HaTS: Hardware-assisted Transaction Scheduler

Zhanhao Chen
Palo Alto Networks Inc, USA
zhachen@paloaltonetworks.com

Ahmed Hassan
Lehigh University, USA
ahmed.hassan@lehigh.edu

Masoomeh Javidi Kishi
Lehigh University, USA
maj717@lehigh.edu

Jacob Nelson
Lehigh University, USA
jjn217@lehigh.edu

Roberto Palmieri
Lehigh University, USA
palmieri@lehigh.edu

Abstract

In this paper we present HaTS, a Hardware-assisted Transaction Scheduler. HaTS improves performance of concurrent applications by classifying the executions of their atomic blocks (or in-memory transactions) into scheduling queues, according to their so called conflict indicators. The goal is to group those transactions that are conflicting while letting non-conflicting transactions proceed in parallel. Two core innovations characterize HaTS. First, HaTS does not assume the availability of precise information associated with incoming transactions in order to proceed with the classification. It relaxes this assumption by exploiting the inherent conflict resolution provided by Hardware Transactional Memory (HTM). Second, HaTS dynamically adjusts the number of the scheduling queues in order to capture the actual application contention level. Performance results using the STAMP benchmark suite show up to 2x improvement over state-of-the-art HTM-based scheduling techniques.

2012 ACM Subject Classification Software and its engineering → Scheduling

Keywords and phrases Transactions, Scheduling, Hardware Transactional Memory

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.31

Funding This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367 and by the National Science Foundation under Grant No. CNS-1814974.

1 Introduction

Without reservation, in-memory transactions have experienced a significant growth in adoption during the last decade. Specifically, the advent of Hardware Transactional Memory (HTM) support in commodity processors [27, 7, 16, 30] has changed the way concurrent programs' execution is handled, especially in terms of performance advantages. Whether a multi-threaded application implements atomic blocks using locks or transactions, HTM can be exploited in both cases (e.g., using Hardware Lock Elision [26] in the former case or Restricted Transactional Memory [27] in the latter case) to accelerate its performance.

Hardware transactions are significantly faster than their software counterpart because they rely on the hardware cache-coherence protocol to detect conflicts, while Software

23rd International Conference on Principles of Distributed Systems (OPODIS 2019).
Editors: Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller; Article No. 31; pp. 31:1–31:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Transactional Memory (STM) [18] adds a significant overhead of instrumenting shared memory operations to accomplish the same goal [9]. Relying on the cache-coherence protocol also makes HTM appealing for mainstream adoption since it requires minimal changes in hardware. However, this inherent characteristic of HTM represents an obstacle towards defining contention management and scheduling policies for concurrent transactions, which are crucial for both progress and fairness of HTM execution in the presence of conflicting workloads. In fact, most TM implementations achieve high concurrency when the actual contention level is low (i.e., few transactions conflict with each other). At higher contention levels, without efficient scheduling, transactions abort each other more frequently, possibly with a domino effect that can easily lead to performance similar to, if not worse than, sequential execution [22, 21, 8].

A *Contention Manager* (CM) is the traditional, often encounter-time technique that helps in managing concurrency. When a transaction conflicts with another one, the CM is consulted to decide which of the two transactions can proceed. A CM collects statistics about each transaction (e.g., start time, read/write-sets, number of retries, user-defined parameters) and decides priorities among conflicting transactions according to the implemented policy. Schedulers are similar to CMs except that they may proactively defer the execution of some transactions to prevent conflicts rather than react to them. In both cases, performance is improved by decreasing abort rate and fairness is achieved by selecting the proper transaction to abort/defer [20, 31, 30].

The conflict resolution strategy of current off-the-shelf HTM implementations is provided entirely in hardware, and can be roughly summarized as follows:

- the L1 cache of each CPU-core is used as a buffer for transactional read and write operations[1];
- the granularity of conflict detection is the cache line; and
- if a cache line is evicted or invalidated, the transaction is aborted (reproducing the idea of read-set and write-set invalidation of STM [11]).

The above strategy thus implies a requester-wins contention management policy [6], which informally means that a transaction T_1 aborts another transaction T_2 if T_2 performed an operation on a memory location that is physically stored in the same cache line currently requested by T_1, excluding the case of two read operations, which never abort each other. Due to this simple policy, classical CM policies cannot be trivially ported for scheduling HTM transaction mainly because of two reasons. First, transactions are immediately aborted when one of the cache lines in their footprint is invalidated, which makes it too late for CM to avoid conflicts or manage them differently (e.g., by deciding which transaction is more convenient to abort). Second, it is hard to embed additional metadata to monitor transactions behavior, since all reads and writes executed within the boundaries of transactions are considered transactional, even if the accessed locations store metadata rather than actual data.

In this paper we introduce HaTS (*Hardware-assisted Transaction Scheduler*), a transaction scheduler that leverages the unique characteristics of HTM to accelerate scheduling in-memory transactions. To overcome the aforementioned limitations of HTM, HaTS neither aims at altering HTM's conflict resolution policy nor adds metadata instrumentation inside hardware transactions, but instead relies on it to relax the need for the scheduler to define a non-conflicting schedule. HaTS effectively arranges incoming transactions according to a set of metadata collected either at compilation time (leveraging developer's annotations) or at run time (after transactions commit/abort). HTM is then used to execute transactions

[1] In some HTM implementation [27], reads can be logged on the L2 cache to increase capacity.

concurrently while maintaining atomicity, isolation, and performance.

In a nutshell, HaTS works as follows. It uses a software classifier to queue incoming transactions with the goal of allowing only those transactions that do not conflict with each other to execute concurrently. The fundamental innovation of HaTS, which makes it practical, is that it works with incomplete or even erroneous information associated with incoming transactions. This is because even if the classifier erroneously allows conflicting transactions to run concurrently, HTM will inherently prevent them from committing (at least one of the conflicting transactions will abort). Therefore, misclassifications cannot impact the correctness of the transactional execution.

