Self-Tuning Intel Transactional Synchronization Extensions

Nuno Diegues (ndiegues@gsd.inesc-id.pt) Paolo Romano (romano@inesc-id.pt)
INESC-ID / Instituto Superior Técnico, University of Lisbon

Abstract

Transactional Memory was recently integrated in Intel processors under the name TSX. We show that its performance can be significantly affected by the configuration of its interplay with the software-based fallback: in fact, there does not seem to exist a single configuration that can perform best independently of the application and workload. We address this challenge by introducing an innovative self-tuning approach that exploits lightweight reinforcement learning techniques to identify the optimal TSX configuration in a workload-oblivious manner, i.e. not requiring any off-line/*a-priori* sampling of the application's workload. To achieve total transparency for the programmer, we integrated the proposed algorithm in the GCC compiler. Our evaluation shows improvements up to 2× over state of the art approaches, while remaining within 5% from the performance achievable using optimal static configurations.

1 Introduction

Multi-core and many-core processors are nowadays ubiquitous. The consequence of this architectural evolution is that programmers need to deal with the complexity of parallel programming to fully unveil the performance potential of modern processors.

Transactional Memory (TM) [15] is a promising paradigm for parallel programming, that responds precisely to the need of reducing the complexity of building efficient parallel applications. TM brings to parallel programming the powerful abstraction of atomic transactions, thanks to which programmers need only to identify the code blocks that should run atomically, and not how atomicity should be achieved [22]. This is in contrast with the conventional lock-based synchronization schemes, where the programmer has to specify how concurrent accesses to shared state are synchronized to guarantee isolation. The TM runtime implements this with optimistic transactions, whose correctness is checked to ensure an illusion of serial executions (possibly aborting and rolling back the transaction), even though transactions run concurrently.

Recently, the maturing of TM research has reached an important milestone with the release of the first mainstream commercial processors providing hardware support for TM. In particular, Intel has augmented their instruction set for x86 with Transactional Synchronization Extensions (TSX), which represents the first generation of mainstream and commodity Hardware Transactional Memory (HTM): TSX is available in the 4^{th} generation Core processor, which is widely adopted and deployed ranging from tablets to server machines.

One important characteristic of this hardware support is its best-effort nature: due to inherent architectural limitations, TSX gives no guarantees as to whether transactions will ever commit in hardware, even in absence of conflicts. As such, programmers must provide a software fallback path when issuing a begin instruction, in which they must decide what to do upon the abort of a hardware transaction. One solution is to simply attempt several times before giving up to software. However, under which circumstances should one insist on using TSX before relying on software to synchronize transactions? We show that the answer to this question is clear: there is no one-size fits all solution that yields the best performance across all possible workloads. This means that the programmer is left with the responsibility of finding out the best choices for his application, which is not only a cumbersome task, but may also not be possible to achieve optimally with a static configuration. In fact, also Intel has recently acknowledged the importance of developing adaptive techniques to simplify the tuning of TSX [18].

1.1 Contributions and Outline

We study, to the best of our knowledge for the first time in literature, the problem of automatically tuning the policies used to regulate the activation of the fallback path of TSX. We first highlight the relevance of this self-tuning problem, and then present a solution that combines a set of lightweight reinforcement learning techniques designed to operate in a workload-oblivious manner. This means that we do not require any a priori knowledge of the application, and execute only runtime adaptation based on the online monitoring of applications' performance. We integrated the proposed self-tuning mechanisms within *libitm*, the TM library of the well-know GCC compiler. This allows achieving transparency to the programmers: a property of paramount importance that allows preserving the most compelling motivation of TM, namely its ease of use [14].

To assess our contributions we used the C++ TM specification [1] and relied on a comprehensive set of bench-

marks (most of which had to be ported to use this standard interface). More in detail, our evaluation study includes the well-known STAMP benchmark suite [21], which comprehends several realistic transactional applications with varying contention levels, size of transactions and frequency of atomic blocks. Besides STAMP, we also used a recent TM-based version of the popular Memcached [27], an in-memory web cache used to help scale web page servicing, which is widely used at Facebook. Finally we consider an example of a concurrent data-structure synchronized with TM, namely, a red-black tree. This is representative of important building blocks that are very hard to parallelize efficiently with locks, while generating typically short transactions (unlike some STAMP benchmarks).

In this large set of benchmarks our contributions allowed an average improvement of 2× over existing approaches, including both static heuristics and a state of the art adaptive solution [29] (although devised for software implementations of TM, and not HTM). We organized the rest of the paper as follows. Section 2 provides background on HTM and Intel TSX, whereas Section 3 motivates the relevance of devising self-tuning mechanisms for TSX. Then, in Sections 4 and 5 we present our learning techniques to self-tune TSX, as well as their integration in GCC in Section 6. Finally, we evaluate our proposals in Section 7, describe the related work in Section 8 and conclude in Section 9.

2 Background on Intel TSX

Intel TSX was released as part of the 4^{th} generation of Core processors (Haswell family). It has two main interfaces, called Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). To support HLE, two new prefixes (XACQUIRE and XRELEASE) were created, which can be placed in LOCK instructions. In older processors these prefixes are ignored, and the instructions are executed normally, meaning that locks are acquired and released normally. However, in Haswell processors these prefixes allow to elide the lock, such that it is only read but not written, effectively allowing concurrent threads to execute the same critical section at the same time. To ensure correctness, namely prevent data races in such optimistic executions, the hardware keeps track of the footprint accessed speculatively and rolls-back the execution if such footprint is invalidated by cache coherency. In such event, the thread re-executes the critical section but this time acquires and releases the lock normally. Such acquisition aborts concurrent elisions of the same lock, because these hardware speculations had read the lock and as such the lock state became part of their transactional footprint.

RTM leverages on the same hardware but accomplishes better flexibility because it exposes new instructions, namely, XBEGIN and XEND. This interface maps directly to the usual constructions of transactional programming of beginning and committing a transaction. Additionally, the XBEGIN instruction requires the programmer to provide a software handler to deal with transaction aborts. This has the advantage of allowing other strategies rather than giving up immediately on hardware transactions, which is the strategy followed by HLE.

The reason for requiring such software fallback is the best-effort nature of Intel TSX. Due to the inherently limited nature of HTMs [7, 16], TSX cannot guarantee that a hardware transaction will ever succeed, even if run in absence of concurrency. Briefly, TSX uses the L1 cache (private to each physical core) to buffer transactional writes, and on the cache coherence protocol to detect data conflicts. A plausible reason for a transaction never to succeed is because its data footprint does not fit in the L1 cache. Hardware transactions are also subject to abort due to multiple reasons that are not justified by concurrency alone, such as page faults and system calls.

