Self-tuning Intel Restricted Transactional Memory

Nuno Diegues\textsuperscript{a,b,}\textsuperscript{*}, Paolo Romano\textsuperscript{a,b}

\textsuperscript{a} INESC-ID, Rua Alves Redol 9, Lisbon, Portugal
\textsuperscript{b} Instituto Superior Técnico, Universidade de Lisboa, Av. Rovisco Pais 1, Portugal

\textbf{ARTICLE INFO}

Article history:
Received 9 April 2015
Revised 2 October 2015
Accepted 3 October 2015
Available online 9 October 2015

Keywords:
Hardware Transactional Memory
Performance
Self-tuning

\textbf{ABSTRACT}

The Transactional Memory (TM) paradigm aims at simplifying the development of concurrent applications by means of the familiar abstraction of atomic transaction. After a decade of intense research, hardware implementations of TM have recently entered the domain of mainstream computing thanks to Intel’s decision to integrate TM support, codenamed RTM (Reduced Transactional Memory), in their last generation of processors.

In this work we shed light on a relevant issue with great impact on the performance of Intel’s RTM: the correct tuning of the logic that regulates how to cope with failed hardware transactions. We show that the optimal tuning of this policy is strongly workload dependent, and that the relative difference in performance among the various possible configurations can be remarkable (up to 10× slow-downs).

We address this issue by introducing a simple and effective approach that aims to identify the optimal RTM configuration at run-time via lightweight reinforcement learning techniques. The proposed technique requires no off-line sampling of the application, and can be applied to optimize both the cases in which a single global lock or a software TM implementation is used as fall-back synchronization mechanism.

We propose and evaluate different designs for the proposed self-tuning mechanisms, which we integrated with GCC in order to achieve full transparency for the programmers. Our experimental study, based on standard TM benchmarks, demonstrates average gains of 60% over any static approach while remaining within 5% from the performance of manually identified optimal configurations.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

The ubiquity of multi-core processors in mainstream architectures has motivated the need to identify programming paradigms capable of simplifying the development of concurrent applications. In this scope, Transactional Memory (TM)\textsuperscript{[1]} is a promising approach because it exposes a simple interface: it only requires programmers to identify which code blocks should run atomically, and not how atomicity should be achieved.

This contrasts with traditional lock-based synchronization schemes, where, in order to achieve good scalability, programmers need to design complex lock acquisition schemes. These are often prone to deadlocks/livelocks\textsuperscript{[2]}, are hard to reason about and debug\textsuperscript{[3–5]}, and, even worse, hinder software composability\textsuperscript{[6]}.

\textsuperscript{*} Corresponding author at: Instituto Superior Técnico, Universidade de Lisboa, Av. Rovisco Pais 1, Portugal. Tel.: +351968427763.
E-mail address: nmld@tecnico.ulisboa.pt (N. Diegues).

http://dx.doi.org/10.1016/j.parco.2015.10.001
0167-8191/© 2015 Elsevier B.V. All rights reserved.
Conversely, TM exposes a simple and familiar abstraction of atomic transactions, which, on one hand, shelters programmers from the complexity of locks [7,8], while allowing on the other hand for efficient implementations. These were shown to achieve performance similar, and sometimes even superior, to complex, ad hoc designed fine-grained lock synchronization schemes [9–12]. As such, TM promises to conflate ease of usage, efficiency and scalability.

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 the ×86 instruction set with Transactional Synchronization Extensions under the name Restricted Transactional Memory (RTM). This commodity Hardware Transactional Memory (HTM) implementation is available in the 4th generation core processor, which is widely adopted and deployed, ranging from tablets to server machines.

1.1. Problem

One important characteristic of the Intel HTM is its best-effort nature: due to inherent hardware/architectural limitations, RTM gives no guarantees as to whether a hardware transaction can commit successfully, even in absence of concurrency and data conflicts. This is easily understandable as the implementation heavily relies on the usage of processors’ caches, which have limited space. Indeed, although solutions providing stronger progress guarantees for HTM have been proposed in literature, the alterations required to existing CPU architectures are currently perceived as overly invasive and risky [13].1 For this reason, the best-effort nature of HTM is a sweet spot in the design choices, being shared by every HTM implementation currently proposed by industrial CPU manufacturers, including Intel [14], AMD [15], IBM [16,17] and Oracle [18], and it appears unlikely that alternative designs will be pursued in the near term future.

As such, a programmer using hardware transactions in RTM must decide what should be done upon the abort of that hardware transaction: under which circumstances should an aborted transaction be re-executed using RTM, or when should it resort to an alternative fall-back software-based synchronization scheme?

In this paper we show that there is no definite answer to this question: there is no one-size fits all solution that yields the best performance across all possible workloads. To better illustrate this important statement, we provide experimental evidence summarized in Fig. 1. This figure shows the performance of three example configurations, using Intel HTM, across some popular TM benchmarks. A detailed description of these configurations and benchmarks will be provided, respectively, in Section 3 and 8. These results show the relative performance of each configuration, with respect to the optimal one that we found for each benchmark. The outcome of this plot supports our claim: no configuration performs consistently better than all the others, and they all perform excellently in some benchmarks and poorly in others.