More in detail, HaTS offers a set of *scheduling queues* to group conflicting transactions. Membership of a queue is determined based on a single metadata object associated with each transaction, called *conflict indicator*. A conflict indicator might be provided by the programmer (e.g., the address of a contended memory location accessed transactionally) or computed by the system (e.g., transaction abort rate).

A queued transaction waits until the scheduler signals it when it becomes top-standing in its respective queue. When the transaction actually executes, the built-in HTM takes care of possible conflicts with transactions dispatched from other queues due to misclassification, which also includes the case where no conflict indicator is provided.

Another key feature of HaTS is that it adapts the number of scheduling queues based on a set of global statistics, such as the overall number of commits and aborts. This adaptation significantly improves performance in two common, and apparently dual cases. On the one hand, since conflict indicators are mainly *best effort* indicators, a single per transactions conflict indicator will not suffice when the application workload is highly conflicting. For this reason, HaTS reduces the number of queues when the overall conflict level increases, *enforcing transactions with different conflict indicators to be executed sequentially*. On the other hand, if the overall conflict level significantly decreases, HaTS increases the number of queues in order to (re-)allow transactions with different conflict indicators to execute concurrently. Additionally, when conflict level remains low, it enables dispatching multiple transactions from the same queue simultaneously, *allowing transactions with the same conflict indicator to execute in parallel*. Our framework aims at adaptively converging on an effective configuration of scheduling queues for the actual application workload. By leveraging HTM and its built-in atomicity guarantees, transitions between different configurations do not entail stalling transaction executions.

We implemented HaTS in C++ and integrated it into the software framework of SEER [13]. We contrasted HaTS performance against Hardware Lock Elision (HLE) [26], Restricted Transactional Memory (RTM) [27], Software-assisted Conflict Management (SCM) [1], and SEER itself. We used the STAMP suite as our benchmark and we used a testbed of four-socket Intel platform with HTM implemented through TSX-NI. Results, including speedup over the sequential non-instrumented code and two abort rate metrics, show that HaTS outperforms competitors in both high contention and low contention scenarios. This stems from both leveraging conflict indicators and performing dynamic adjustment of scheduling queues, which leads to a notable performance improvement (e.g., 2x speedup in execution time for `Kmeans` and a 50% improvement for the `Vacation` benchmarks).

The rest of this paper is structured as follows. We review previous work in Section 2 and the limitations of the current HTM implementations in Section 3. The design and implementation details of HaTS are presented in Section 4. We compare the performance of HaTS against state-of-art competitors in Section 5, and we conclude our paper in Section 6.

2 Related Work

TM schedulers can be classified based on whether they target STM systems [28, 15, 14, 5] or HTM systems [13, 25, 1, 4, 29, 31]. Although we deploy and test HaTS in an HTM-based environment, due to its performance advantages, HaTS can be deployed in STM systems as well.

Among the HTM-based schedulers, the closest one to HaTS is SEER [13]. SEER's main idea is to infer the probability that two atomic blocks conflict, basing its observation on the commit/abort pattern witnessed in the past while the two were scheduled concurrently. Thus, HaTS and SEER are similar in their best-effort nature: both of them do not require precise information on the pair of conflicting transactions nor on the memory location(s) where they conflict, which is the key to coping with the limitations of current HTM implementations.

HaTS differs from SEER in two core points. First, SEER focuses only on managing HTM limitations, and thus it schedules transactions based on their commit/abort patterns. On the other hand, HaTS is a generic scheduler that uses HTM to preserve atomicity and consistency, and thus it defines a generic *conflict indicator* object that can embed both online and offline metadata. Second, SEER adopts a fine-grained (pairwise) locking approach to prevent transactions that are more likely to conflict from running concurrently. HaTS replaces this fine-grained locking scheme with a lightweight queueing scheme that controls the level of conflict by increasing/decreasing the number of scheduling queues. Our experimental results in Section 5 show that this lightweight scheme results in a lower overhead in different scenarios, as the case of Vacation where the percentage of committed transactions in HTM is comparable with SEER's but overall application execution time is about 50% faster.

The work in [1] proposes a Software-assisted Conflict Management (SCM) extension to HTM-based lock elision (HLE) [26], where aborted transactions are serialized (using an auxiliary lock) and retried in HTM instead of falling back to a slow path with a single global lock. The main advantage of this approach is avoiding the lemming effect that causes new (non-conflicting) transactions to fall back to the slow path as well. As opposed to HaTS, SCM uses a conservative scheme where all aborted transactions are serialized without any further (even imprecise) conflict indicators, which limits concurrency. Moreover, SCM does not leverage the observed conflict pattern to proactively prevent conflicts in the future.

The idea of using scheduling queues to group conflicting transactions has been briefly discussed in [25], where authors introduced the concept of Octonauts. Octonauts uses statistics from transactions that committed in HTM to speculate over the characteristics of the associated transaction profile for future classification. HaTS is an evolution of Octonauts where a comprehensive software infrastructure, along with conflict indicators and a dynamic scheduling queuing techniques have been used to improve application performance.

A few other HTM-based schedulers were proposed prior to the release of Intel TSX extensions [27]. However, they either assume HTM implementations that are different from the currently deployed hardware [4, 29] or rely on a conservative single-lock-based serialization scheme similar to SCM [31].