In Alg. 1 we illustrate how GCC compiles transactional applications to use TSX (in the latest stable version 4.8.2 of GCC). In this approach (that we refer as GCC) transactions are attempted in hardware at most twice; if a hardware transaction aborts, the flow of execution reverts back to line 2 with an error code (the transaction can be aborted at any point between lines 3-15). This means that if TSX was always successful, then lines 4-8 would not be executed. When all attempts are exhausted, the execution falls through the fallback and acquires a global lock to execute normally, i.e., without hardware speculation.

To ensure correctness, a hardware transaction reads the lock (line 9) and aborts, either immediately if it finds it locked, or if some concurrent pessimistic thread acquires it before the hardware transaction commits. This mechanism safeguards the correctness of pessimistic executions that run without any instrumentation [4].

Note that HLE can be seen as a degenerate case of Alg. 1 in which the variable *attempts* is initialized with the value 1. In Fig. 1 we use a concurrent red-black tree

Algorithm 1 TSX in GCC

```
1:  int attempts ← 2
2:  int status ← XBEGIN
3:  if status ≠ ok then
4:      if attempts = 0 then
5:          acquire(globalLock)
6:      else
7:          attempts−−
8:          goto line 2
9:  if is_locked(globalLock)
10:     XABORT
11: ▷ ...transactional code
12: if attempts = 0 then
13:     release(globalLock)
14: else
15:     XEND
```

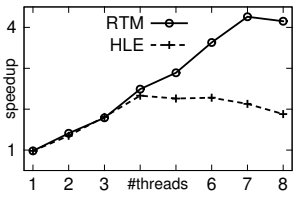


Fig. 1: Red-black tree synchronized with TSX.

Abort code
`retry`: Transient failure
`conflict`: Contention to data
`capacity`: Exceeded cache
`explicit`: XABORT invoked
`other`: Faults, preemption, ...

Fig. 2: Error codes in TSX.

synchronized with both interfaces of Intel TSX and show the speedup relatively to a sequential execution without synchronization. All the experimental results shown in this paper were obtained using an Intel Haswell Xeon E3-1275 processor with 32GB RAM, which has 8 virtual cores (4 physical, each with hyper-threading). Note that, for the time being, this is the maximum number of cores available in a machine with Intel TSX. We ran our experiments in a dedicated machine running Ubuntu 12.04, by using a minimum of 10 trials to obtain statistically meaningful results.

We focus our attention on the significant discrepancy in the performances in Fig. 1. This difference stems from the importance of insisting on the hardware support even when it fails (transiently). Since HLE gives up after one failed attempt, this creates a chain of failed speculations that acquire the lock and prevent concurrent speculations from proceeding via hardware — naturally, this occurs more often with higher concurrency, as the plot shows.

These considerations motivate the use of RTM over HLE (unless there are backwards compatibility concerns). However, as we shall discuss next, the use of RTM raises concerns of different nature, in particular related to the difficulty of properly tuning it.

3 Static Tuning of TSX

So far, we have motivated the usage of RTM, but only presented the simple approach that is implemented currently in GCC, which, as we shall see, is far from optimal. Indeed, in the light of recent findings [8, 17, 31], and based on the experience that we gathered after experimenting extensively with TSX, more effective mechanisms can be used to regulate the activation of the fallback path of TSX. We describe such mechanisms (which we call HEURISTIC) in Alg. 2.

The first point that we raise is that GCC (in Alg. 1) ignores the error codes returned by TSX's begin operation. The error codes available are briefly described in Fig. 2. Taking this into account, we consider that RETRY and CONFLICT codes represent ephemeral aborts, and as such we do not consume the attempts' budget in those cases (line 9). Furthermore, we consider CAPACITY errors to be permanent, and drop all attempts left (line 11). The objective is to avoid trying fruitlessly to use the hardware when it is not able to complete successfully, and short-cut right away to the fallback path.

Secondly, we set the attempts to 5 as that was reported by recent works as the best all-around figure [17, 31]. Choosing a given number is always going to be sub-optimal in some scenario, as it depends tremendously on the workload. Thirdly, we perform a lazy check for the global lock, which safely allows some concurrency with a pessimistic thread and hardware transactions [4].

Finally, we note that GCC suffers from the so-called *lemming effect* [8], in which one thread proceeding to the fallback (by acquiring the global-lock) causes all other concurrent transactions to do so too. This chain reaction can exhaust the attempts, and make it difficult for threads to resume execution of transactions in hardware. One way to avoid it, is by checking the lock before starting the transaction, and waiting in case it is locked (line 2).

Algorithm 2 HEURISTIC based approach for TSX.

```
1:  int attempts ← 5
    ▷ avoid the lemming effect
2:  while(is_locked(global-lock)) do pause   ▷ x86 instruction
3:  int status ← XBEGIN
4:  if status ≠ ok then
5:      if attempts = 0 then
6:          acquire(global-lock)
7:      else
8:          if status = explicit ∨ status = other then
9:              attempts ← attempts - 1      ▷ skipped if transient
10:         else if status = capacity then
11:             attempts ← 0  ▷ give up, likely that it always fails
12:         goto line 2
13: ▷ ...code to run in transaction
14: if attempts = 0 then
15:     release(global-lock)
16: else
17:     if is_locked(global-lock) then
18:         XABORT    ▷ check for concurrent pessimistic thread
19:     XEND
```

An alternative way to deal with the *lemming effect* is to use an auxiliary lock [2]. Briefly, the idea is to guard the global lock acquisition by another auxiliary lock. Aborted hardware transactions have to acquire this auxiliary lock before restarting speculation, which effectively serializes them. However, this auxiliary lock is not added to the read-set of hardware transactions, which avoids aborting concurrent (and successful) hardware transactions. If this procedure is attempted some times before actually giving up and acquiring the global lock, then the chain reaction effect can be avoided, as the auxiliary lock serves as a fallback manager preventing hardware transactions from continuously acquiring the fallback lock and preventing hardware speculations.

3.1 No One-size Fits All

The previous algorithms have a number of tuning knobs that needs to be properly configured. Summarizing: 1) How many times should a transaction be retried in hardware? 2) Should we trust the error codes to guide the decision to give up? and 3) What is the best way to manage the contention on the fallback path?

In order to assess the performance impact that these configuration options can have in practice, we conducted an experimental study in which we considered a configuration's space containing all possible combinations of feasible values of the following three parameters:

• Contention management — **wait** uses the simple wait

and pause approach of Alg. 2; **aux** uses the auxiliary lock [2]; **none** allows transactions to retry freely.