This important fact means that the programmer is left with the responsibility of finding out the best choices for his application—a problem that is cumbersome and time consuming to tackle via off-line profiling, given that there are many available configurations. Even worse, in fact, there may not exist a single optimal solution to be found statically, when in the presence of dynamic workloads. These facts have also been recently acknowledged by Intel researchers, highlighting the importance of developing adaptive techniques to simplify the tuning of RTM [19].

1 IBM System Z processors represent a notable exception: they guarantee transactions to commit if they abide certain constraints in terms of footprint and instructions, provided that they do not conflict. However, we note that those requirements are quite strict (at most 32 instructions, accessing up to 256 bytes of memory).
1.2. Contributions

In this paper we contribute to addressing the aforementioned challenge by studying the problem of automatically tuning the policies used to regulate the activation of the fall-back path of RTM. In more detail, our contributions are structured as follows:

- In Section 2 we begin by providing background on the HTM support available in the last generation of Intel processors. We also evaluate, to the best of our knowledge for the first time in literature, a set of techniques aimed at optimizing the two most common fall-back synchronization mechanisms: namely a single lock or a software TM (STM). This allows us to ensure an appropriate tuning of the experimental test-bed platform used in the remainder of the paper—an important precondition for the results presented later on in this work.
- Then, in Section 3, we show that self-tuning is essential to achieve robust performances across different workloads. In particular, we present evidence showing that no single configuration exists that outperforms all others. Furthermore, we find that the performance of any static configuration can deliver losses up to $10 \times$ when compared to the optimal solution for each individual workload.
- In Section 4 we present a novel solution that relies on lightweight reinforcement learning techniques to perform run-time adaptation based on the online monitoring of applications’ performance. For this to be possible, it is crucial to reduce any overhead of the profiling and to strike a good balance between exploring new configurations and exploiting available knowledge on already sampled configurations. In Section 5 we also discuss different designs for our solution, and their inherent trade-offs. We then explain, in Section 6, how our algorithms have been integrated in the GCC compiler in order to achieve full transparency for programmers. This is an important contribution as we preserve the simple abstraction of TM, and effectively conceal our self-tuning mechanisms to adapt the TM configuration to the workload characteristics.
- Finally, we instantiate our TM adaptive approach to the two aforementioned RTM fall-back paths: namely with a fall-back based on a single-global lock, as well as an STM-based fall-back. We study their performance, in Section 8, by using a large set of TM benchmarks, in which we obtain average gains of 60% over the Best Static alternatives and up to two-fold improvements in specific benchmarks. This is obtained also while decreasing the performance gap to the optimal configuration in each benchmark to within a 5% interval.

2. Background on TM

The TM programming model [1] exposes a simple abstraction to the programmer to synchronize concurrent applications. It relies on the familiar abstraction of transactions to implement atomic blocks: a programmer needs only to enclose the fragments of code that should be executed atomically, i.e., equivalently to a sequential run, within atomic blocks. The TM run-time executes atomic blocks with transactions, which are typically processed in a speculative fashion in order to maximize concurrency. Consistency is ensured by aborting transactions that perform unsafe operations, and progress is attained by automatically re-executing transactions until their successful completion.

There are three main types of TM implementations: hardware based (HTM), software based (STM), and combinations thereof (HybridTMs).

HTMs are typically implemented as an extension of the cache coherence protocol of the processor [17]. This is a natural synergy, because implementing transactions requires to keep track of reads and writes to shared memory, so to detect conflicts with concurrently executing transactions. In particular, the invalidation of cache lines associated with memory locations accessed by an active transaction $T$ is exploited to detect contention and trigger the immediate abort of $T$.

Given their hardware nature, HTM can effectively reduce the overhead for tracing conflicts between concurrent transactions. On the other hand, HTM also suffers of several restrictions that can cause spurious aborts of transactions. For instance, if a transaction performs an excessive number of read/write accesses and exhausts the capacity of the corresponding hardware buffers, it is deterministically aborted even if it runs in the absence of concurrency. Because of this best-effort nature of HTM, in order to ensure liveness it requires specifying a fall-back synchronization mechanism (also called fall-back path), which is typically implemented as a single-global lock.

Given their purely software-based nature, STMs are the most flexible and least constrained class of TM systems, and the literature on STMs has explored a broad range of alternative designs (e.g., [20–25]). Independently of the specific algorithm that they implement, STMs require software instrumentation of read and write accesses to shared memory. This allows the STM run-time (typically implemented as a library) to validate the read-set and apply the write-set of transactions, analogously to what the cache coherence protocol does for HTMs.

Naturally, this instrumentation adds considerable overhead to STM when compared to pure hardware-based executions. However, as we shall see, HTM is also faced with several restrictions, which gives room to explore synergies between STMs and HTMs: these combinations are called HybridTMs [26]. The key idea there is to use an STM on the fall-back path of an HTM, and to introduce additional mechanisms to make both TM implementations aware of each other and ensure their correct interplay.