STM-based schedulers rely on more precise information about transactions conflict. ProPS [28] uses a probabilistic approach similar to SEER but with a precise knowledge of the pair of conflicting transactions. Shrink [15] additionally uses the history of recently completed transactions' read/write-sets to predict conflicts. CAR-STM [14] and SOA [5] use per-core queues such that an aborted transaction is placed in the queue of the transaction that causes it to abort. The necessity of precise information represents an obstacle towards adopting such techniques in HTM-based systems.

3 Background: Scheduling Best-effort Hardware Transactions

HTM provides a convenient concurrent programming abstraction because it guarantees safe and efficient accesses to shared memory. HTM executes atomic blocks of code optimistically, and during the execution all read and write operations to shared memory are recorded in a per-transaction log, which is maintained in a thread-local cache. Any two operations generated by two concurrent transactions accessing memory mapped to the same cache line trigger the abort of one of the transactions. HTM is known to have limited progress guarantees [3, 8, 24]. To guarantee progress, all HTM transactions are guaranteed to commit after a number of retries as HTM transactions by exploiting the traditional software lock-based fallback path [27]. To implement that, hardware transactions check if the fallback lock is acquired at the beginning of their execution. If so, the transactional execution is retried; otherwise the execution proceeds in hardware and mutual exclusion with the fallback path is implemented by leveraging the strong atomicity property of HTM, which aborts any hardware execution if the fallback lock is acquired at any moment. To reduce the well-known lemming effect [10] in HaTS, a transaction is not retried in HTM until the global lock is released.

For simplicity, in the rest of the paper we refer to HTM execution as the above process, which encompasses hardware trials followed by the fallback path, if needed. It is important to note that, since HaTS does not assume a specific methodology to provide HTM with progress, more optimized alternative solutions [22, 8, 12] can be integrated into our HTM execution to improve performance even further.

The off-the-shelf HTM implementation only provides limited information about reasons behind aborted transactions, which makes it very hard for programmer to introduce modifications that would increase the likelihood for that transaction to commit. As a result, in the presence of applications with contention, HTM might waste many CPU cycles until a transaction can succeed by either retrying multiple times, or by falling back to a single global lock where the protected HTM execution can be relaxed in favor of the mutual exclusion implemented by the lock itself.

Contention management for practical transaction processing systems is often formulated as an online problem where metadata, in the form of statistics (e.g., actual access pattern), can be collected by aborted and committed transactions in order to fine-tune scheduling activities. However, this methodology cannot be directly ported to HTM-protected concurrent executions since HTM cannot distinguish between a cache line that stores actual application data, or scheduling metadata. Because of that, conflicting accesses to shared metadata executed by two concurrent hardware transactions may cause at least one of them to abort, even if at the semantic level no conflict occurred.

The above issues motivated us to design a transaction scheduler where HTM is exploited as-is, instead of providing software innovations or hardware extensions aimed at influencing the HTM conflict resolution mechanism [2], which likely lead to degradation of HTM effectiveness.

4 Hardware Transaction Scheduler

In this section we overview the two core components of HaTS, namely the transaction conflict indicator (Section 4.1) and the dynamic scheduling queues (Section 4.2), along with a description of the complete transaction execution flow (Section 4.3) and the details of how threads execution passes through the scheduling queues (Section 4.4).

Terminology. HaTS has a set of N concurrent queues, called *scheduling queues.* Each thread that is about to start a transaction (i.e., an atomic block) is mapped to one of those scheduling queues, and it starts executing only when HaTS dispatches it from that queue. Each scheduling queue has one (or more) *dispatcher thread*(s). As we detail later, the mapping between each transaction and its corresponding scheduling queue is based on the transaction's *conflict indicator* and the mapping is implemented using hashing. The overall transaction commit/abort statistics are collected by HaTS and recorded into a shared *knowledge base*. HaTS periodically consults the knowledge base to increase/decrease the number of scheduling queues or the number of dispatcher threads, dynamically.

4.1 Transaction Conflict Indicator

HaTS uses a so called *transaction conflict indicator*, provided as a parameter to `TM-BEGIN` in our experimental study, to represent in a compact way characteristics that affect the probability of aborting a hardware transaction due to conflict. Having this information is indeed powerful because it allows HaTS to group transactions that access the same system's hot spot in the same conflict queue, which saves aborts and increases throughput.

The transaction conflict indicator is an abstraction that can be deployed in many different ways. A simple and effective example of conflict indicator is the address of the memory location associated with the accessed system hot spot. As a concrete example in a real application, let us consider a monetary application where transactions work on given bank accounts. A transaction would use the address of the accessed bank account, which uniquely identifies that object (or memory location) in the system, as its conflict indicator. This way, although transactions might still (occasionally) conflict on other shared memory elements, HaTS will be able to prevent conflicts between accesses to the same bank account, which is one of the most contended set of objects in the system. Because of its effectiveness, in our evaluation study we focused on system hot spots as the transaction conflict indicator.

Other examples of transaction conflict indicators include:

- *Abstract data types of accessed objects:* transactions accessing (possibly different) objects of the same abstract data type will have the same conflict indicator. This represents a more conservative approach than our adopted (per-object) conflict indicator, and it can work better in workloads with higher contention levels.
- *HLE fallback lock(s):* if hardware transactions are used for lock elision (HLE) [26], the fallback paths of HTM transactions acquire the elided locks rather than a single global lock, as in RTM. Using this fallback lock as a conflict indicator, transactions that elide the same lock are grouped together.
- *Profile of aborted transactions:* HaTS's knowledge base can record the profile identification of aborted transactions within a window of execution, and group incoming invocations of those transactions using a single conflict indicator. This can significantly reduce abort rate because those transactions are more prone to conflict in the future as well. To avoid unnecessary serialization, the knowledge base can record only transactions aborted due to conflict, and exclude transactions aborted for any other reason (i.e., capacity and explicit aborts).
- *Core ID:* transactions running on the same physical core are assigned the same conflict indicator. This can be beneficial because multiple hardware threads running on the same physical core share the same L1 cache, which means that transactions concurrently invoked by those threads are more prone to exceed cache capacity and abort.