• Capacity aborts — **giveup** exhausts all attempts upon a capacity abort; **half** drops half of the attempts on a capacity abort; **stubborn** decreases one attempt only.

• Budget of attempts — varying from 1 to 16.

We tested all these combinations in our suite of benchmarks, with varying concurrency degrees, reaching the conclusion that there is no *one-size fits all* solution. In Table 1 we show some of the results from these experiments. Naturally, it is impossible to show all the 144 combinations for each benchmark and workload of our suite. We focus only on experiments with 4 threads, one workload per benchmark, and show only the best variant together with GCC (Alg. 1) and HEURISTIC (Alg. 2).

In general, the HEURISTIC algorithm yields some considerable improvements over GCC (notice Genome and Yada, with over 50% reduction in time), although in the other benchmarks it performs either similarly or slightly worse (notice Vacation, with 30% increase in time). However, the most important results are on the rightmost column, where we can see that the best performing variant varies a lot in its characteristics. Furthermore, the best result obtained is also typically better than those obtained by the algorithms seen so far: the geometric mean loss (i.e., additional time) of using GCC or HEURISTIC is of 30% and 21% (respectively) when compared to the best variant that we experimented with. These losses can extend up to 4× and 2× (e.g., Yada). To complement those results, we also show the performance of different TSX configurations in Genome when varying the number of threads, in Fig. 3: we can see that the best performance is obtained with widely different settings, and that even GCC and HEURISTIC can perform better than each other at different concurrency degrees.

The bottom line here is that static configurations of TSX can deliver suboptimal performance as a consequence of the high heterogeneity of the workloads generated by TM applications. The numbers above illustrate how much we could win in this set of benchmarks, workloads and concurrency degree, if we had a dynamic approach that adapts to the workload. Naturally, it is undesirable to require the programmer to come up with an optimal configuration for each workload, in particular because they may be unpredictable or even vary over time.

Another interesting result highlighted by this study is that the parameters' search space can be reduced by one dimension, i.e. contention management, as the **wait** or **aux** had in the vast majority of the cases similar speedups over **none**; whereas, whenever **none** reported to be the best, it was only by a very small margin. This finding suggests therefore to focus on the tuning of two main tuning knobs: i) the policy used to cope with capacity aborts and ii) the settings of the maximum number of attempts for a hardware transaction.

Table 1: Completion time (seconds, less is better) when using different RTM variants (described in Section 3.1).

Benchmark	GCC	HEURISTIC	Best Variant	
genome	3.46	1.69	1.59	wait-giveup-4
intruder	7.06	7.92	4.79	wait-giveup-4
kmeans-h	0.41	0.42	0.37	none-stubborn-10
rbt-l-w	6.25	6.40	5.27	aux-stubborn-3
ssca2	5.92	5.97	5.72	wait-giveup-6
vacation-h	6.81	8.99	5.83	aux-half-5
yada	28.5	11.6	6.96	wait-stubborn-15

4 Self-Tuning TSX

The proposed self-tuning solution for TSX adopts an online, feedback based design, which performs lightweight profiling of the applications at runtime. This allows to better fit the workloads of typical irregular applications that benefit most from synchronization facilities such as TSX [14], for which fully offline solutions are likely to fall prey of the over-approximations of solutions based on static analysis techniques. Another appealing characteristic of the proposed approach is that it does not necessitate any preliminary training phase, unlike other self-tuning mechanisms for Software TM (STM) based on off-line machine-learning techniques [10, 24].

Clearly, keeping the overhead of our techniques very low is a crucial requirement, as otherwise any gain is easily shadowed, for instance due to profiling inefficiencies or constant decision-making. Another challenge is the constant trade-off between exploring alternative configurations versus exploiting the current one, with the risk of getting stuck in a possibly sub-optimal configuration.

The proposed technique pursues overhead minimization in a twofold way. It employs efficient and concurrency friendly profiling techniques, which infer system's performance by sampling the x86's TSC cycle counter (avoiding any system call) and relying solely on thread-local variables to avoid inter-thread synchronization/interference. Besides that, it employs a combination of lightweight techniques, borrowed from the literature on reinforcement learning and gradient descent algorithms, which were selected, over more complex techniques, precisely because of their high efficiency.

Another noteworthy feature of the proposed self-tuning mechanism is that it allows for individually tuning the configuration parameters of *each* application's atomic block, rather than using a single global configuration. This feature is particularly relevant in programs that include transactions with heterogeneous characteristics (e.g., large vs small working sets, are contention-prone or not, etc.), which could benefit from using radically different configurations.

Before detailing the proposed solution, we first overview a state of the art solution [3] for a classical reinforcement learning problem, the multi-armed ban-

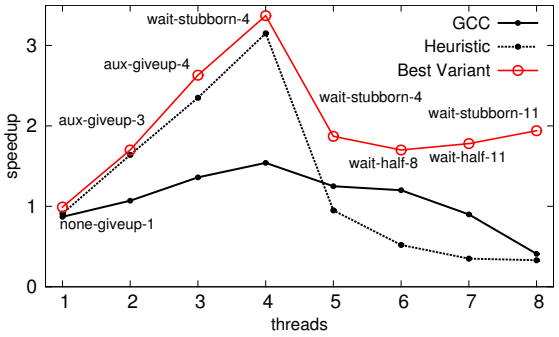


Figure 3: Speedup in Genome (higher is better).

dit [28]. This reinforcement learning technique is the key building block of the mechanism that we use to adapt the capacity abort management policy, which will be described in Section 4.2. We then explain the adaptation of how stubborn should one be in using TSX, i.e. the budget of attempts, in Section 4.3. The combination of those techniques is presented in Section 5.

4.1 Bandit Problem and UCB

The "bandit" (a.k.a. "multi-armed bandit") is a classic reinforcement learning problem that states that a gambling agent is faced with a bandit (a slot machine) with k arms, each associated with an unknown reward distribution. The gambler iteratively plays one arm per round and observes the associated reward, adapting its strategy to maximize the average reward. Formally, each arm i ($0 \leq i \leq k$) is associated with a sequence of random variables $X_{i,n}$ representing the reward of the arm i, where n is the number of times the lever has been used. The goal is to learn which arm i maximizes the average reward : $\mu_i = \sum_{n=1}^{\infty} \frac{1}{n} X_{i,n}$. To this purpose, the learning algorithm needs to try different arms to estimate their average reward. On the other hand, each suboptimal choice of an arm i costs, on average, $\mu^* - \mu_i$, where μ^* is the average obtained by the optimal lever. Several algorithms have been studied to minimize the regret (defined as $\mu^* n - \mu_i \sum_{i=1}^{K} E[T_i(n)]$, where $T_i(n)$ is the number of times arm i has been chosen).