In the following sections we provide additional background information regarding the TM implementations that we target in this work. First, in Section 2.1, we present the key characteristics of Intel’s HTM implementation, i.e., RTM. Then we focus on two possible fall-back synchronization mechanisms for RTM:

1. In Section 2.2, we look at optimizing the usage of a single-global lock with RTM—which we call RTM-SGL.
Table 1

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Xeon E3-1275 v3 3.5 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>4 (each with hyper-threading)</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32 kB 8-way (per core)</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256 kB 8-way (per core)</td>
</tr>
<tr>
<td>L3 cache</td>
<td>8 MB (shared)</td>
</tr>
<tr>
<td>Cache line</td>
<td>64 B</td>
</tr>
<tr>
<td>RAM size</td>
<td>32 GB</td>
</tr>
<tr>
<td>Operating system</td>
<td>Ubuntu 12.04</td>
</tr>
</tbody>
</table>

Fig. 2. Red-black tree synchronized with both Intel RTM and HLE. The ability to define properly tuned fall-back paths in RTM often makes it more competitive than HLE.

2. In Section 2.3, we explain how an STM can be integrated as the fall-back of RTM, yielding a HybridTM. Specifically, we use a lightweight STM, called NOrec [21], on the fall-back of RTM and discuss several techniques that allow to couple the two TM implementations in an efficient way.

2.1. A primer on Intel Restricted Transactional Memory

The Intel Transactional Synchronization Extensions (TSX), released as part of the Haswell processors (such as the one we use; more details in Table 1), comprises two possible interfaces to software programs: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM).

The former allows to elide locks and execute code speculatively in a backwards-compatible manner. As such, its interface is designed to prefix `lock` instructions, and does not allow for software control, i.e., the processor has full control on how many attempts should be performed in a speculative fashion, before falling back to a pessimistic approach and acquiring the lock. For this reason, in this work we focus on optimizing the performance of RTM, which exposes a conventional TM abstraction to the programmer and simply requires to demarcate code fragment within atomic blocks.

RTM leverages on the same hardware as HLE, but accomplishes better flexibility because it allows for controlling, in software, the retry logic of transactions. More in detail, RTM exposes two instructions for transaction demarcation, namely, `xbegin` and `xend`. This interface maps directly to the usual constructs of transactional programming to begin and commit an atomic block. A very important detail is that the `xbegin` instruction requires to specify a software handler to deal with transaction aborts. This has the advantage of allowing the development of more flexible and effective transaction retry logics than the one embedded in HLE. In Fig. 2 we show an experimental evidence of the above claim, with a typical configuration of RTM [14,27]. Due to the ability of tuning the fall-back in RTM, one can obtain considerable gains with respect to HLE, whose fixed strategy is overly pessimistic in giving up on hardware transactions in many cases.

As RTM provides feedback on the causes of transactions’ aborts, this enables the possibility of exploiting this information, in software, to determine the transaction retry logic. The transaction retry logic is typically handled by a software library, and in Algorithm 1, we illustrate the simple policy employed by the Gnu C Compiler (GCC) support for TM [29] in version 4.8.
Table 2
On the left: possible bits that may be set by RTM in register EAX to indicate reasons to abort a transaction. On the right: possible events that may be monitored during RTM execution via model-specific registers.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Reason (examples)</th>
<th>Event</th>
<th>Reason (examples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>retry</td>
<td>Transient failure</td>
<td>MISC1</td>
<td>Conflicts, capacity overflow</td>
</tr>
<tr>
<td>conflict</td>
<td>Contention to data</td>
<td>Capacity</td>
<td>Specifically write-set overflow</td>
</tr>
<tr>
<td>capacity</td>
<td>Exceeded cache size</td>
<td>MISC3</td>
<td>Forbidden instructions, page faults</td>
</tr>
<tr>
<td>explicit</td>
<td>xabort invoked</td>
<td>MISC4</td>
<td>Illegal memory accesses (I/O)</td>
</tr>
<tr>
<td>other</td>
<td>Faults, preemption</td>
<td>MISC5</td>
<td>Interrupts</td>
</tr>
</tbody>
</table>

Algorithm 1 How GCC compiles an application’s atomic block to use RTM.

```
1: int attempts ← 2  # budget of attempts for hardware transactions to abort
2: ▶ aborted transactions automatically jump to line 3 and return an error from the call to xbegin; otherwise it returns ok when the transaction was begun
3: int status ← xbegin  # result of begin is placed in EAX (omitted here)
4: if status ≠ ok then
5: while is_locked(global-lock) do pause
6: if (attempts = 0) ∨ (status & retry ≠ 0) then
7: acquire(globalLock)  # global lock software fall-back alternative to RTM
8: goto 14
9: else
10: attempts ←−  # execution failed, avoid infinite loop by decreasing budget
11: goto line 3  # jump back to initiate a new hardware transaction
12: if is_locked(globalLock)
13: xabort
14: ▶ application atomic block code runs here
15: if attempts = 0 then
16: release(globalLock)  # finalize software fall-back path
17: else
18: xend  # finalize hardware transaction fast path
```

The algorithm essentially shows the code that the compiler places around an atomic block. We call this approach simple because it essentially tries to use RTM twice, ignoring the abort reason. If it keeps failing (for any of the reasons listed above) the atomic block is synchronized with a single global lock in the software fall-back path. Whenever a hardware transaction aborts, the flow of execution reverts back to line 3 with an error code (the transaction can be aborted at any point between lines 4 and 18). This means that if RTM was always successful, then lines 6–11 would never be executed. After each abort, the thread waits for the global lock to be free before proceeding (in line 5): this is meant to avoid the lemming effect, which we describe in the following section. When all attempts are exhausted, the execution resorts to the fall-back and acquires a global lock to execute the atomic block normally, i.e., without hardware speculation (the lock is acquired in line 7 and released in line 16). Alternatively, the thread may also short cut through this path in case the status returned by the failed hardware transaction does not advise to retry (i.e., the retry bit described in Table 2 is not set).