The last two points reflect similar ideas already used in literature in different ways [13, 1, 28, 15, 14, 5]. Although we refer to Section 2 for a detailed comparison of these approaches with HaTS, it is worth to mention here that HaTS's innovations relies on the fact that it deploys those ideas in an abstract way, using conflict indicators, which allows for a better scheduling management.

HaTS allows for the specification of a single conflict indicator per transaction. Although a single indicator might seem limited in terms of expressiveness, we adopt this approach because of the following reasons. First, there will always be a trade-off between the precision achieved by allowing multiple indicators and the additional cost needed to analyze them. Our decision is the consequence of an empirical study, that we excluded for space limitations, where we analyzed this trade-off. Second, the way HaTS adapts the number of scheduling queues (as detailed in the next section) is a dominant factor to manage contention that mitigates the effect of having imprecise, yet lightweight, conflict indicators. Finally, it is still possible to extend HaTS's infrastructure to support multiple indicators. For example, Bloom Filters can be used to compact multiple indicators and bit-wise operations (using either conjunction or disjunction operators) can be used to hash each bloom filter to the corresponding queue. As a future work, we plan to study the trade-off mentioned above; however, our current evaluation shows that even with a single indicator, HaTS outperforms existing approaches.

4.2 Dynamic Distribution of Scheduling Queues

Mapping transaction conflict indicators to scheduling queues is critical for achieving the goal of HaTS because it guarantees that transactions with the same conflict indicators are grouped in the same queue. However, using a static number of scheduling queues in such a mapping might lead to issues such as unbalancing, unfair scheduling, and poor adaptivity to some application workloads. For this reason, HaTS deploys a dynamic number of scheduling queues to cope with application workloads and effectively provide an elastic degree of parallelism. As we detailed in Section 4.3, this number is decided at run time according to the overall commit/abort statistics calculated in HaTS's knowledge base.

The following two examples clarify the need for having a dynamic set of scheduling queues. First, consider two incoming transactions with different conflict indicators. Since we have a finite number of scheduling queues, it is possible that those two transactions are mapped to the same queue. When the number N of queues is increased, the probability of mapping those transactions to the same queue decreases, and the level of parallelism increases. Second, consider a workload where transactions are highly conflicting so that the conflict indicator is not sufficient to capture all raised conflicts. In this case, decreasing the number of queues reduces parallelism and potentially reduces abort rate.

Adaptively changing the number of queues also covers more complex, yet not uncommon, cases. For example, it covers the case when transactions' data access pattern is hard to predict; therefore having a single conflict indicator per transaction may not be sufficient (e.g., when each transaction accesses multiple system hot spots). Also, it covers the cases when no effective conflict indicator can be defined but the workload experiences high abort rates due to other reasons (e.g., aborts due to false sharing of the same cache lines). Finally, it allows schedulers to temporarily disable the usage of conflicting indicator as a medium for grouping transactions, in favor of a random policy, without hindering performance.

As will be clear in Section 4.3, dynamically changing the number of scheduling queues neither introduces blocking phases nor trades off correctness, thanks to the underlying HTM.

4.2.1 Multiple Dispatchers

An interesting example that is not covered by the aforementioned policy is when transactions with the same conflict indicator (and hence grouped in the same queue) are actually able to execute concurrently. Although it may appear as an infrequent case, we recall that conflict indicators are best-effort indicators that can be imprecise. Also, since conflicts are raised at runtime according to the transaction execution pattern, it may happen that two conflicting concurrent transactions succeed to commit even if they run concurrently (e.g., one of them commits before the other one reaches the conflicting part).

HaTS addresses this case by allowing multiple dispatcher threads for a single scheduling queue. Similar to the way we increase/decrease the number queues, we use abort rate as an indicator to increase/decrease the number of dispatchers per queue. For additional fine-tuning, we allow programmers to statically configure the number of scheduling queues. In that sense, transactions with the same conflict indicator are executed in parallel only if the overall contention level is low.

4.3 Transaction Execution Flow

Transactional operations are executed directly by application threads, without relying on designated worker threads managed by HaTS. In fact, HaTS's role is to dispatch thread executions.

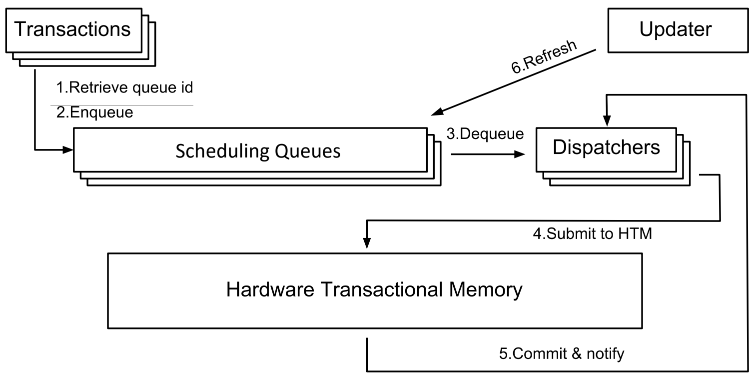

Figure 1 HaTS software architecture and high-level threads execution flow.

HaTS restricts transactions mapped to the same scheduling queue to run sequentially, while offloading the concurrency control handling to the underlying HTM engine. Figure 1 shows the execution flow of a transaction T executed by an application thread T_r. T_r first hashes the *conflict indicator* of T (using `module` N hashing, where N is the current number of scheduling queues) in order to find the matching scheduling queue Q (Step 1). After that, T_r effectively *suspends* its execution by enqueuing itself into Q (Step 2) and waiting until a dispatcher thread *resumes* its execution (Step 3).