Building on the idea of confidence bounds, the technique of Upper Confidence Bounds (UCB) creates an overestimation of the reward of each possible decision, and lowers it as more samples are drawn. Implementing the principle of optimism in the face of uncertainty, the algorithm picks the option with the highest current bound. Interestingly, this allows UCB to achieve a logarithmic bound on the regret value not only asymptotically, but also for any finite sequence of trials. More in detail, UCB assumes that rewards are distributed in the [0,1] interval, and associates each arm with a value:

$$\bar{\mu}_i = \bar{x}_i + \sqrt{2\frac{\log n}{n_i}} \quad (1)$$

where $\bar{\mu}_i$ is the current estimated reward for lever i; n is the number of the current trial; $\bar{x}_i$ is the reward for lever i; and n_i is the number of times the lever i has been tried. The right-hand part of the sum is an upper confidence bound that decreases as more information is acquired. By choosing, at any time, the option with maximum $\bar{\mu}_i$, the algorithm searches for the option with the highest reward, while minimizing the regret along the way.

4.2 Using UCB learning

We applied UCB to TSX by considering that each atomic block of the application has a slot machine (in the nomenclature of the previous Section 4.1), i.e., a corresponding UCB instance. With it, we seek to optimize the consumption of the attempts upon capacity aborts. In some sense, this models a belief on whether the capacity aborts witnessed are transient or deterministic, which cannot be assessed correctly based only on the error codes returned by aborted transactions. How many capacity aborts should we have to consider that an atomic block is failing deterministically? It is not obvious one can explicitly model such belief in that way; hence why UCB becomes appealing in this context.

We consider the three options identified in Section 3.1 with respect to capacity aborts. This creates three levers ($0 \leq i < 3$) in each UCB instance. We then use Eq. (1), for which we recall n is the number of the current trial (i.e., number of decisions so far for the atomic block), and n_i is the number of times the UCB instance chose lever i. We now specify the reward function for the levers (represented by $\bar{x}_i$). For this we used the number of processor cycles that it takes to execute the atomic block for each configuration. Hence we keep a counter c_i for each lever with the cycles that it consumed so far, and compute the reward $\bar{x}_i$ for lever i with:

$$\bar{x}_i = \frac{1}{c_i/n_i} \quad (2)$$

which means that we normalize the cycles of lever i, giving us a reward in the interval [0,1].

4.3 Using Gradient Descent Exploration

In order to optimize the number of attempts configured for each atomic block, we use an exploration technique, similar to hill climbing/gradient descent search [25]. The alternative of using UCB was dismissed because the parameter has a large space of search that does not map well to the lever exploration. This optimization problem is illustrated by the experiments in Fig. 4, where we show the

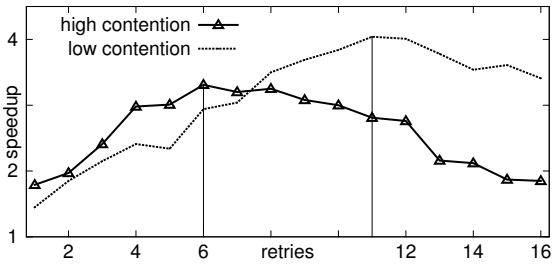


Figure 4: Speedup in Kmeans given attempts (8 threads).

performance improvement at 8 threads in Kmeans when varying the number of attempts for the configuration that yielded the best results. In the plot we show both low and high contention workloads of Kmeans, for which there are significantly different number of attempts yielding maximum values of improvement (namely, 6 and 11).

For this decision we also use the processor cycles that it takes to execute the atomic block, and at the end we execute gradient descent exploration. We augment it with probabilistic jumps to avoid getting trapped in a local maximum during the exploration. Furthermore, we memorize the best configuration seen so far to recover from unfortunate jumps.

For this, we store: the best configuration and performance seen so far (*best*); the last configuration and corresponding performance (*last*); and the current configuration (*current*). Note that here the configuration means simply the number of attempts. Then we use the following rules to guide exploration (strategy called GRAD):

1) with probability 1-p_{jump} play according to classic gradient descent; if performance improved along the current direction of exploration, keep exploring along that direction; otherwise reverse the direction of exploration.
2) With p_{jump} probability, select randomly the attempts with uniform probability for the next configuration. If after the jump performance decreased by more than *maxLoss*, then revert to the *best* known configuration.

Further, in order to enhance stability and avoid useless oscillations once identified the optimal solution, if, after a configuration change, performance did not change by more than *min*Δ, we block the gradient descent exploration and allow only probabilistic jumps (to minimize the risk of getting stuck in sub-optimal configurations).

Concerning the settings of the p_{jump}, *maxLoss*, and *min*Δ, we set them respectively to 1%, 40% and 5%, which are typical values for this type of algorithms [28] and whose appropriateness in the considered context will be assessed in Section 7.

5 Merging the Learning Techniques

So far we have presented: 1) UCB to optimize the consumption of attempts upon capacity aborts (Section 4.2); and 2) GRAD to optimize the allocation of the budget of attempts (Section 4.3). We now present their integration in our algorithm called TUNER.

The concern with the integration in TUNER is that the two optimization strategies overlap slightly in their responsibilities. The advantage is that this allows to simultaneously optimize the configuration accurately for atomic blocks that sometimes exceed capacity in a deterministic way, whereas, in other scenarios, can execute efficiently using TSX. This may be, for instance, depending on the current state of shared memory, or some input parameter used in the atomic block. It is possible to achieve this because UCB shall decide to short-cut the attempts when capacity aborts happen, whereas GRAD can keep the attempts' budget high to allow successful TSX execution when capacity aborts are rare.

One problematic scenario arises when an atomic block is not suitable for execution in hardware: either GRAD can reduce the attempts to 0, or UCB can choose the **giveup** mode. However, we may be unlucky and get an inter-play of the two optimizers such that they affect each other and prevent convergence of the decisions.

To solve this problem with their integration, we create a hierarchy among the two optimizers, in which UCB can force GRAD to explore in some direction and avoid ping-pong optimizations between the two. For this, we create a rule that is activated when the attempts' budget is exhausted: in such event we trigger a random jump to force GRAD to explore in the direction that is most suitable according to UCB, that is, explore more attempts if the UCB belief is **stubborn** and less attempts otherwise.