To ensure correctness, a hardware transaction reads the lock (line 12) and aborts, either immediately if it finds it locked, or if some concurrent thread acquires it in the fall-back path. This mechanism safeguards the correctness of fall-back executions that run without any instrumentation [30]. This is required because the thread executing in the software fall-back path has no monitoring in place that prevents inconsistent reads (i.e., no instrumentation). Consider for instance an application which preserves the invariant \( x = y \), and a thread that executes in the fall-back path and reads \( x = 0 \). If one let a concurrent hardware transaction write \( x = y = 1 \) and commit, then the fall-back path thread could read \( y = 1 \) inside the atomic section and violate the invariant. With the verification in line 12, pessimistic executions are safeguarded from inconsistencies because hardware transactions abort preemptively to avoid any such hazard.

2.2. RTM-SGL: optimizing statically RTM with a global lock fall-back

In this section we focus on RTM with a single global lock as the fall-back, which we call RTM-SGL. Before introducing the mechanisms for self-tuning the logic in charge of activating the fall-back, we look into two static optimizations that aim at improving the performance of RTM-SGL: (i) avoiding the, so called, lemming effect [31], and (ii) selecting an efficient lock implementation.

The lemming effect refers to scenarios in which one thread that executes using the fall-back (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
Table 3
Overhead (%) of each lock implementation (as the fall-back of RTM-SGL) with respect to the optimal choice in each execution. The results are the average across the STAMP benchmarks [33]. We show both the overhead in terms of performance lost, and in terms of cache misses.

<table>
<thead>
<tr>
<th>Lock implementation</th>
<th>Overhead wrt best (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Performance</td>
</tr>
<tr>
<td>Ticket [34]</td>
<td>1.0</td>
</tr>
<tr>
<td>MCS [35]</td>
<td>2.4</td>
</tr>
<tr>
<td>CLH [36]</td>
<td>2.9</td>
</tr>
<tr>
<td>Read-write (pthreads)</td>
<td>14.2</td>
</tr>
<tr>
<td>Test-and-test-and-set [37]</td>
<td>15.2</td>
</tr>
<tr>
<td>Spin [38]</td>
<td>16.4</td>
</tr>
</tbody>
</table>

threads to resume execution of transactions in hardware. A simple way to avoid this issue is checking the lock before starting the transaction, and waiting in case it is busy. An alternative way, relying on an auxiliary lock that is not read transactionally (contrary to the single global lock), was also proposed [32]; however, our experience shows that their gains are equivalent.

We also studied the impact of different implementations of the single global lock. The goal is to understand if there is some implementation that consistently performs above the average across all parallelism degrees and benchmarks. The results, shown in Table 3, are the average across the STAMP standard TM benchmarks [33]. There, we list the different locks according to their average overhead with respect to the best performing lock in each experiment (an idealized lock). We can see that the Ticket lock was the best performing and that this is tightly related with the overhead in terms of cache misses (measured with the perf tool in Linux). The Ticket lock seems to strike a balance between being lightweight (in contrast to MCS and CLH) and avoiding repeated expensive synchronization operations (such as compare-and-swaps for the other implementations).

In the rest of the paper, we shall use the two optimizations described above, in order to ensure the proper tuning of RTM-SGL and, hence, the representativeness of the evaluation of the proposed self-tuning mechanisms.

Finally, we also used the Thread-Caching Malloc memory allocator to reduce the contention between concurrent threads allocating memory, which was shown to be the best for Intel RTM [39]. This optimization is also used in all our experiments and variants of RTM.

2.3. RTM-NOrec: optimizing statically RTM with the NOrec STM fall-back

Although we drive our discussion with RTM-SGL, for its simplicity, our objective of self-tuning RTM is also applicable to other fall-backs such as one based on an STM. In this section we introduce one such example, RTM with NOrec [21] as the fall-back STM (which we refer to as RTM-NOrec), which we later also evaluate with our self-tuning proposal.

In STM NOrec there is a global lock, which is acquired during the commit of a transaction that contains new updates to shared memory. Furthermore, the lock also works as a logical clock, whose monotonically increasing values are used to notify concurrent transactions about updates to memory. As such, transactions keep a read- and write-set, and whenever they perform an operation in which they notice the global lock to have increased – remember it is also a logical clock – they re-validate the read-set. This ensures that all memory positions read up to that point are still up to date, and as such the transaction is still serializable in that moment. This same validation is performed at commit before writing the updates to shared memory (under the global lock).

The global lock of NOrec has the advantage that commit operations perform only one expensive operation (i.e., acquiring the lock), in contrast with other STMs that may perform many atomic operations [22,23]. On the other hand, this has the disadvantage of requiring running transactions to validate the whole read-set when the global lock increases: as some memory locations were updated, the transaction must review all its reads, as there is no clue as to which locations may have been invalidated (if any). This provides the correctness guarantee of opacity [40], which, informally, ensures that running transactions (even those that may eventually abort) observe a snapshot that could have been produced by a serial execution; this property is deemed as important to guarantee that concurrent applications never crash due to observing inconsistent states (which are prevented with opacity).

