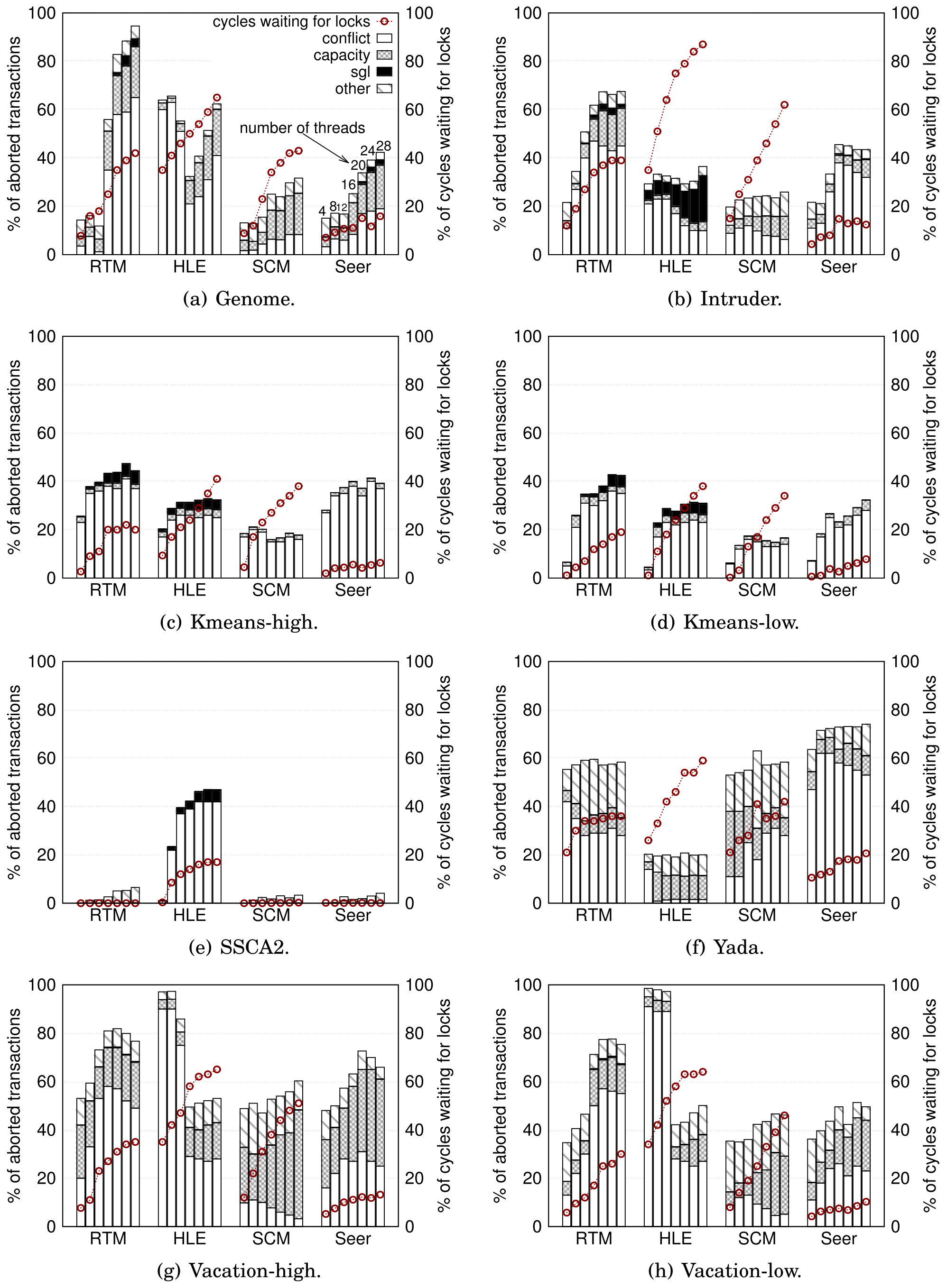
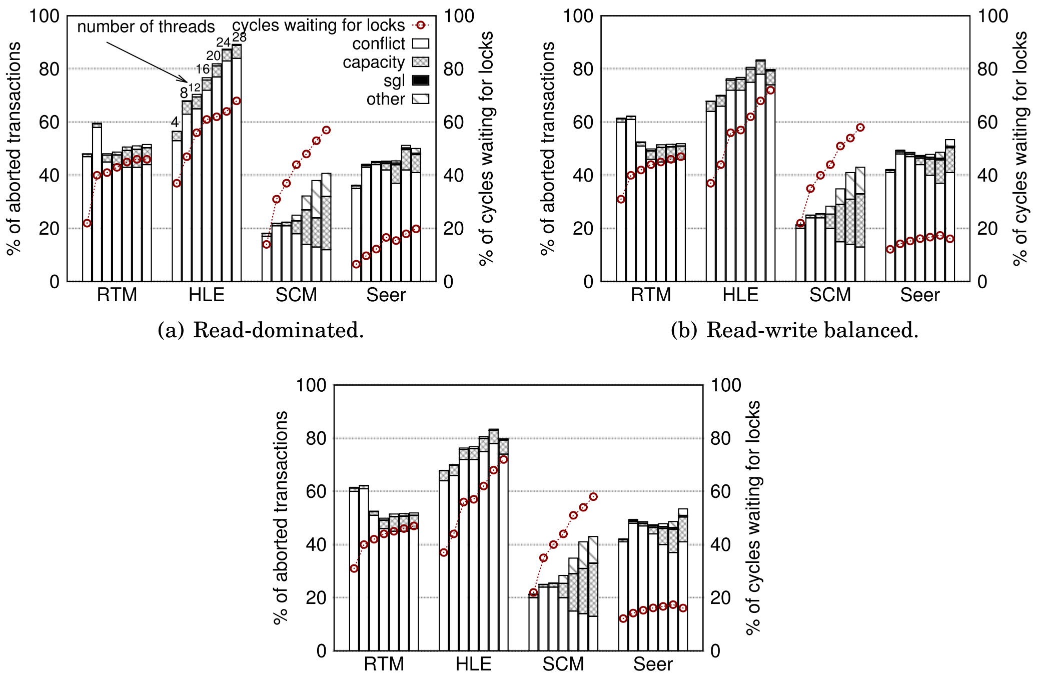