We compute the extension of the random jump for GRAD (based on the direction decided by UCB), by taking into account information about the types of aborts incurred so far. Namely, we collect the number of aborts due to capacity (*ab-cap*) and due to other reasons (*ab-other*). Then, if UCB suggests exploring more attempts (i.e., UCB belief is **stubborn**), we choose the length of the jump, noted J, proportionally to the relative frequency of *ab-other*:

$$J = \frac{ab\text{-}other}{ab\text{-}cap + ab\text{-}other} \cdot (maxTries - cur)$$

where *cur* is the current configuration of the budget of attempts and $maxtries = 16$. If UCB is different from **stubborn**, the jump has negative direction, and length:

$$J = \frac{ab\text{-}cap}{ab\text{-}cap + ab\text{-}other} \cdot cur$$

We now assess the efficiency of each of the optimization techniques alone, and their joint approach described above as TUNER. In this joint strategy we seek to understand if the two optimization techniques work well together: Fig. 5 shows the speedup of TUNER relatively to UCB and GRAD individually — we average the results across benchmarks since they yielded consistent results.

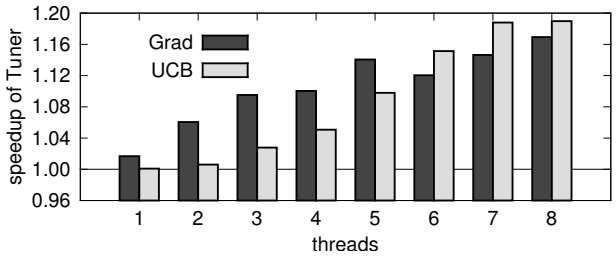


Figure 5: Geometric mean speedup of TUNER over UCB and GRAD across benchmarks and threads.

We can see from our experiments that the joint strategy provided average results that are always better than at least one of the approaches alone. More than that, for most cases TUNER improved over both individual strategies, which shows that employing them in synergy provides better results than the best approach alone. This is an encouraging result because tuning the attempts and dealing with capacity aborts is not entirely a disjoint concern. Overall, the results show that the joint approach yielded up to 20% improvement. Notice that each technique individually already improves over the baselines presented earlier, so any improvement when merged further reduces the gap with respect to the optimal result.

6 Integration in GCC

In this section we detail our implementation of TUNER, which we have integrated in the latest stable version of the Gnu C Compiler (GCC version 4.8.2), inside its *libitm* component. This component is responsible for implementing the C++ TM Specification [1] in GCC, which allows programmers to write atomic constructs in their programs that are compiled to calls to the TM runtime.

One important aspect of *libitm* is that it defines an interface that can be implemented by external libraries to plug in different TM runtimes and replace the implementations available inside GCC. Our initial expectation was that we could craft a TSX based runtime relying on TUNER as an external library. However, *libitm* does not completely delegate its work to such external library; it still keeps control on matters such as irrevocability (atomic blocks that cannot execute optimistically; e.g. those with I/O operations). This may cause performance loss because a single-lock TSX benefits from merging the irrevocability lock with the fallback lock. Furthermore, the choice of integrating TUNER into GCC allows achieving total transparency and maximum ease of use for the programmer.

We begin by laying out a high-level description of TUNER in Fig. 6. The flow starts every time a thread enters an atomic block. Since TUNER uses per atomic block statistics and configurations, we use the program counter as an identifier of the atomic block and retrieve the corresponding metadata kept by our algorithm. Every per block metadata is maintained in thread-local variables: hence threads perform self-tuning in an independent fashion. This has the advantage of avoiding synchronization and allowing threads to reach different configurations, which can be useful in case the various application threads are specialized to process different tasks (and generate different workloads).

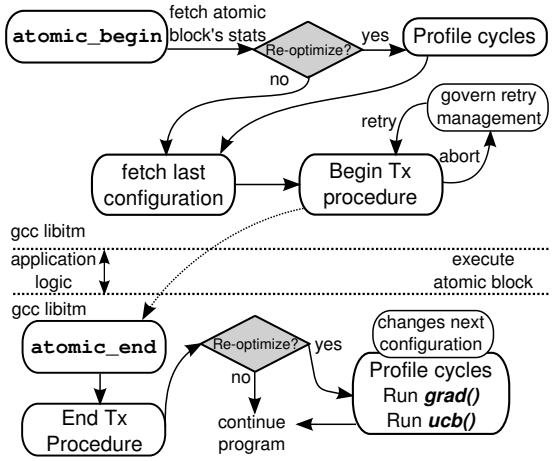


Figure 6: Workload-Oblivious tuning of TSX.

After fetching the metadata, we check whether it is time to re-optimize the configuration for that atomic block. This condition is a result of the sampling that we use to profile the application. For this, we keep a counter of executions in the metadata of the atomic block (recall that it is thread local) so that we only re-optimize periodically. This classic technique allows to keep the overheads low without missing noticeable accuracy in the decisions taken [19, 29, 30]. Hence we place the check for re-optimization in the begin and end of the atomic block. In the negative case, we simply execute the atomic block with the last configuration set up for it and proceed without any extra logic or profiling.

In the case that we re-optimize, this enables profiling of the cycles that it takes to execute the atomic block. For this, we use the `RDTSC` instruction in x86, which we use as a lightweight profiling tool to measure the relative cost of executing the block in different configurations. After this we attempt to start the transaction itself, which is better described in Alg. 3. Lines 8-16 describe the retry management policy. During a re-optimization period, if the attempts' budget is exhausted, this triggers the forced random jump over GRAD according to the description of TUNER in Section 5 (line 9), before proceeding to the fallback path. Note also that upon a capacity abort we adequately reduce the available budget according to the belief of UCB set in the current configuration (line 13).

After the application executed the atomic block, it calls back to *libitm*, and TUNER executes the usual procedure to finish the transaction. After this, it checks whether it is re-optimizing the atomic block, and in the positive case it runs GRAD and UCB to adapt the configuration for the next executions. To do so, it uses

Algorithm 3 TUNER adaptive configuration.

```
 1: int ucbBelief ← ▷ last configuration used
 2: int attempts ← ▷ last configuration used
 3: if reoptimize() then
 4:     long initCycles ← obtainRDTSC()
 5: while is_locked(global-lock) do pause
 6: int status ← XBEGIN
 7: if status ≠ ok then
 8:     if attempts = 0 then
 9:         if reoptimize() then tuneAttempts(ucbBelief)
10:         acquire(global-lock)
11:     else
12:         if status = capacity then
13:             ▷ set attempts according to ucbBelief
14:         else if status = explicit ∨ status = other then
15:             attempts ← attempts - 1
16:         goto line 5
17: ▷ ...code to run in transaction
18: if attempts = 0 then
19:     release(global-lock)
20: else
21:     if is_locked(global-lock) then XABORT
22:     XEND
23: if reoptimize() then
24:     long totalCycles ← obtainRDTSC() - initCycles
25:     ucbBelief ← UCB(totalCycles)    ▷ rules of Section 4.2
26:     attempts ← GRAD(totalCycles)    ▷ rules of Section 4.3
```

the processor cycles consumed, and applies the rules described throughout Section 4 to configure the budget and consumption of attempts in the metadata of the atomic block. Similarly to other metrics, assessing performance via processor cycles is also subject to thread preemption, which may inflate the actual cost of executing the atomic block. We mitigate this by binding threads to logical cores, and evaluating scenarios with up to as many threads as logical cores, as more than those typically deteriorates performance anyway.