We now consider a HybridTM with NOrec as the fall-back path of RTM. We use the baseline algorithm proposed in Hybrid-NOrec [26] – which at the time was designed to operate with both AMD’s and Sun’s HTMs – and we evaluate several performance optimizations in the scope of Intel RTM, for the first time in the literature to the best of our knowledge.

The base idea is quite simple: replacing the software fall-back path of RTM-SGL, which relies on a single global lock for synchronization, with the NOrec software transactions. To ensure correctness between concurrent hardware and software transactions, it is necessary to have a mechanism similar to the lock subscription of RTM-SGL (presented in line 12 of Algorithm 1). The idea of HybridNOrec, which we summarize here, is to use NOrec’s single global lock to manage that interaction: hardware transactions are successful only when NOrec’s lock is not taken. This constrains concurrency less than the lock in RTM-SGL, because NOrec’s single lock is taken only during the commit of a software transaction (and not during the whole transaction); hence there is more room for parallelism between hardware and software transactions as the single lock is held for smaller
Table 4
Speedup of static configurations with 4 threads relatively to a sequential execution in sample benchmarks of our study.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>GCC</th>
<th>GiveUp-5</th>
<th>Best configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome</td>
<td>0.65</td>
<td>2.64</td>
<td>2.84</td>
</tr>
<tr>
<td>Intruder</td>
<td>0.73</td>
<td>2.48</td>
<td>3.05</td>
</tr>
<tr>
<td>K-means (high)</td>
<td>2.74</td>
<td>2.85</td>
<td>2.99</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>0.99</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>SSCA2</td>
<td>2.81</td>
<td>2.88</td>
<td>3.27</td>
</tr>
<tr>
<td>Vacation (high)</td>
<td>0.74</td>
<td>1.76</td>
<td>2.64</td>
</tr>
<tr>
<td>Yada</td>
<td>0.79</td>
<td>0.87</td>
<td>0.92</td>
</tr>
</tbody>
</table>

periods. Furthermore, update hardware transactions must also increment NOrec’s global lock – similar to what software transactions already do – to notify concurrent software transactions that some shared data was mutated.

In RTM-NOrec implementation used in this article we include three recently proposed optimizations for RTM-NOrec [26,41]:

• Hardware transactions keep a thread-local flag to register whether they have performed some write to shared memory. This entails that write operations are instrumented to set that flag to true. If not, then the hardware transaction need not increment NOrec’s lock, avoiding contention to it with concurrent transactions.

• Software transactions in NOrec increment a counter when they begin (and respectively decrement it when they finish) to register whether there is any software transaction alive. If not, then the hardware transactions need not increment NOrec’s lock.

• We have also instrumented the TM benchmarks with two code paths specifically tailored for hardware (and respectively software) execution of the atomic blocks. As such, the hardware path is faster because it does not incur instrumentation overheads for reads and writes. This is not true for the software path, as explained above for STM. As a result, in the beginning of an atomic block, we decide which code path to use given the mode chosen by RTM-NOrec.

These optimizations were recently shown to enhance significantly performance of RTM-NOrec in a wide variety of workloads [39]—which is the reason why we opted for including them in the RTM-NOrec implementation considered later on in this article.

3. Making the case for adaptation: no one-size fits all

As already mentioned the software-based transaction retry logic, either for the case of hardware or of hybrid TM implementations, can be governed via two main tuning knobs: the budget (i.e., the total number) of attempts available for hardware transactions, before resorting to the fall-back path, and how to consume such budget depending on the causes of transactions’ aborts.

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 two parameters:

• Budget of attempts—Varying from 0 to 20. This means the software handler may choose to avoid at all to call RTM, or to insist up to 20 times before stop calling it to execute a transaction successfully.

• Capacity aborts—From Table 2, it is possible to obtain feedback from RTM about capacity overflow aborts. In such cases we allow three possible configurations: GiveUp exhausts all attempts upon a capacity abort; HALVEN drops half of the attempts on a capacity abort; and STUBBORN decreases one attempt only. The objective is to have different behaviours according to how serious the impediments are to commit successfully in hardware.2

We tested all the 63 possible combinations of the above configuration in our suite of benchmarks, with varying concurrency degrees, reaching the conclusion that there is no one-size fits all solution. In Table 4 we show some of the results from these experiments. For the sake of brevity, we focus on some benchmarks only, and with 4 threads: we show the speedups (of the whole benchmark duration, relative to a non-instrumented sequential execution) of the best configuration, GCC (Algorithm 1) and a policy which we call GiveUp-5 (see Algorithm 2). The GiveUp-5 policy is inspired by recent works, which have used a static value of 5 attempts in their configuration [14,27]. Also, as the name suggests, this policy resorts to the fall-back path upon the first occurrence of a capacity abort. We consider this configuration in our study in order to include a static policy that takes reasonable choices regarding the setting of the retry budget and the management of capacity exceptions.