Fig. 8. Number and types of aborts suffered by hardware transactions across the **STAMP** benchmarks, together with the percentage of processor cycles spent waiting for the single-global lock to be available (including the auxiliary lock for SCM). In the case of SEER, we include also the cumulative percentage of cycles spent spinning on transaction and core locks.



(a) Read-dominated.

(b) Read-write balanced.

(c) Write-dominated.

Fig. 9. Similar to Figure 8, but instead for the **STMBench7** workloads.

cycles spent waiting for locks, we have that Seer has 4× less than HLE, 3.5× less than SCM, and 3× less than RTM. Notice that, while RTM has the closest cycles spent waiting compared to Seer, it is also the alternative that consistently aborts more. This tension between waiting for locks and aborting transactions is a tradeoff that is best managed by Seer's ability to generate fine-grained scheduling that prevents aborts while at the same time avoiding too much waiting.

Another interesting aspect to highlight is that, in some benchmarks, the ability of Seer to reduce conflict aborts leads to an increase of capacity aborts. This can be explained by considering that transactions whose conflicts are avoided, thanks to Seer's transaction locks, now advance longer in their execution and hence become more vulnerable to capacity exceptions. We observed this phenomenon experimentally, for instance, in Vacation-high. In this case, we can see that the transaction locks of Seer reduce the conflicts (when comparing to RTM), but then Seer ends up with more capacity aborts—which the core locks do not amortize entirely. RTM, on the other hand, does not observe those capacity aborts because the hardware transactions abort early due to conflicts.

6.3 How Are Transactions Being Scheduled?

To understand the reasons that lead Seer to reduce aborts and wait time for the fall-back lock, we now present results that explain how transactions are being scheduled. Namely, we seek to understand how often transactions rely on the single-global lock (thus giving up on the HTM support) or when they acquire other locks (in the case of SCM and Seer).

Table 7 provides a breakdown of the usage of locks for each considered approach, providing additional insights on the reasons underlying the performance gains achieved by Seer. We report

Table 7. Breakdown of Percentage (%) of Transaction Execution Modes When Running with 14 and 28 Threads

		Genome		*Intruder*		*Kmeans-H*		*Kmeans-L*		*SSCA2*	
Variant	Execution Mode	14t	28t	14t	28t	14t	28t	14t	28t	14t	28t
HLE	HTM no locks	71	71	77	77	70	70	71	72	76	78
	SGL fall-back	29	29	23	23	30	30	29	28	24	22
RTM	HTM no locks	97	90	92	89	76	76	89	82	100	100
	SGL fall-back	3	10	8	11	24	24	11	18	0	0
SCM	HTM no locks	91	82	84	72	76	74	75	75	98	98
	HTM + Aux lock	7	16	14	24	23	25	24	24	2	2
	SGL fall-back	2	2	2	4	1	1	1	1	0	0
Seer	HTM no locks	96	79	87	76	75	59	90	76	100	100
	HTM + TxLocks	2	3	5	11	8	24	5	14	0	0
	HTM + CoreLocks	0	4	0	2	0	0	0	0	0	0
	HTM + Tx + CoreLocks	0	8	0	2	0	1	0	1	0	0
	SGL fall-back	2	6	8	9	17	16	5	9	0	0

		Yada		*Vacation-H*		*Vacation-L*		*STMB7-R*		*STMB7-RW*		*STMB7-W*	
Variant	Execution Mode	14t	28t	14t	28t	14t	28t	14t	28t	14t	28t	14t	28t
HLE	HTM no locks	82	84	66	64	53	65	90	90	92	90	92	90
	SGL fall-back	18	16	34	36	47	35	10	10	8	10	8	10
RTM	HTM no locks	83	83	75	77	80	79	92	92	94	94	95	97
	SGL fall-back	17	17	25	23	20	21	8	8	6	6	5	3
SCM	HTM no locks	68	65	64	58	68	68	58	60	54	57	51	51
	HTM + Aux lock	18	20	30	35	27	28	36	33	40	36	42	42
	SGL fall-back	14	15	6	7	5	4	6	6	6	7	7	7
Seer	HTM no locks	84	78	84	59	88	76	95	94	96	95	98	96
	HTM + TxLocks	2	5	5	8	5	0	3	3	3	2	1	0
	HTM + CoreLocks	0	2	0	12	0	5	0	0	0	0	0	1
	HTM + Tx + CoreLocks	0	0	0	1	0	3	0	1	0	1	0	1
	SGL fall-back	14	15	11	20	7	16	2	2	2	2	1	2

data for 14 and 28 threads, for each benchmark and workload. To interpret this data, it is important to consider that, for each approach, it is desirable that the top row(s) have the highest frequency, because they are the ones that constrain parallelism the least, while still executing with HTM support successfully.