7 Evaluation

We now present our final set of experiments, in which we compare TUNER with the following baselines:

- GCC - corresponding to Alg. 1, which is the implementation available in *libitm* in GCC 4.8.2.
- HEURISTIC - corresponding to Alg. 2 for which we tried to use static heuristics to better tune TSX.
- ADAPTIVELOCKS - proposed to decide between locks and TM for atomic blocks [29]; an analytical model is used and fed with statistics sampled at run-time (similarly to TUNER). We adapted their code (using CIL) to our environment integrated in GCC.
- TUNER - our contribution described in Alg. 3.
- Best Variant - an upper bound on the best result possible, obtained by picking the best settings of the considered parameters among all possible configurations for each benchmark and degree of parallelism. As such, this alternative does not correspond to a real tuning algorithm, but rather to an optimal, static configuration.

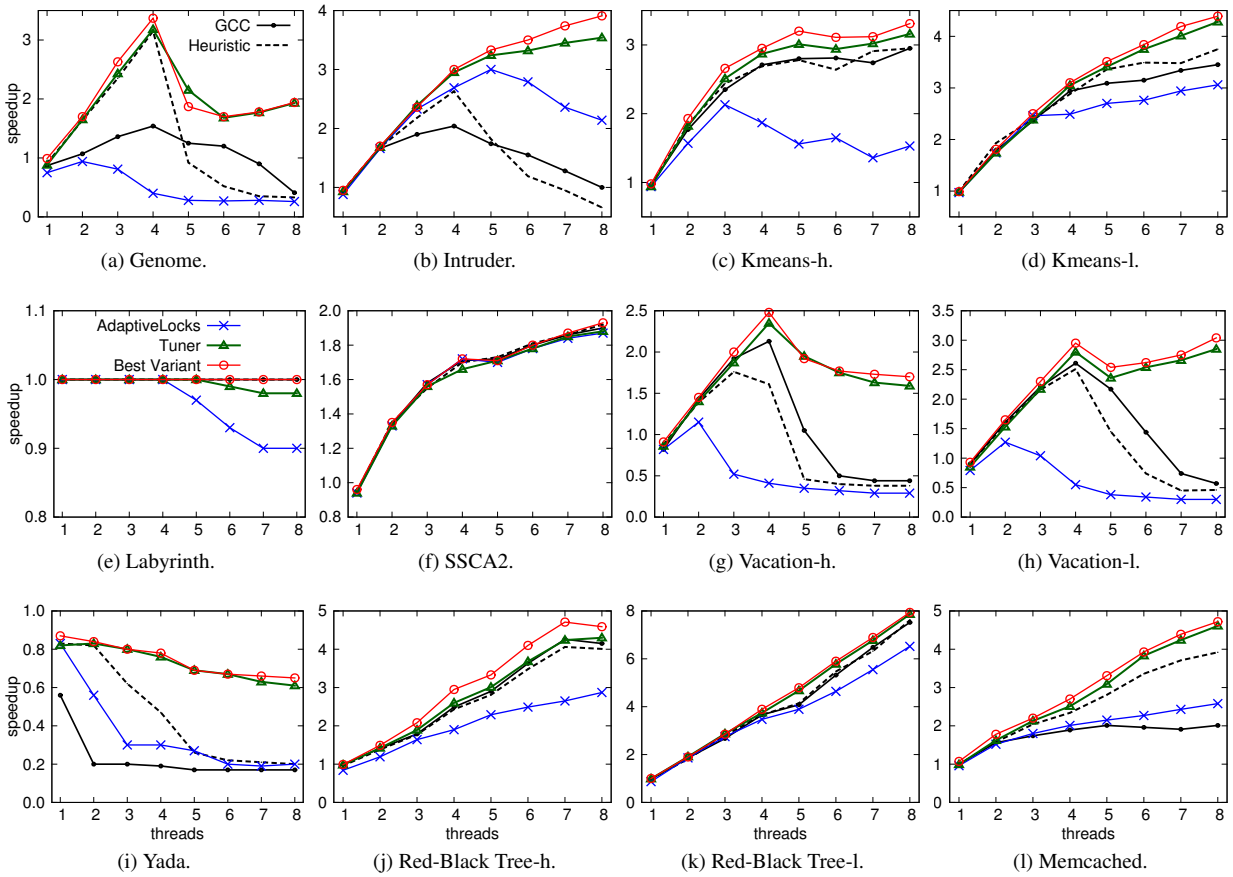
We used the standard parameters for the STAMP benchmarks and show workloads for low and high contention when available. For the red-black tree we used two workloads: low contention with 1 million items and 10% transactions inserting/removing items whereas the rest only performs fetch operations; and high contention with 1 thousand items and 90% transactions mutating the tree. For these benchmarks we present the speedup of each variant relatively to a sequential, non-synchronized execution. Finally we use a balanced workload for Memcached, configured with 50% gets and sets, and always set an equal number of worker and client threads. For this, we used the memslap tool in a similar fashion to [27]. In Memcached there is no sequential execution since there is always concurrency due to maintenance threads. As such, we use speedups relative to GCC at 1 thread, by having each execution last 60 seconds and measuring its throughput.

In general this set of experiments (Fig. 7) shows a typical gap in performance between the static configurations and the best possible variant. This gap is usually more noticeable as the concurrency degree increases — as we can see for instance in Kmeans-l — which is expected, since that is when the configuration parameters matter most to decide when it is profitable to insist on the hardware transactions of TSX. In short, these gaps in performance between the static alternatives and the best variant possible is exactly the room of improvement that we try to explore with TUNER in this paper.