In general, the GiveUp-5 algorithm yields some considerable improvements over GCC. However, the most important results are on the rightmost column, where we can see that the best performing variant varies significantly in its characteristics. For instance, in the SSCA2 benchmark there are very few aborts due to the limitations of RTM, so the best configuration is stubborn

2 We focus on capacity aborts because this is the only specific reason placed in the EAX register that may represent a deterministic impediment for a hardware transaction to succeed. In our experience, abort reasons captured by events in model specific registers (MSRs) are too expensive to monitor online in a fine-grained fashion because reading them implies issuing system calls.
Algorithm 2 Possible static configuration of RTM (we call it GiveUp-5).

1: int attempts ← 5  ▶ increased budget of attempts to sustain spurious aborts
2: while(is_locked(global-lock)) do x86_pause ▶ avoid the lemming effect
3: int status ← xBEGIN
4: if status ≠ ok then
5: if attempts = 0 then
6: acquire(global-lock)
7: goto line 17
8: else
9: if status = capacity then
10: attempts ← 0
11: goto line 6
12: else
13: attempts−−
14: goto line 2
15: if is_locked(globalLock) then
16: XABORT
17: ▶ ...application transactional code runs here
18: if attempts = 0 then
19: release(global-lock)
20: else
21: XEND

![Speedups](image)

**Fig. 3.** Speedups – across all benchmarks and number of threads evaluated in this paper – of different RTM configurations with respect to the optimal configuration for the considered experiment. For each data point we consider four statistical metrics over the speedups: the best, median, average and worst speedups that each configuration obtained across all the experiments (benchmarks and number of threads).

with respect to capacity aborts and insists quite a lot in hardware transactions (10 retries). In contrast, Yada does trigger many deterministic capacity aborts, so it is best to desist immediately upon their occurrence; however, if the transaction aborts for other reasons, it is actually good to insist also quite a lot (7 retries).

Overall we can see that the optimal configuration varies significantly across different benchmarks. The complexity of identifying a single static configuration emerges even more clearly when considering Fig. 3, which reports speedups across all benchmarks considered (several data-structures and the STAMP benchmarks [33], described further in Section 8) and number of the threads (from 1 to 8). Each speedup is the performance of some RTM configuration, relative to the optimal configuration for the considered experiment (identified off-line via an exhaustive exploration). The tested configurations result from the cross product of the three aforementioned ways to deal with capacity aborts and by varying the number of retries from 1 to 20 (described in Section 3).

By analysing the experimental data reported in Fig. 3, it is possible to see that the performance loss with respect to the optimum for any configuration can range from 40% to 10×. Furthermore, all configurations performed quasi-optimally in some benchmark and parallelism degree, as evidenced by the data points of the best line shown. Overall, this experimental data,
4. Self-tuning Intel Restricted Transactional Memory

Our approach to tackle the problem of “no one-size fits all” in the RTM software fall-back is to perform online lightweight profiling and adaptation. We use a simple self-tuning feedback loop based on a target performance metric and instantiate reinforcement learning algorithms that guide the exploration of the possible configurations towards the optimal one. This allows to better fit the workloads of typical irregular applications that benefit most from synchronization with TM [42], for which fully offline optimizations are likely to fail 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 [43,44].

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 solution seeks to minimize overhead in a two-fold way. First, it avoids any synchronization among concurrent threads, and it employs simple metrics for performance sampling (such as × 86’s Time Stamp Counter (TSC) cycle counter). 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 [45] for a classical reinforcement learning problem, the multi-armed bandit [46]. This reinforcement learning technique is the key building block of the mechanism that we use to adapt the policy used to deal with capacity aborts, which will be described in Section 4.2. We then explain the adaptation of how stubborn should one be in using RTM, i.e., the budget of attempts, in Section 4.3. The combination of those techniques is presented in Section 4.4.

4.1. Bandit problem and UCB

The “bandit” problem (also known as “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 independent 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 $i$ ($0 \leq i < K$) – where $K$ is the number of arms – 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 ($\mu_i$) computed as

$$\mu_i = \frac{1}{n} \sum_{n=1}^{\infty} X_{i,n}.$$  \hspace{1cm} (1)

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 $\mu^*$ is the average obtained by the optimal lever. Several algorithms have been studied to minimize this regret (i.e., loss with respect to the optimal), defined as

$$\mu^* n - \mu_i \sum_{i=1}^{K} E[n_i],$$  \hspace{1cm} (2)

where $n_i$ is the number of times arm $i$ has been chosen, given that there are $K$ total arms.

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 i with a value:

\[
\tilde{\mu}_i = \bar{x}_i + \sqrt{\frac{2 \log n}{n_i}}.
\]  

(3)

where \(\mu_i\) 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 \(\mu_i\), the algorithm searches for the option with the highest reward, while minimizing the regret along the way.

As such, the intuition behind UCB is to balance the trade-off between exploring new choices versus exploiting those already known. The advantage of UCB is in providing bounds on the errors committed over time when searching for the optimal choice.

4.2. Using UCB learning

We applied UCB by considering that each atomic block of the application is associated with a bandit, i.e., a corresponding UCB instance. With it, we seek to optimize the choice of what to do when a capacity abort occurs in a hardware transaction. In some sense, this models a belief on whether the capacity aborts witnessed are transient or deterministic, a fact that cannot be established correctly based only on the error codes returned by aborted transactions.