HaTS provides one (or more) dispatcher thread T_Q per conflict queue Q. Each dispatcher thread resumes one waiting transaction execution at a time, in a closed-loop manner, meaning the next queued transaction execution is resumed only after the previous one is successfully committed. For the sake of fairness, each queue is implemented as a priority queue, where the priority of each transaction is proportional to the number of aborts its enclosing atomic block experienced in former executions (similar to the approach used in SEER [13] to infer the conflict pattern between atomic blocks.).

When a thread execution is resumed, the corresponding transaction starts to execute leveraging the HTM implementation (Step 4). During the hardware transaction execution, T_Q waits until T_r completes its transactional execution. After T_r's commit, T_Q takes control

(Step 5) and performs two operations: it updates the *knowledge base* with the needed information (namely the number and types of aborts before committing); and it dispatches the next thread execution waiting in Q.

A special background thread, called *updater*, is used to dynamically change the number of scheduling queues depending upon the effectiveness of the parallelism achieved by the current scheduling queues configuration. To do so, the updater thread queries the knowledge base and decides, according to the transaction abort rate measured so far, whether the total number N of scheduling queues should be increased, decreased, or unchanged (Step 6). In our implementation, we adopt a simple hill-climbing approach similar to the one used in [17, 12]. Briefly, if the observed abort rate is greater (less) than the last observed rate, we decrease (increase) the number of queues by one. The maximum number of queues is set to the number of physical cores and the minimum is set to one. We also allow programmers to override this policy by setting a fixed number of scheduling queues in order to eliminate the overhead of this dynamic behavior, especially when the most effective configuration is known. Interestingly, as we show later in our experimental results, this simple approach pays off in most of the tested benchmarks. As a future work, we plan to investigate more complex approaches, such as collecting more detailed information (e.g., the types of aborts) an use reinforcement learning to reach better estimates.

Changing the scheduling queues configuration does not cause stalls of the transaction execution and does not affect execution correctness. This is a great advantage of leveraging HTM conflict detection. Let us consider two incoming transactions T_1 and T_2 that in the current scheduling queues configuration would map to the same scheduling queue Q_1. Let us assume that the scheduling queues configuration changes after T_1 is enqueued in Q_1 and before T_2 is enqueued. In this case, there is the possibility for T_2 to be mapped to another queue Q_2 in the new configuration, which ends up having T_1 and T_2 in two different queues (even though it might be the case they were both to be mapped to the same queue Q_2 in the new configurations). Although one can consider this scenario as an example of misclassification of incoming transactions due to ongoing change of N, safety cannot be affected because of leveraging the HTM execution. Even if those transactions are conflicting, they will be correctly serialized by (likely) enforcing one of them to abort and fallback to HTM's global locking phase.

4.4 Suspending/Resuming Executions with Scheduling Queues

As we mentioned in the previous section, a thread that wants to execute a transaction suspends and enqueues its execution until a dispatcher thread of the mapped queue resumes it. In order to synchronize this suspend/resume process, we use two synchronization flags[2], one handled by the application thread and the other handled by the dispatcher thread.

Figure 2 pictures the synchronization scheme between an application thread T_r performing a transaction T and a dispatcher thread T_Q responsible for handling the scheduling queue Q that matches T's conflict indicator. Numbers represent the sequence of operations completion.

When T_r wants to execute T, it creates a flag object L^{T_r} initialized to 0 and (atomically) enqueues it in Q, which effectively suspends its execution. After that, T_r spins until L^{T_r} is set to 1. When L^{T_r} becomes top standing in Q, T_Q dequeues it. Then, T_Q sets a flag associated with Q, called L^Q, and also sets L^{T_r} to 1 (in that order). By setting L^{T_r}, T_r will be signaled to proceed with its HTM execution. By setting L^Q, T_Q is suspended until the

[2] Flags are implemented as `volatile` shared memory locations.

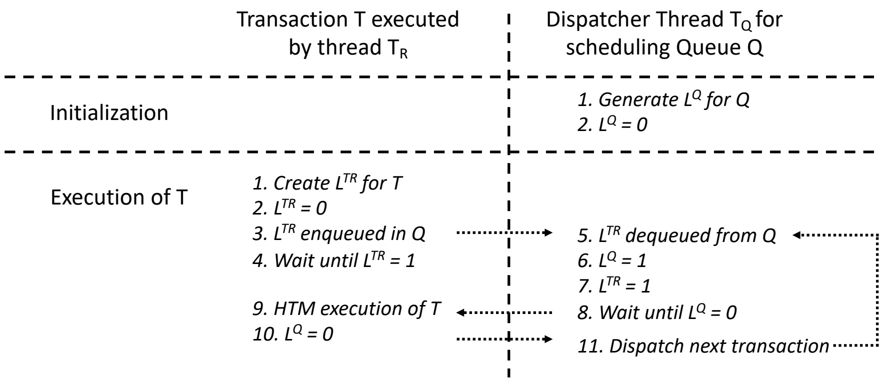

Figure 2 Synchronization between application threads and dispatcher threads. For simplicity, the example accounts for a single dispatcher thread per scheduling queue.

completion of T by T_r. This suspension is implemented by spinning over the L^Q flag. When T is committed, T_r resets L^Q so that T_Q can dequeue the next thread execution waiting on Q. Note that T_Q is not notified if T is aborted and restarted for another HTM trial or if T's execution falls back to the software path. T_Q is resumed only after T's successful completion.