In fact, TUNER is able to achieve performance improvements in all benchmarks with the exception of Labyrinth and SSCA2, in which it yields the same performance as the static approaches. In Labyrinth transactions are always too large to execute in hardware, and the benchmark executes about five hundred such large operations, which means the length of the transaction dominates the benchmark and no noticeable performance changes exist with regard to different configurations that do not insist too much on the hardware. In SSCA2 there is little time spent in atomic blocks, and some barriers synchronizing the phases of the workload, resulting in bottlenecks that are independent of the atomic blocks and that make all configurations perform similarly.

Table 2: Geometric mean speedup (across benchmarks) of each algorithm relatively to sequential executions.

Algorithms	threads			
	2	4	6	8
GCC	1.25	1.74	1.51	1.29
HEURISTIC	1.46	2.01	1.37	1.28
ADAPTIVELOCKS	1.26	1.19	1.10	1.11
TUNER	1.46	2.25	2.34	2.54
Best Variant	1.51	2.35	2.41	2.66

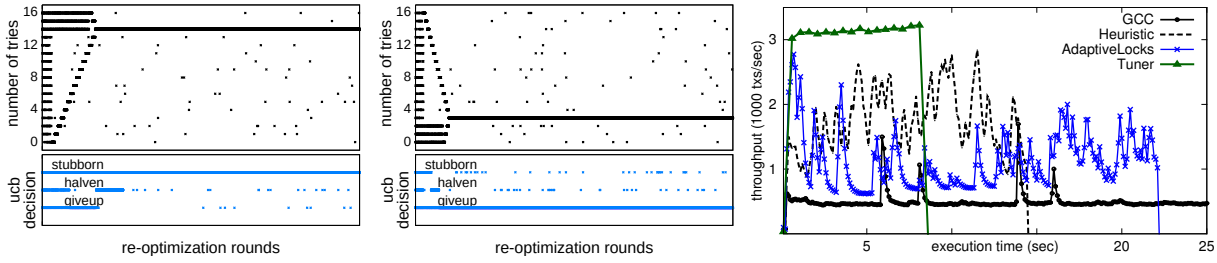


(a) Genome. (b) Intruder. (c) Kmeans-h. (d) Kmeans-l.

(e) Labyrinth. (f) SSCA2. (g) Vacation-h. (h) Vacation-l.

(i) Yada. (j) Red-Black Tree-h. (k) Red-Black Tree-l. (l) Memcached.

Figure 7: Speedup of different approaches to tune TSX relative to sequential executions in all benchmarks.



(a) Yada atomic block A. (b) Yada atomic block B. (c) Throughput in Yada with 8 threads.

Figure 8: Exploration and adaptation of TUNER on two different atomic blocks (left) and global throughput (right).

For all other benchmarks and workloads we find TUNER typically close to the best variant. Table 2 summarizes our findings across this extensive set of benchmarks: at 8 threads, the maximum hardware parallelism available for TSX, TUNER obtains an approximate improvement of 2× over GCC, HEURISTIC and ADAPTIVELOCKS, while remaining roughly 5% off the optimal solution identified by means of the exhaustive, offline exploration of the parameters' space.

We also note that the current hardware is limited in terms of hardware parallelism: in fact, some times going over 4 threads is not profitable as hyper-threading is not beneficial due to the extra pressure on L1 caches [13]. This, however, is an issue that has been tackled by related work (e.g. [9]) and whose importance shall be relatively diminished by the availability of new hardware to be released with more cores and without hyper-threading.

Finally, manual profiling and inspection revealed that TUNER consistently converged to configurations similar to the ones that performed best in our extensive offline testing. We present an example of the adaptation performed by TUNER in Fig. 8 in the Yada benchmark (we show the adaptation of one thread among 8 running concurrently). There, we can see the configuration of two atomic blocks being re-optimized, and converging to two drastically different configurations: the left

block executes efficiently with TSX whereas the right one does not. This illustrates two advantages of our solution: 1) the adaptation allows heterogeneous threads and atomic blocks to converge to different configurations; 2) an atomic block, such as that in Fig. 8b, can still insist moderately on using TSX as long as capacity aborts do not occur, but react quickly in case they appear. As a result, we can see the significant performance increase of our solution, depicted in Fig. 8c with the throughput of each solution as the benchmark executes. We highlight the steadiness of TUNER against the irregular and spiky performance of the static solutions that can, at best, only fit the workload for limited time periods.

8 Related Work

Transactional Memory was initially proposed as an extension to multi-cores' cache coherence protocols [15]. Due to the inaccessibility of rapidly prototyping in such environment, researchers resorted to software implementations (STM) to advance the state of the art [14]. These STMs require instrumenting the code (either manually or compiler-assisted) to invoke the TM software runtime in every read and write to shared memory. As a result, STMs impose some overhead in sequential executions, but they are efficient enough to pay off with some meaningful degree of parallelism [11, 12].

Recently, implementations in hardware (HTM) became available in commercial processors delivered by major industry players. Beyond Intel, IBM also provided support for HTM [16], in processors mostly used on high performance computing. We only had access to an Intel machine, but we believe the techniques described here should also be applicable to IBM's HTMs due to their similar nature. Furthermore, the mainstream nature of Intel processors increases significantly the relevance of works, like this, aimed to optimize its performance.

We are not aware of any work that self-tunes TSX (or any similar HTM). Works that studied TSX's performance [17, 31] obtained promising results, but relied on manual tuning and provided only brief textual insights as to how the decisions to configure it were taken.

Given the best-effort nature of this first generation of HTM in commodity processors, it is desirable to consider an efficient solution for the fallback path in software. One interesting idea is to use STMs combined with HTM (HybridTMs), so that transactions that are not successful in hardware can execute in software without preventing all concurrency, as is the case of pessimistic coarse locking-based schemes. In this scope, some work has obtained promising results with a simulator for HTM support from AMD [6,23]. However, there are no official plans to integrate AMD's proposal [5] in a commercial processor. More recently, the Reduced TM technique has been proposed for Intel TSX [20], but it was only evaluated in an emulated environment. In this paper we take a step back, and try to optimize as much as possible the HTM usage, before trying to integrate it with more complex fallback paths than that of a global lock. We believe that for most common situations this should be enough, as evidenced by the recent application of Intel TSX in SAP Hana database [17].

Additionally, there have been other proposals for adaptation in TMs in software. Adaptive Locks [29], VOTM [19], and Dynamic Pessimism [26] adapt between optimistic (with STM) and pessimistic (via locking) execution of atomic blocks. Unfortunately, these works do not map directly to best-effort HTMs, as we showed in our evaluation (by considering Adaptive Locks, as the authors kindly provided us with their code). More complex adaptation schemes have been proposed to self-tune the choice between different STM algorithms in AutoTM [30]. The main drawback of these kind of works, with regard to the HTM setting studied in this paper, is that these self-tuning proposals require knowledge that is not available from the HTM support that we have, such as the footprint of transactions (their read- and write-sets). That is, unless we instrument reads and writes to obtain it, which would defeat the purpose of HTM to lower the overhead of TM over STMs.