Table 8. Distribution of Transaction Duration (Processor Cycles) across Benchmarks

Benchmark	50th Perc	90th Perc	95th Perc	99th Perc
Genome	10k	12k	13k	55k
Intruder	<1k	7k	12k	22k
Kmeans-High	<1k	<1k	<1k	<1k
Kmeans-Low	<1k	<1k	<1k	<1k
SSCA2	<1k	<1k	<1k	1k
Vacation-High	14k	22k	25k	42k
Vacation-Low	10k	15k	17k	24k
Yada	<1k	68k	83k	108k
STMBench7-R	<1k	130k	132k	133k
STMBench7-RW	<1k	3k	131k	133k
STMBench7-W	1k	4k	7k	130k

By considering Genome, we can see that HLE has a considerable portion of transactions executing under the single-global lock, due to its fragile retry-on-abort policy. This is a trend that persists across benchmarks. As for SCM, we can see that it relies very little on the single-global lock, because it ends up replacing it with the auxiliary lock most of the time.

When comparing RTM with Seer, we can see that the latter reduces a few percentage of the usage of the single-global lock (1% at 14 threads and 4% at 28 threads). This is an interesting result, because we have verified that the performance gains can be quite substantial (e.g., on Genome), despite having only a small effect on the global lock usage. The main reason for this is that not all transaction types are born equal: some generate much longer transactional executions than others.

This phenomenon is quantified in Table 8, which reports data on the distribution of the processor cycles spent to execute transactions (serially, without any instrumentation) across all the considered benchmarks. These data highlight that, for Genome, the largest 1% transactions take 5× longer to execute than the median. As a result, even though Seer may prevent only a small portion of transactions to use the single-global lock, the consequence may be a considerable performance improvement in case those transactions are much longer than the median.

Analogously to Genome, we can then see that Intruder and the STMBench7 workloads evidence exactly the same pattern: the single-global lock reduction of Seer is not astounding, but in contrast we can see a large discrepancy between the median and greater than 90th percentiles (of up to $> 100\times$ difference in transaction length).

With both Vacation workloads, we verify that the transaction lengths are more homogeneous. In fact, in these workloads, the performance gains of Seer are directly related to a significant reduction of the usage of the single-global lock: 14% in Vacation-High and 13% in Vacation-Low, when compared to RTM at 14 threads (which is close to the peak of speedup).

As for the Kmeans workloads, we had previously seen that Seer did reduce transactional conflicts. This new data now clarifies that such result is achieved thanks to the significant usage of transaction locks (up to 24% in Kmeans-High) and a negligible usage of core locks (given that there are almost no capacity aborts). Yet, as discussed earlier, we now confirm in Table 8 that the transactions are very small and largely homogeneous in these workloads. For these reasons, the usage of the single-global lock has a minor impact on performance, nullifying the gains of Seer.

We conclude this analysis by presenting, in Table 9, average results across all benchmarks, but reporting data for a larger spectrum of thread counts. These results confirm that HLE is the least

Table 9. Breakdown of Percentage (%) of Types of Transactions Used on Average across All Benchmarks

Variant	Execution Mode	4t	8t	12t	16t	20t	24t	28t
HLE	HTM no locks	80	70	71	75	76	75	75
	SGL fall-back	20	30	29	25	24	25	25
RTM	HTM no locks	96	94	91	89	88	88	88
	SGL fall-back	4	6	9	11	12	12	12
SCM	HTM no locks	84	82	82	82	80	82	79
	HTM + Aux lock	13	16	16	15	17	16	19
	SGL fall-back	3	2	2	3	3	2	2
Seer	HTM no locks	97	94	91	85	82	80	81
	HTM + Tx Locks	1	2	4	6	7	8	7
	HTM + Core Locks	0	0	0	1	1	2	2
	HTM + Tx + Core Locks	0	0	0	1	2	1	1
	SGL fall-back	2	4	5	7	8	9	9

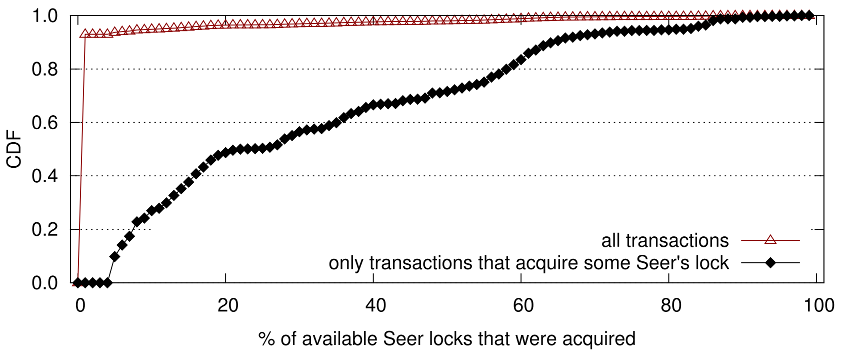


Fig. 10. Cumulative distribution function for the ratio (in %) of available locks that Seer chose to take, averaged across all the scenarios evaluated in this section. We show this distribution for all transactions, and also for the subset of transactions (on average 7%) that acquires at least one Seer lock.

effective solution among the considered alternatives, with the largest usage of the single-global lock (mainly due the lemming effect (Dice et al. 2008)). RTM improves over those results, but it still yields an average usage of the single-global lock that is larger than 10% in most cases. SCM reduces that substantially to at most 3%, but at the cost of using more than 10% of the time the auxiliary lock. This is only slightly better than queuing all transactions on the single-global lock.