In our implementation we use simple flags to synchronize two threads (i.e., application thread and dispatcher thread) because we deploy one dispatcher thread for each scheduling queue. As introduced earlier, HaTS allows for multiple dispatcher threads per queue in order to cope with the case where the mapping between conflict indicators and scheduling queues is unnecessarily unbalanced, meaning many transactions, possibly with different conflict indicators, are grouped on the same scheduling queue. In the case where multiple dispatcher threads are deployed per conflict queue, the same synchronization scheme illustrated before applies, with the following differences. First, L^{T_r} flags should be atomically set (e.g., using a Compare-And-Swap operation) to synchronize between dispatcher threads. Also, multiple L^Q flags, one per dispatcher thread, are needed to signal each dispatcher thread that it may proceed to schedule the next transaction.

Scheduling queues are implemented in a lock-free manner [19] in order to speed up the thread execution's suspension step. Also, in the above description we simplified the presentation by saying that application threads and dispatcher threads spin over flags to suspend/resume their execution. In the actual implementation, threads yield their execution in order to let computing resources available so that the machine can be oversubscribed (if needed) to maximize CPU utilization.

5 Evaluation

HaTS is implemented in C++ and integrated into the software framework of SEER [13]. An advantage of using a unique software architecture for all competitors is that independent optimization of low-level components does not bias performance towards some implementation. In other words, the performance differences reported in our plots are due to the algorithmic differences between competitors.

Our goal is to assess the practicality of HaTS in standard benchmarks for in-memory transactional execution. Hence, we used STAMP [23], a benchmark suite of eight concurrent applications that span several application domains with different execution patterns. Due to space limitations, we present in detail the results of two applications, `Kmeans` and `Vacation`, since they cover two different and important cases in which the characteristics of HaTS are highlighted. Then, we summarize the results with the other applications.

We compare HaTS against the scheduling techniques provided in the SEER framework, which are (in addition to SEER itself): Hardware Lock Elision (HLE) [26], Restricted Transactional Memory (RTM) [27], and Software-assisted Conflict Management (SCM) [1]. Shortly, SEER is the state-of-the-art probability-based scheduling mechanism that uses fine-grained (pairwise) locks to prevent transactions that are likely to conflict from executing concurrently. HLE transforms atomic blocks protected by locks into hardware transactions and its software fallback path is protected by the original lock itself. In STAMP, a single lock for each atomic block is deployed. RTM supports a configurable number of retries before falling back to a single global lock shared among all hardware transactions. SCM implements a scheduling technique that serializes the aborted transactions to decrease the chance of further aborts. In all implementations, except HLE, transactions try at most five times in hardware before migrating to the software fallback path.

Experiments were conducted using a multi-processor platform equipped with 4 Intel Xeon Platinum 8160 processors (2.1GHz, 24 cores per CPU). The machine provides 96 physical cores and a total of 768 GB of memory divided into 4 NUMA zones. In our experiments we ran up to 64 application threads to leave resources for dispatcher threads (one per queue) and the updater thread. The maximum number of scheduling queues is statically set to 30 prior execution, and we used the default operating system policy to map application threads to cores.

In Figure 3, we report for each application three performance metrics: (left column) speedup over sequential non-instrumented execution; (middle column) percentage of transactions committed through HTM; (right column) among those committed in HTM, percentage of transactions retried more than one time. Generally, the last two metrics are indicators of the scheduling effectiveness in reducing abort rate. The speedup metric is an indicator of whether such a reduction in abort rate is reflected in overall performance improvement or the scheduling overhead nullifies performance benefits. Performance at one application thread represent the slowdown of the sequential instrumented execution. All results are the average of 10 repeated tests.

The first application is `Kmeans`, a clustering algorithm that groups points into K clusters. Transactions are used by the application to synchronize the concurrent updates to the same cluster's center node. For this reason, we select the address of the center node updated by the transaction as its conflict indicator. We implemented that by passing the address of this center node as a parameter to STAMP's `TM-BEGIN` function. Our main observation is that identifying this conflict indicator allows HaTS to significantly reduce abort rate, reaching up to 1.5x reduction with respect to the closest competitor, and improve performance, reaching up to 2x better speedup over the closest competitor. SEER's probabilistic approach is the second best in terms of abort rate, which means that its approach is still able to capture conflicts, but not as effectively as using HaTS's conflict indicator. Moreover, SEER's speedup significantly decreases with higher number of threads, due to its locking overhead.

HLE does not perform well due to the lemming effect, which is visible as soon as few transactions fall back to locking. RTM is generally better than HLE due to its multiple retries in HTM before falling back to locking. SCM provides the worst performance. This is because the way SCM serializes conflicting transactions does not prevent new conflicts of incoming transactions, as opposed to the proactive scheduling approach, such as the one of HaTS and SEER. Also, the probability SCM executes non-conflicting transactions serially is higher than HaTS because it does not use any conflict indicators.