We tackle the issue of how to manage capacity aborts by gathering feedback on the performance yielded when using the three options identified in Section 3, i.e., (i) decrementing linearly the number of retries, (ii) halving the number retries, and (iii) using the fall-back path upon the occurrence of the first capacity abort. We associate each option with a bandit lever and use the UCB algorithm to solve the trade-off between exploration and exploitation, i.e., testing strategies that appear to be sub-optimal based on available evidence, versus selecting the strategy that, so far, yielded on average the best performance.

The rationale that motivates the choice of using only these three alternative strategies is to keep the number of UCB levers low. This is a relevant design, as the higher the number of bandit’s levers, the longer time UCB will spend exploring sub-optimal solutions.

In order to instantiate Eq. (3), we associate with n the number of decisions taken so far for the atomic block, and with \(n_i\) the number of times the UCB instance chose lever i. As for the reward function associated with the levers (represented by \(\bar{x}_i\)), we consider the number of processor cycles that it takes to execute (i.e., commit, after possibly some retries) the atomic block using each different lever/strategy. To this end, we associate a counter \(c_i\) with each lever that we use to track how many cycles were consumed so far, i.e., to execute \(n_i\) times the atomic block using lever i. We also keep track of the best (i.e., lowest) number of cycles so far to process atomic block i as \(\text{best}_i\). Then, we compute the reward \(\bar{x}_i\) for lever i with

\[
\bar{x}_i = \frac{\text{best}_i}{c_i/n_i}.
\]  

(4)

which means that we normalize the cycles of lever i, giving us a reward in the interval [0,1]. If a given configuration leads to executing an atomic block fast enough (i.e., close to the best number of cycles spent so far) then this produces a high reward. In contrast, for slow executions, the average number of cycles in the denominator will be larger than the best recorded value on the numerator, and thus the reward will be low.

4.3. Using gradient descent exploration

The other space of configurations that we want to optimize is the number of retries to use in hardware transactions before triggering the fall-back software path. The optimization problem that we want to solve here is illustrated by the experiments in Fig. 4, where we show the performance improvement of using RTM with 8 threads relative to a sequential version of the code (with no instrumentation), in the TM benchmark Vacation [33], when varying the number of attempts for the configuration. In the plot we show two workloads for Vacation that generate low and high contention, for which there are significantly different number of attempts yielding maximum values of improvement (namely, 12 and 16).

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 [47]. Its idea is to perform an iterative search, in which each step consists of evaluating some gain function: if the last change (i.e., increment or decrement by one unit) of the budget of retries produced an improvement, then the same change is applied again.

The alternative of using UCB was dismissed because the parameter – number of attempts – has a large space of search that makes UCB quite cumbersome (e.g., UCB requires an initial, uniform sampling of every possible configuration).

To implement the function that measures the gain we use also the processor cycles that it takes to execute the atomic block. We augmented the classical gradient descent with probabilistic jumps to avoid getting trapped in local maxima during the exploration. This is because these classical techniques end up exploring neighbouring configurations and may not be able to escape a local maxima that is surrounded by worse configurations—even though a global optimal configuration may exist elsewhere. Furthermore, we memorize the best configuration seen so far to recover from unfortunate probabilistic jumps that yield excessive performance degradation.
To implement all of this, we store in our library: the best configuration and performance seen so far (best); the last configuration and corresponding performance (last); and the current configuration (current). Note that 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. As we shall see in the next section, when merging the usage of gradient descent (to configure the number of retries) with that of UCB (to configure the reaction to capacity aborts), we actually create a rule to decide better the extent and direction of these probabilistic jumps.

As we shall see, in the implementation of these techniques in Section 6, we perform this search/optimization step with some periodicity of a number of transactions. That is, we do not explore a new configuration on every new transaction but, instead, let several (e.g., a hundred) transactions execute before triggering a new exploration.

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 $min/\Delta_1$, 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 $min/\Delta_1$, we set them respectively to 1%, 40%, and 5%, which are typical values for this type of algorithms [46] and whose appropriateness in the considered context will be assessed in Section 8.

4.4. 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 partially 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 RTM. This may be, for instance, dependent on the current state of shared memory, or some input parameters 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 RTM 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 $ab$-$other$:

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

where $cur$ is the current configuration of the budget of attempts and $maxTries = 20$. If UCB is different from Stubborn, the jump has negative direction, and length:

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

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, with each individual strategy using the respective static configuration from GiveUp-5 (i.e., UCB uses 5 attempts and Grad uses GiveUp)—we average the results across benchmarks since they yielded consistent results.

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.

5. Granularity of tuning

In this section we consider the trade-offs in possible designs of our proposal for self-tuning RTM.

On one extreme, it is possible to self-tune RTM for each atomic block defined in the application. This fits better applications with high heterogeneity of atomic blocks: it is conceivable to imagine that one atomic block $a_1$ defined by the programmer may execute very fast and with a small footprint, whereas another one $a_2$ in the same application is very large and often exceeds the capacity of RTM. As such, $a_1$ may be optimized to have a high budget of attempts and not to insist upon capacity aborts, whereas $a_2$ may immediately give up RTM when faced with a capacity abort.