Finally, Seer is able to improve over all previously described alternatives, exactly because the frequency with which it uses a single-global lock is lower: at most 9% average for 28 threads. Furthermore, the other locks that Seer exploits have a much finer granularity—one per transaction and one per core. Hence, unlike the single-global or the auxiliary locks, Seer's locks avoid serializing all transactions.

To complement the results so far, we also show in Figure 10 the cumulative distribution function for the ratio of available locks that Seer decides to acquire, averaged across all the tested scenarios. Among all transactions, we note that only about 7% acquire at least one Seer lock, when averaging across benchmarks and number of threads. We highlight that in 50% of these cases, in which *some* lock of Seer is acquired, the fraction of locks that are actually acquired is lower than 23%

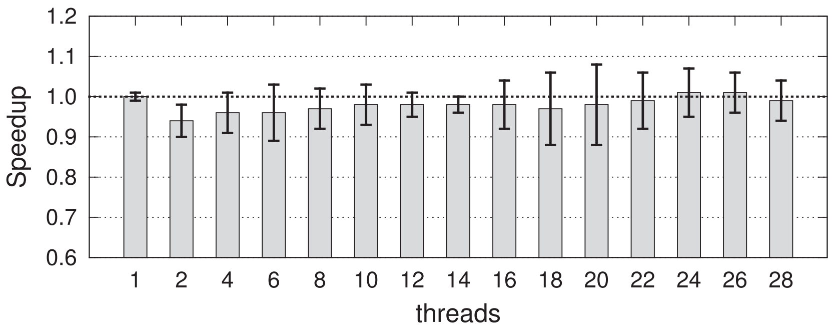


Fig. 11. Geometric mean overhead of Seer, when profiling and calculating locks to acquire, across **all benchmarks and workloads**.

of the available ones. This experimental result confirms the ability of the proposed lock inference mechanism to synthesize effective fine-grained locks for scheduling.

6.4 What Is the Overhead of Running Seer?

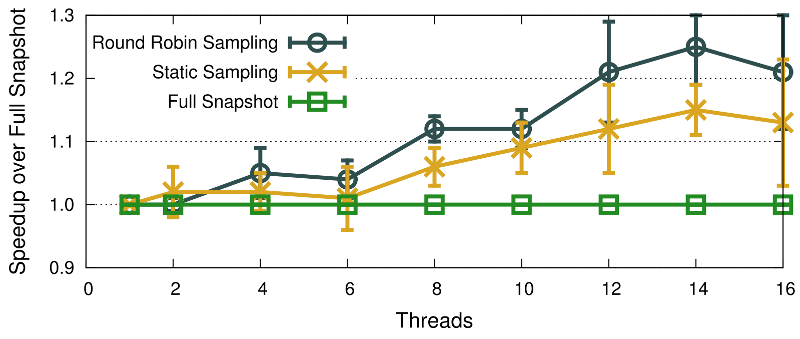
We first assess the overhead of the monitoring, lock-inference, and self-tuning mechanisms of Seer. For this, we ran a variant of Seer that incurs the overheads of all its mechanisms, without, however, acquiring any lock.

In Figure 11, we show the average speedup of this Seer's variant relative to RTM (that consistently performed second best in our previous evaluation results). In this new plot, we present the geometric mean across all benchmarks (of STAMP and STMBench7). The mean slowdown across all these number of threads is 2%, which confirms that the infrastructure used by Seer to gather data and perform statistical inference is, in fact, quite lightweight. The peak overhead is actually with two threads, with 6% slowdown, because in that case we enable the infrastructure of Seer—which is disabled when there is only one thread—and thus half of the threads (i.e., one thread) now periodically run the algorithms for devising the lock scheme and self-tuning the thresholds. As the number of threads increases, this overhead is amortized by the presence of additional threads that execute transactions and merely increase their thread-local statistics counters.

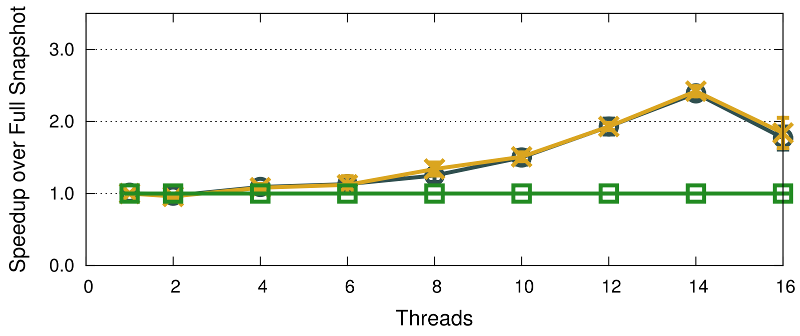
Note that the technique used to sample the *activeTxs* list has the utmost importance in keeping these overheads low as we increase the number of threads. We recall that each thread, upon committing or aborting with a conflict, samples the transaction being executed by only one other thread in the system. This is in contrast with the alternative extreme approach of obtaining all transactions of all other threads (an approach that we call *Full Snapshot*). The problem with this latter alternative is that its overheads increase with the number of concurrently active threads. As such, the intent of sampling just one entry of *activeTxs* is to keep those overheads constant.