9 Conclusions

In this paper we studied the performance of the hardware support available via Intel TSX in the latest generation of x86 Core processors. This interface allows some flexibility in the definition of the software fallback mechanism triggered upon transactional aborts, with regard to when and how to give up executing hardware transactions. We showed that no single configuration of the software fallback can perform efficiently in every workload and application. Motivated by these findings, we presented TUNER, a novel self-tuning approach that combines reinforcement learning techniques and gradient-descent exploration-based algorithms to self-tune TSX in a workload-oblivious manner. This means that TUNER does not require a priori knowledge of the application, and executes fully online, based on the feedback on system's performance gathered by means of lightweight profiling techniques. We integrated TUNER in the well known GCC compiler, achieving total transparency for the programmer. We evaluated our solution against available alternatives using a comprehensive set of applications showing consistent gains.

Acknowledgements: We thank Konrad Lai, Ravi Rajwar and Richard Yoo from Intel for the insightful discussions about TSX during the conduct of this research.

This work was supported by national funds through FCT (Fundação para a Ciência e Tecnologia) under project PEst-OE/EEI/LA0021/2013 and by project GreenTM EXPL/EEI-ESS/0361/2013.

References

[1] ADL-TABATABAI, A., SHPEISMAN, T., AND GOTTSCHLICH, J. Draft Specification of Transactional Language Constructs for C++. In *Intel* (2012).

[2] AFEK, Y., LEVY, A., AND MORRISON, A. Programming with Hardware Lock Elision. In *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2013), pp. 295–296.

[3] AUER, P., CESA-BIANCHI, N., AND FISCHER, P. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn. 47*, 2-3 (May 2002), 235–256.

[4] CALCIU, I., SHPEISMAN, T., POKAM, G., AND HERLIHY, M. Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. In *9th Workshop on Transactional Computing (TRANSACT)* (2014).

[5] CHRISTIE ET AL., D. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *Proceedings of the 5th EuroSys* (2010), pp. 27–40.

[6] DALESSANDRO ET AL., L. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011), pp. 39–52.

[7] DICE, D., LEV, Y., MOIR, M., AND NUSSBAUM, D. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009), pp. 157–168.

[8] DICE ET AL., D. Applications of the Adaptive Transactional Memory Test Platform. In *3rd Workshop on Transactional Computing (TRANSACT)* (2008).

[9] DIDONA, D., FELBER, P., HARMANCI, D., ROMANO, P., AND SCHENKER, J. Identifying the Optimal Level of Parallelism in Transactional Memory Applications. In *Proceedings of the 1st Inernational Conference on Networked Systems (NETYS)* (2013), pp. 233–247.

[10] DIDONA, D., ROMANO, P., PELUSO, S., AND QUAGLIA, F. Transactional auto scaler: elastic scaling of in-memory transactional data grids. In *Proceedings of the International Conference on Autonomic Computing (ICAC)* (2012), pp. 125–134.

[11] DIEGUES, N., AND CACHOPO, J. Practical Parallel Nesting for Software Transactional Memory. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC)* (2013), pp. 149–163.

[12] DIEGUES, N., AND ROMANO, P. Time-warp: Lightweight Abort Minimization in Transactional Memory. In *Proceedings of the 19th Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2014), pp. 167–178.

[13] DIEGUES, N., ROMANO, P., AND RODRIGUES, L. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2014).

[14] DRAGOJEVIĆ, A., FELBER, P., GRAMOLI, V., AND GUERRAOUI, R. Why STM Can Be More Than a Research Toy. *Commun. ACM 54*, 4 (Apr. 2011), 70–77.

[15] HERLIHY, M., AND MOSS, J. E. B. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)* (1993), pp. 289–300.

[16] JACOBI, C., SLEGEL, T., AND GREINER, D. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 45th Symposium on Microarchitecture (MICRO)* (2012), pp. 25–36.

[17] KARNAGEL ET AL., T. Improving In-Memory Database Index Performance with Intel TSX. In *Proceedings of the 20th Symposium on High Performance Computer Architecture (HPCA)* (2014).

[18] KLEEN, A. Scaling Existing Lock-based Applications with Lock Elision. *Queue 12*, 1 (Jan. 2014), 20:20–20:27.

[19] LEUNG, K.-C., CHEN, Y., AND HUANG, Z. Restricted admission control in view-oriented transactional memory. *The Journal of Supercomputing 63*, 2 (2013), 348–366.

[20] MATVEEV, A., AND SHAVIT, N. Reduced Hardware Transactions: A New Approach to Hybrid Transactional Memory. In *Proceedings of the 25th Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2013), pp. 11–22.

[21] MINH, C. C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)* (2008), pp. 35–46.

[22] PANKRATIUS, V., AND ADL-TABATABAI, A. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the 23rd Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2011), pp. 43–52.

[23] RIEGEL, T., MARLIER, P., NOWACK, M., FELBER, P., AND FETZER, C. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the 23rd Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2011), pp. 53–64.

[24] RUGHETTI, D., DI SANZO, P., CICIANI, B., AND QUAGLIA, F. Machine Learning-Based Self-Adjusting Concurrency in Software Transactional Memory Systems. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2012), pp. 278–285.

[25] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2009.

[26] SONMEZ, N., HARRIS, T., CRISTAL, A., UNSAL, O., AND VALERO, M. Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In *Proceedings of the International Symposium on Parallel Distributed Processing (IPDPS)* (2009), pp. 1–10.

[27] SPEAR, M., VYAS, T., AND RUAN, WENJIA LIU, Y. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014), pp. 399–412.

[28] SUTTON, R. S., AND BARTO, A. G. *Introduction to Reinforcement Learning*, 1st ed. MIT Press, Cambridge, MA, USA, 1998.

[29] USUI, T., BEHRENDS, R., EVANS, J., AND SMARAGDAKIS, Y. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *Proceedings of the 18th Conference on Parallel Architectures and Compilation Techniques (PACT)* (2009), pp. 3–14.

[30] WANG, Q., KULKARNI, S., CAVAZOS, J., AND SPEAR, M. A Transactional Memory with Automatic Performance Tuning. *ACM Trans. Archit. Code Optim. 8*, 4 (Jan. 2012), 54:1–54:23.

[31] YOO, R. M., HUGHES, C. J., LAI, K., AND RAJWAR, R. Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2013), pp. 1–11.