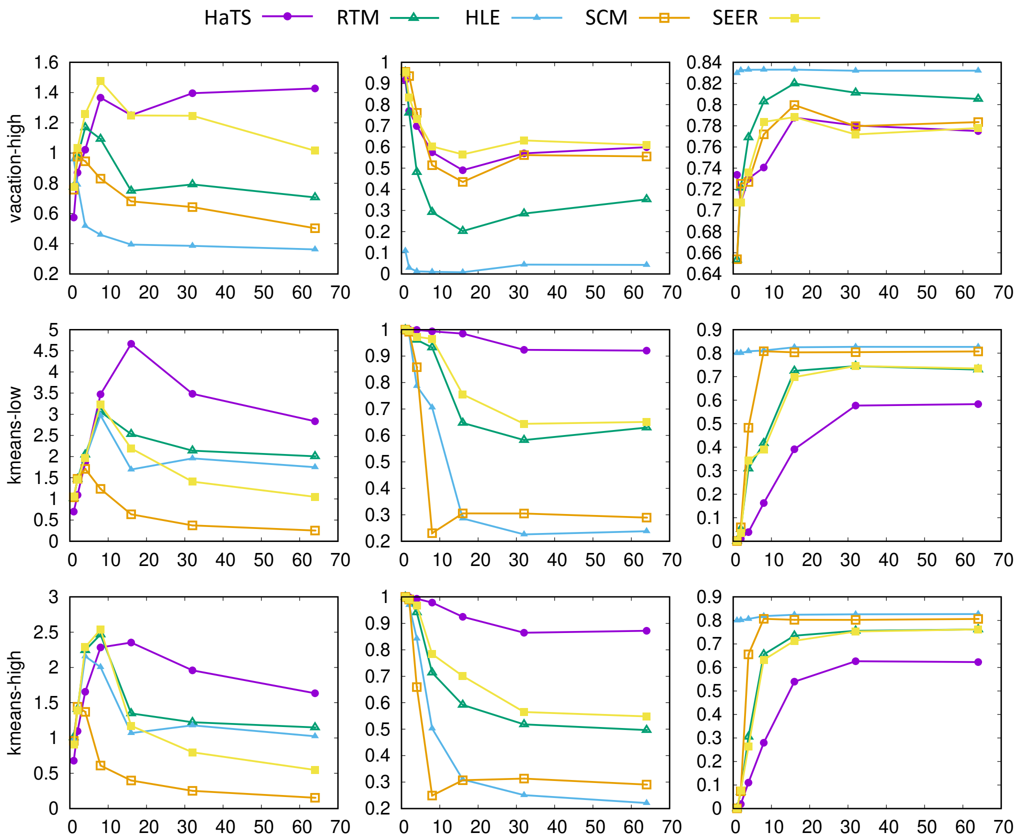

Figure 3 Results with applications of STAMP benchmark. For each row, the left plot shows speedup over sequential (non-instrumented) execution; center plot shows the ratio of committed transactions in HTM; the right plot shows the ratio of transactions retried more than one time. X-axis shows number of application threads.

The difference between the high and low configuration of Kmeans is mainly in the maximum achieved speedup. However, the patterns of the compared algorithms remain the same, which shows the capability of HaTS to gain performance even in relatively low-contention workloads.

The second application is `Vacation`, which simulates an online reservation system. Unlike `Kmeans`, most transactions in `Vacation` apply operations on a set of randomly selected objects, therefore with this pattern it is hard to identify a single conflict indicator per transaction. For that reason, we adopt an approach where each transaction uses a unique conflict indicator, with the exception of transactions that access a single customer object, where we use the customer ID as a conflict indicator. Our rationale behind this decision is that even if transactions are conflicting, HaTS's dynamic adjustment of scheduling queues reduces the level of parallelism and saves aborts. Indeed HaTS achieves an abort rate similar to SEER, and moreover it scales better than SEER.

Summarizing our results of the other STAMP benchmarks (Figure 4) `Intruder` and `Yada` give the same conclusions: the lightweight queuing approach in HaTS allows it to perform better than SEER, especially for high number of threads, due to the overhead of SEER's locking mechanism. `SSCA` and `Genome` are low-contention benchmarks, and their abort rates are very low even without scheduling or contention management. Hence, none of the compared

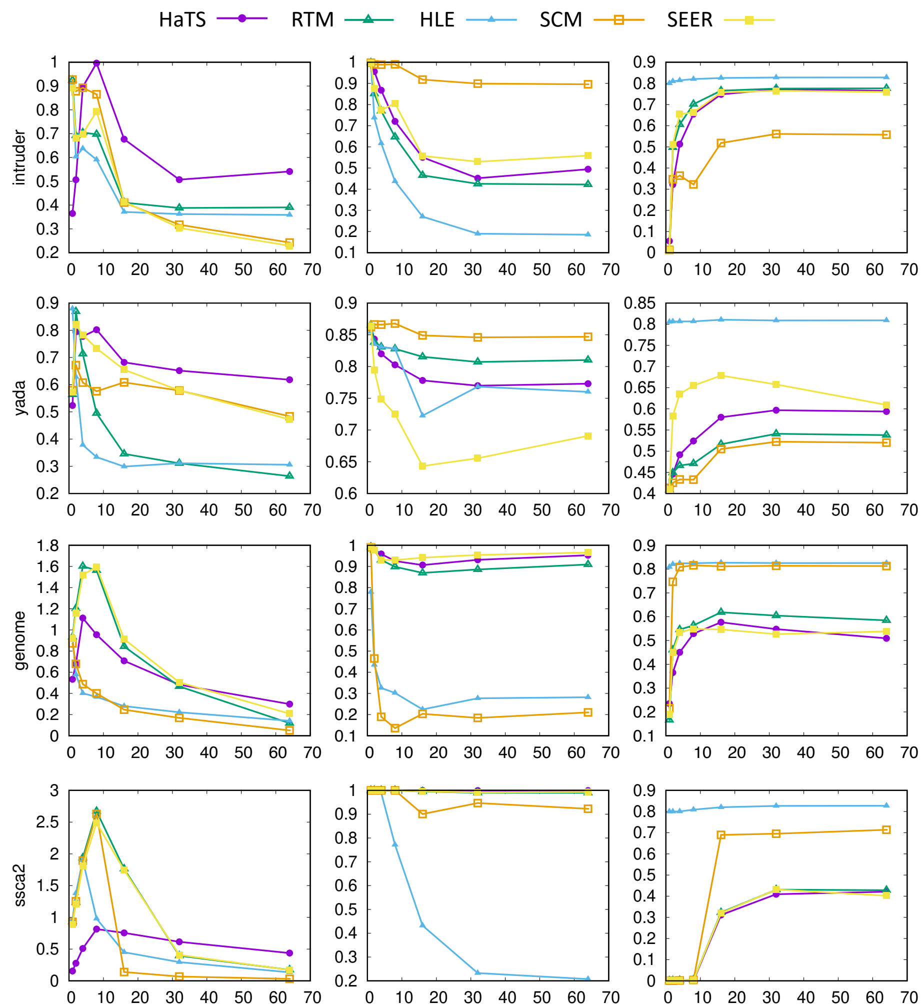

Figure 4 Performance results on the remaining applications of STAMP benchmark. X-axis shows number of application threads.