In Figure 12, we show the speedup of Seer's approach in *Round -Robin Sampling* when compared to that of the *Full Snapshot* approach. We also show a slightly different alternative that we call *Static Sampling*, in which each thread always samples one given (different) thread, instead of changing the sampled thread over time as *Round-Robin Sampling* does.

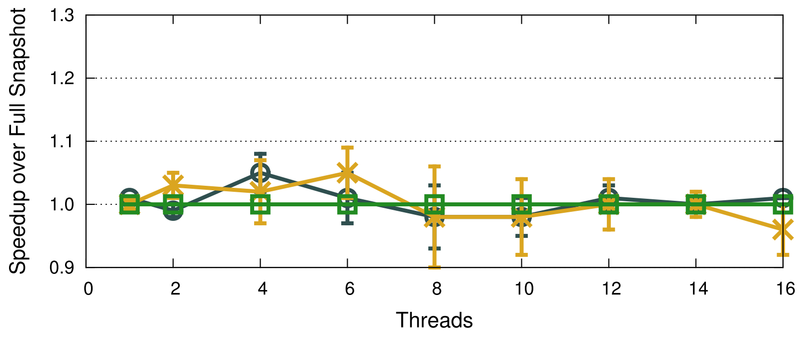
The averaged performance improvements (across all benchmarks) shown in Figure 12(a), relative to *Full Snapshot*, clearly show that *Round Robin* yields improvements over the two alternatives, more so as the number of threads increases. The improvements over *Full Snapshot* are explained by the reduced overheads, as shown in Figure 12(b) for the benchmark SSCA2, where there are almost no aborts: both sampling alternatives provide considerable speedup over a *Full Snapshot*. Note that this is not necessarily always the case, as shown in Figure 12(c), where we



(a) Average across **all benchmarks and workloads**.



(b) Workload with small transactions and abort ratio (SSCA2 from STAMP).



(c) Workload with large time-dominant transactions (STMBench7).

Fig. 12. Comparison of performance improvements for three schemes to sample the transactions running concurrently in Seer (relative to an approach that snapshots all concurrent transactions). In all cases, Seer runs the same algorithm and changes only the sampling scheme of the *activeTxs* list.

use an STMBench7 workload: this has very large transactions, and thus the overhead of sampling at the end of the transaction is amortized significantly in contrast with workloads such as that of SSCA2. As a consequence, the three sampling schemes end up having very similar performance.

Finally, the difference in performance between both sampling techniques is explained by the better quality of sampling in conflict-prone workloads: by using round robin over all threads, for instance, this prevents cases where a thread constantly samples the same active concurrent transaction in a thread, which is struggling to progress due to aborts (perhaps due to capacity overflow). As such, *Round -Robin Sampling* appears to be the most robust solution, by providing higher-quality sampling (over *Static Sampling*), while at the same time reducing the overheads (over *Full Snapshot*).

6.5 How Much Does Each Design Choice Contribute to Seer?

The design of Seer encompasses the following five aspects: (1) capturing statistics about commits, aborts, and concurrent transactions; (2) acquiring transaction locks when aborts occur; (3) acquiring a core lock when a capacity abort happens; (4) acquiring transaction locks by using a hardware transaction to reduce the overheads of multiple compare-and-swaps; and (5) adapting the thresholds $\mathcal{T}h_1$, $\mathcal{T}h_2$ via a stochastic hill-climbing algorithm.

To quantify the relative relevance of each of the mechanisms integrated in Seer, we conducted a series of experiments, whose results, shown in Figure 13, evaluate the speedup of different variants of Seer. We consider as baseline the Seer variant previously considered for the plots in Figure 11, which incurs the costs of collecting statistics and updating the locking strategy, without ever acquiring any lock. Then, we consider four progressively enhanced variants, where we cumulatively add: the transaction locks acquisition (+ *tx-locks*), the core locks acquisition (+ *core-locks*), the acquisition of locks by employing a hardware transaction (+ *htm locks*), and the adaptation of the thresholds via hill climbing (+ *hc*).

In general, the transaction locks provide the largest boost in performance. Unsurprisingly, the core locks are only beneficial when using 16 or more threads, that is, when we start executing multiple hardware threads on the same core. Note that, although we strive to reduce both conflict and capacity aborts, the latter are many times unavoidable independently of preventing cache sharing in the same core via the core locks. Instead, many of the data conflicts are effectively prevented by constraining the parallelism via the fine-grained scheduling.

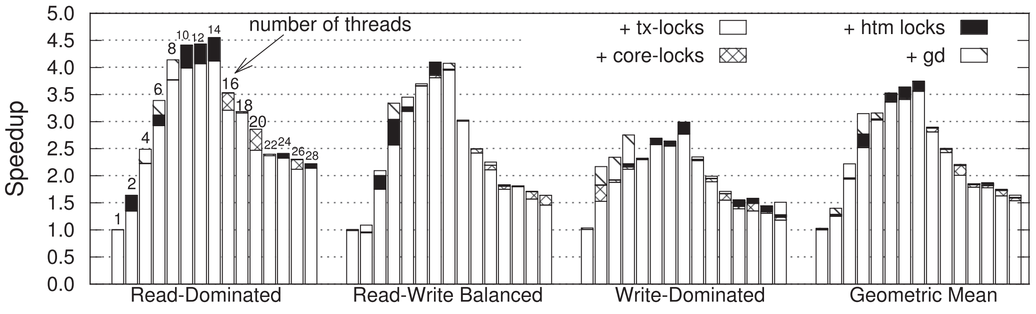
The hardware lock acquisition also shows improvements in general across all numbers of threads. Also, a similar gain is provided by adapting online the thresholds used in the transaction locks probabilities calculations.

On average, when considering the geometric mean in the last plot, we get that (1) transaction locks yield an average improvement of up to 74%, (2) on top of which the core locks add an improvement of up to 10%, (3) the lock acquisition via hardware transactions adds an improvement of up to 9%, and (4) the stochastic hill climbing adds an improvement of up to 21%.

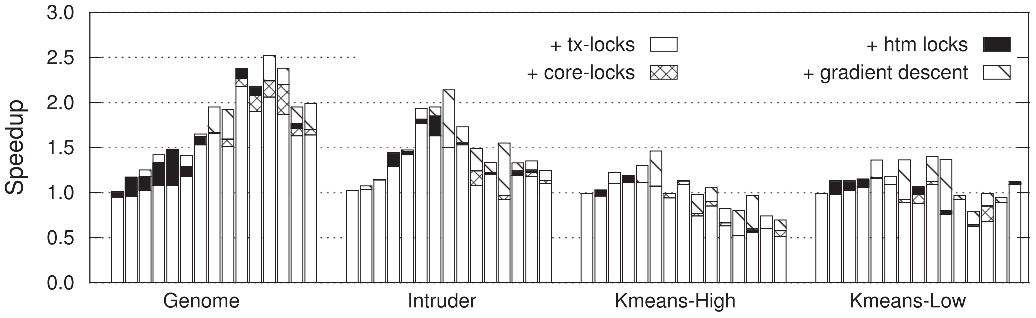
7 LIMITATIONS AND POSSIBLE OPTIMIZATIONS

In this section, we discuss what, we argue, represents the main limitations of Seer and discuss alternative approaches, optimizations, and open research questions that could be pursued in future work to further enhance Seer's efficiency.

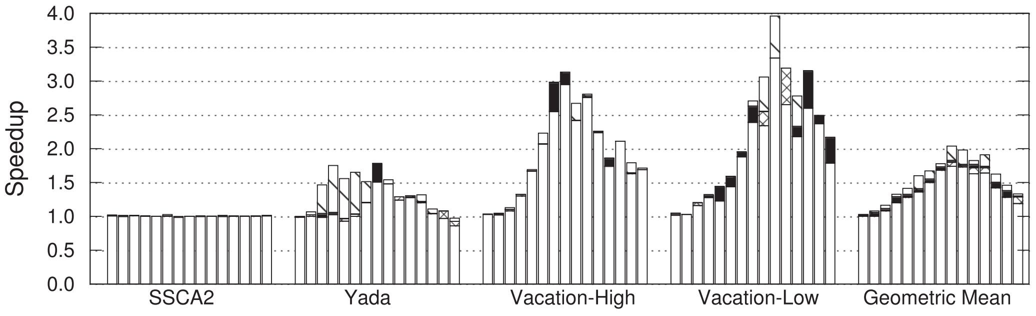
Enforcing Transaction Scheduling. A limitation of the current implementation of Seer is that its locking strategy, based on the use of mutual exclusion locks, may generate overly conservative scheduling policies. This issue can be illustrated considering the following example scenario. Assume that two transactions, say, T_1 and T_3, are to be prevented from running concurrently with T_2; that is, they acquire in the *Acquire-tx-locks* function the lock associated with T_2 (line 68). Because of this, T_1 and T_3 will also be prevented from executing concurrently. We have opted for the current design because of its simplicity and low overhead. Yet, investigating alternative approaches that addressed this current limitation of Seer is an interesting open research question. For instance, one could use a matrix of locks of size equal to num_tx^2 (where num_tx is the number of transaction types of the application), where each lock would be associated with a pair of transaction types. This approach would allow, in the previous example, T_1 and T_3 to acquire different locks,



(a) STMBench7 workloads.



(b) STAMP benchmarks (1/2).



(c) STAMP benchmarks (2/2).

Fig. 13. 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.

and require T_2 to check *num_tx* different locks in line 78. Clearly, though, such a solution would imply a larger space and time overhead than the solution currently employed by Seer.

Updating the Scheduling Policy. We note that the current approach, which relies on a single thread to update the lock matrix (in line 73), is effective if the application level threads execute transactions at approximately the same frequency—which is actually the case for all the applications that we evaluated in this work. However, the current scheme would clearly be inadequate if the first core of the processor (the one designated in the current Seer's implementation) was to run few transactions. A more robust, although slightly more sophisticated, approach would consist of delegating the update of the lock matrix to threads that blocked on the single-global lock.