algorithms had a significant improvement over the others. However, HaTS maintains its performance better than others when the number of threads (and thus contention level) increases. We excluded `Bayes` and `Labyrinth` because it is known they provide unstable, and thus unreliable, results [13].

6 Conclusion

In this paper we presented HaTS, a Hardware-assisted Transaction Scheduler. HaTS groups incoming transactions into scheduling queues depending upon the specified conflict indicators. HaTS exploits the HTM conflict resolution to cope with the possibility of having erroneous conflict indicators or when conflict indicators are complex to identify. Results using the STAMP benchmark show that HaTS provides improvements in both high contention and low contention workloads.

References

1. Yehuda Afek, Amir Levy, and Adam Morrison. Software-improved hardware lock elision. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 212–221, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2611462.2611482, doi:10.1145/2611462.2611482.
2. Dan Alistarh, Syed Kamran Haider, Raphael Kübler, and Giorgi Nadiradze. The transactional conflict problem. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 383–392, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3210377.3210406, doi:10.1145/3210377.3210406.
3. Dan Alistarh, Justin Kopinsky, Petr Kuznetsov, Srivatsan Ravi, and Nir Shavit. Inherent limitations of hybrid transactional memory. *Distributed Computing*, 31(3):167–185, Jun 2018. URL: https://doi.org/10.1007/s00446-017-0305-3, doi:10.1007/s00446-017-0305-3.
4. Mohammad Ansari, Behram Khan, Mikel Luján, Christos Kotselidis, Chris C. Kirkham, and Ian Watson. Improving performance by reducing aborts in hardware transactional memory. In *High Performance Embedded Architectures and Compilers, 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings*, pages 35–49, 2010.
5. Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris C. Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *High Performance Embedded Architectures and Compilers, Fourth International Conference, HiPEAC 2009, Paphos, Cyprus, January 25-28, 2009. Proceedings*, pages 4–18, 2009.
6. Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, pages 81–91, 2007.
7. Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *ISCA*, pages 225–236, 2013.
8. Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, PACT '14, 2014.
9. Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, September 2008. URL: http://doi.acm.org/10.1145/1454456.1454466, doi:10.1145/1454456.1454466.
10. Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM. URL: http://doi.acm.org/10.1145/1508244.1508263, doi:10.1145/1508244.1508263.
11. Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2006. URL: http://dx.doi.org/10.1007/11864219_14, doi:10.1007/11864219_14.
12. Nuno Diegues and Paolo Romano. Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing*, ICAC '14. USENIX Association, 2014.
13. Nuno Diegues, Paolo Romano, and Stoyan Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 224–233, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2755573.2755578, doi:10.1145/2755573.2755578.

14 Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 125–134, 2008.

15 Aleksandar Dragojevic, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, pages 7–16, 2009.

16 Ricardo Filipe, Shady Issa, Paolo Romano, and João Barreto. Stretching the capacity of hardware transactional memory in ibm power architectures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 107–119, New York, NY, USA, 2019. ACM. URL: `http://doi.acm.org/10.1145/3293883.3295714`, `doi:10.1145/3293883.3295714`.

17 Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Transactional interference-less balanced tree. In *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 325–340, 2015.

18 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. URL: `http://doi.acm.org/10.1145/165123.165164`, `doi:10.1145/165123.165164`.

19 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

20 Junwhan Kim, Roberto Palmieri, and Binoy Ravindran. Enhancing concurrency in distributed transactional memory through commutativity. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, pages 150–161, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

21 Alexander Matveev and Nir Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 11–22, New York, NY, USA, 2013. ACM. URL: `http://doi.acm.org/10.1145/2486159.2486188`, `doi:10.1145/2486159.2486188`.

22 Alexander Matveev and Nir Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71, New York, NY, USA, 2015. ACM. URL: `http://doi.acm.org/10.1145/2694344.2694393`, `doi:10.1145/2694344.2694393`.

23 Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46, Sept 2008. `doi:10.1109/IISWC.2008.4636089`.

24 M. Mohamedin, R. Palmieri, A. Hassan, and B. Ravindran. Managing Resource Limitation of Best-Effort HTM. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2299–2313, Aug 2017. `doi:10.1109/TPDS.2017.2668415`.

25 Mohamed Mohamedin, Roberto Palmieri, and Binoy Ravindran. Brief announcement: On scheduling best-effort htm transactions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 74–76, New York, NY, USA, 2015. ACM. URL: `http://doi.acm.org/10.1145/2755573.2755612`, `doi:10.1145/2755573.2755612`.

26 Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, USA, December 1-5, 2001*, pages 294–305, 2001.

27 James Reinders. Transactional synchronization in haswell. *Intel Software Network. URL: http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/*, 2012.

28 Hugo Rito and João P. Cachopo. Props: A progressively pessimistic scheduler for software transactional memory. In *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, pages 150–161, 2014.

29 Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 87–102, 2007.

30 Lingxiang Xiang and Michael L. Scott. Conflict reduction in hardware transactions using advisory locks. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 234–243, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2755573.2755577, doi:10.1145/2755573.2755577.

31 Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, pages 169–178, 2008.