These could check if the lock scheme was updated too long ago (e.g., using the Intel 64 RDTSC instruction) and, if so, acquire a lock protecting the update process, update the metadata and time, and release the lock. This would occur off the critical path (since these threads were waiting for the single-global lock anyway) and, hence, is expected to incur negligible cost.

Optimality of the Scheduling Policy. Finally, one may argue that an ideal scheduler would ensure that performance plateaus (but does not drop) when the system is brought to operate beyond its optimal degree of parallelism.

With fewer than 14 threads, in several workloads, Seer approximates reasonably well this ideal behavior (see, e.g., Figures 6(a) and 5(f)). But there are also a few notable exceptions. Kmeans-high (Figure 5(c)) is probably one of the clearest cases: performance scales up to six threads and then drops visibly. As discussed in Section 6.1, this benchmark, due to its very short duration, does not allow Seer to devise an effective locking scheme. STMBench7 read-write balanced and write-dominated workloads show, again when the thread count is lower than 14, a similar, although more reduced, decay of performance beyond the optimal degree of parallelism, which in these workloads is around two to three threads. Unlike Kmeans-high, STMBench7 workloads are long running. However, this benchmark has a much larger number of different transaction types, which increases significantly the complexity of identifying a perfect locking strategy that could effectively approximate the performance of an ideal scheduler.

Above 14 threads, the CPU employed in our tests starts using hyperthreading, which is known to hamper performance of some HTM-based applications (Diegues et al. 2014). An ideal scheduler could instantaneously and accurately detect whether it would be beneficial to prohibit hyperthreading by exploiting the core locks mechanism. This would ensure robust performance even above 14 threads. While Seer does mitigate the performance penalty caused by operating in hyperthreading mode with respect to RTM (see Figure 7), it still incurs a nonnegligible performance drop. We argue that this depends on the policy used by Seer to decide whether to acquire core locks, that is, upon a capacity exception. This approach has the advantage of being very simple and lightweight, and it does provide nonnegligible speedups with realistic workloads (see Figure 13). However, this approach is far from accounting for all the complex factors that can affect the performance of the system when operating in hyperthreading mode. Further, this heuristic can suffer from "false positives," in the sense that capacity exceptions could not be caused by the use of hyperthreading, but simply by the excessive length of the transaction footprint.

Overall, although Seer was shown to provide strong performance benefits in a broad range of workloads, there are several aspects of the logic currently used by Seer to determine the transaction scheduling policy that could be enhanced. A possible research avenue that could be pursued to design more efficient schedulers might consist of exploiting additional information extracted via code analysis techniques regarding (1) the likelihood of conflicts between different transaction types (e.g., using static analysis techniques analogous to the ones employed by Riegel et al. (2008) to automatically identify memory partitions accessed by different sets of transactions) and (ii) the size of the transaction footprint. The former information could be exploited to reduce the search space explored using Seer's probabilistic inference scheme, for example, by ruling out pairs of transaction types that have no or little probability of conflicting. In fact, narrowing the search space would inherently reduce the complexity of the problem, and, hence, it could increase both the convergence speed of the inference scheme and its accuracy. The latter could be used to design more informed/accurate heuristics for governing the acquisition of core locks, reducing the risk of false positives. An alternative approach to decide whether to acquire or not core locks (either at the core level or at the level of the whole system) could be based on gathering feedback from the system at runtime using online learning techniques, like hill climbing (Felber et al. 2008), UCB (Diegues and Romano 2015), or Bayesian Optimization (Didona et al. 2016), to guide the exploration of the possible various configurations.

8 CONCLUSIONS

In this article, we presented Seer, the first *fine-grained* scheduler designed to cope with the 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 a dynamic locking scheme that is used to serialize contention-prone transactions in a fine-grained fashion.

We evaluated our solution, against several alternatives and in many scenarios, using one of the largest HTM-enabled processors available from Intel. As a result, we verified that Seer yields average improvements of 65% across the various benchmarks and concurrency degrees, with speedups of up to 3.6×.

REFERENCES

Allon Adir, Dave Goodman, Daniel Hershcovich, Oz Hershkovitz, Bryan Hickerson, Karen Holtz, Wisam Kadry, Anatoly Koyfman, John Ludden, Charles Meissner, Amir Nahir, Randall R. Pratt, Mike Schiffli, Brett St. Onge, Brian Thompto, Elena Tsanko, and Avi Ziv. 2014. Verification of transactional memory in POWER8. In *Proceedings of the Annual Design Automation Conference (DAC'14).* Article 58, 6 pages.

Yehuda Afek, Amir Levy, and Adam Morrison. 2014. Software-improved hardware lock elision. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC'14).*

Mohammad Ansari, Behram Khan, Mikel Luján, Christos Kotselidis, Chris C. Kirkham, and Ian Watson. 2010. Improving performance by reducing aborts in hardware transactional memory. In *Proceedings of the Conference on High Performance Embedded Architectures and Compilers (HiPEAC'10).* 35–49.

Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris C. Kirkham, and Ian Watson. 2009. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *Proceedings of the Conference on High Performance Embedded Architectures and Compilers (HiPEAC'09).* 4–18.

Harold Cain, Maged Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. 2013. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13).* 225–236.

Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. 2010. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the European Conference on Computer Systems (EuroSys'10).* 27–40.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the Symposium on Cloud Computing (SoCC'10).* 143–154.

Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. 2006. Hybrid transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06).* 336–346.

Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. 2008. Applications of the adaptive transactional memory test platform. In *3rd Workshop on Transactional Computing (TRANSACT'08).*

Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2014. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA'14).* 188–197.

Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. 2016. ProteusTM: Abstraction meets performance in transactional memory. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16).* 757–771. DOI: http://dx.doi.org/10.1145/2872362.2872385

Nuno Diegues and Paolo Romano. 2015. Self-tuning Intel restricted transactional memory. In *Elsevier Parallel Computing.*

Nuno Diegues, Paolo Romano, and Stoyan Garbatov. 2015. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*. 224–233.

Nuno Diegues, Paolo Romano, and Luis Rodrigues. 2014. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'14)*. 3–14.

Shlomi Dolev, Danny Hendler, and Adi Suissa. 2008. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC'08)*. 125–134.

Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. 2009a. Stretching transactional memory. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'09)*. 155–165.

Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. 2009b. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC'09)*. 7–16.

Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. 237–246.

Sergio Miguel Fernandes and Joao Cachopo. 2011. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. 179–188.

Keir Fraser and Timothy L. Harris. 2007. Concurrent programming without locks. *ACM Transactions on Computer Systems* 25, 2 (2007).

Vincent Gramoli and Rachid Guerraoui. 2014. Democratizing transactional programming. *Communications of ACM* 57, 1 (Jan. 2014), 86–93.

Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2007. STMBench7: A benchmark for software transactional memory. In *Proceedings of the European Conference on Computer Systems (EuroSys'07)*. 315–324.

Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. 2008. Composable memory transactions. *Communications of the ACM* 51, 8 (2008), 91–100.

Tomer Heber, Danny Hendler, and Adi Suissa. 2012. On the impact of serializing contention management on STM performance. *Journal of Parallel and Distributed Computing* 72, 6 (June 2012), 739–750.

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA'93)*. 289–300.

João Cachopo Hugo Rito. 2015. Adaptive transaction scheduling for mixed transactional workloads. *Elsevier Parallel Computing Journal* 41 (2015), 31–49.

Christian Jacobi, Timothy Slegel, and Dan Greiner. 2012. Transactional memory architecture and implementation for IBM system Z. In *Proceedings of the Symposium on Microarchitecture (MICRO'12)*. 25–36.

Andi Kleen. 2014. Scaling existing lock-based applications with lock elision. *Communications of the ACM* 57, 3 (March 2014), 52–56.

Yossi Lev, Mark Moir, and Dan Nussbaum. 2007. PhTM: Phased transactional memory. In *2nd Workshop on Transactional Computing (TRANSACT'07)*.

Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. 2010. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the European Conference on Computer Systems (EuroSys'10)*. 41–54.

Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. 2010. Scheduling support for transactional memory contention management. In *Proceedings of Principles and Practice of Parallel Programming (PPoPP'10)*. 79–90.

Maged M. Michael. 2013. The balancing act of choosing nonblocking features. *Communications of ACM* 56, 9 (Sept. 2013), 46–53.

Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the Symposium on Workload Characterization (IISWC'08)*. 35–46.

Mohamed Mohamedin, Roberto Palmieri, and Binoy Ravindran. 2015. Brief announcement: On scheduling best-effort HTM transactions. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*. 74–76.

Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. 2015. Quantitative comparison of hardware transactional memory for blue gene/Q, zenterprise EC12, intel core, and POWER8. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*. 144–157.

Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. 2008. Design and implementation of transactional constructs for C/C++. In *Proceedings of the Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA'08)*. 195–212.

Victor Pankratius and Ali-Reza Adl-Tabatabai. 2014. Software engineering with transactional memory versus locks in practice. *Theory of Computer Systems* 55, 3 (Oct. 2014), 555–590.

Ravi Rajwar and James R. Goodman. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the Symposium on Microarchitecture (MICRO'01)*. 294–305.

Torvald Riegel, Christof Fetzer, and Pascal Felber. 2008. Automatic data partitioning in software transactional memories. In *Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPA'08)*. 152–159. DOI: http://dx.doi.org/10.1145/1378533.1378562

Hugo Rito and Joao Cachopo. 2014. ProPS: A progressively pessimistic scheduler for software transactional memory. In *Proceedings European Conference on Parallel Processing (Euro-Par'14)*. 150–161.

Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. 2007. TxLinux: Using and managing hardware transactional memory in an operating system. In *Proceedings of Symposium on Operating Systems Principles (SOSP'07)*. 87–102.

Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. 2010. Is transactional programming actually easier? *SIGPLAN Notifications* 45, 5 (Jan. 2010), 47–56.

Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. 2012. Evaluation of blue gene/Q hardware support for transactional memories. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. 127–136.

Lingxiang Xiang and Michael L. Scott. 2015. Conflict reduction in hardware transactions using advisory locks. In *Proceedings of Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*. 234–243.

Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'07)*. 261–272.

Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*. 1–11.

Richard M. Yoo and Hsien-Hsin S. Lee. 2008. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA'08)*. 169–178.

Received January 2016; revised March 2017; accepted August 2017