On the other extreme, we may self-tune the application as a whole, independently of its atomic blocks. In contrast with the alternative above, this has the advantage of requiring less metadata: a single UCB and gradient instances, instead of one per atomic block. As such, it also takes less executions of each atomic block to gather statistically meaningful data to conduct the reinforcement learning procedures, as they are all gathered into the same metadata.

By avoiding multiple concurrent optimizations, this design has the benefit of avoiding possible interference. As an example, atomic block $a_1$ may be experimenting with a strategy that causes it to give up and acquire the global lock often (in the case of RTM-SGL). As a result, now atomic block $a_2$ is often executed concurrently with $a_1$ and $a_1$ gets aborted in hardware frequently due to the lock being taken by $a_1$, and this may mislead the optimizer used by $a_2$. Due to the interference of $a_1$’s optimizer, in fact, $a_2$’s optimizer may associate a low reward with the strategy, say $s$, currently in use for $a_2$. However, if $a_1$ eventually were to converge to a different strategy, strategy $s$ may eventually become optimal for $a_2$ and have, at steady state, a higher reward. Indeed, by performing self-tuning at the granularity of the whole application (and not of the individual atomic blocks), we avoid this sort of interferences, although at the cost of a reduced tuning flexibility.

Finally, it is also possible to perform the optimizations on a per-thread basis, or globally across threads. This raises similar trade-offs to those above, as multiple concurrent optimizations may have undesirable side-effects that mislead the other reinforcement learners. Conversely, a single global optimization across all threads may fail to capture heterogeneity in the threads: it is conceivable to imagine an application in which some threads execute atomic block $a_1$ with different parameters that cause them to have heterogeneous access patterns for the same code in $a_1$. As such, different threads may have very different behaviours within $a_1$, for which different RTM usages are optimal.
To capture the different trade-offs of these designs, we create two versions of our proposal. We use the name of Tuner, on which we have focused so far, for the algorithm that optimizes each atomic block independently in a per-thread fashion. In contrast, we use the name G-Tuner for a ‘global’ version of the algorithm that optimizes the whole application (considering all atomic blocks as one) and with all threads using the same learner and following the same optimization.

6. Implementation details

In this section we provide additional details on our implementations. We focus on the case of RTM-SGL, for its simplicity, although the principles that we explain are the same for the application of the proposed tuning strategies to RTM-NOrec. We begin by explaining how we integrated our algorithms 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 [29] in GCC that allows programmers to write atomic constructs in their programs, which are compiled to contain calls to our TM run-time library.

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. This is an implementation of the TM standard ABI [48], which compilers use, thus allowing our contributions to also being applicable in a portable fashion to other target architectures.

Our initial expectation was that we could craft a RTM based run-time relying on our algorithms 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 RTM benefits from merging the irrevocability lock with the fall-back lock. Furthermore, the choice of integrating our algorithms into GCC allows achieving total transparency and maximum ease of use for the programmer.

We now consider our Tuner, which uses per atomic block statistics and configurations. For this, we modified GCC to uniquely identify each atomic block, and to pass that information to calls to the TM run-time. We begin by laying out a high-level description of Tuner in Fig. 6. To the purpose of this presentation, G-Tuner shares most of its implementation with Tuner, thus, despite their antagonistic design choices, we describe mostly how to integrate Tuner. We then describe the details that differ among the two.

The flow in Fig. 6 starts every time a thread enters an atomic block, at which point the corresponding metadata of the atomic block is fetched, by using its unique id. Every per atomic 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).

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—we set this period to 100 transactions in our evaluation. This classic technique allows to keep the overheads low without missing noticeable accuracy in the decisions taken [49–51]. 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 Algorithm 3. Lines 8–19...
Algorithm 3 Tuner applied to RTM-SGL.

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: goto line 21
12: else
13:   if status = capacity then
14:     → set attempts according to ucbBelief
15:   else
16:     attempts ← attempts - 1
17:     if attempts = 0 then
18:        goto line 9
19:     goto line 5
20: if is_locked(global-lock) then XABORT
21: → ...application transactional code runs here
22: if attempts = 0 then
23:  release(global-lock)
24: else
25:  XEND
26: if reoptimize() then
27:   long totalCycles ← obtainRDTSC() - initCycles
28:  ucbBelief ← ucb(totalCycles)
29:  attempts ← grad(totalCycles)

3 We note that in our evaluation setup, which has 8 hardware threads, we have confirmed that not binding threads to cores does not produce a statistically measurable difference in performance. This, however, may not remain true in machines with architectures different from the one used in this paper.
7.1. Granularity of the self-tuning

In this work we presented self-tuning proposals that adapt the choice of the RTM fall-back for a given (or all) atomic blocks presented in the source-code. This works best in the case each atomic block is often invoked from the same context, i.e., its call stack and relevant scoped variables are the same.

Let us present a contrasting example to explain why that is the case. Suppose that a concurrent red-black tree is written with an atomic block around an insert operation (as is the case in our evaluation following up). Then, an application may instance two such red-black trees in the same piece of code and then populates them with contrasting workloads: one tree may be very small and the other very large. As a result, the same insert operation will now be invoked over these different trees, and the resulting workload will be dramatically different as one will typically run fine in RTM whereas the other may fail deterministically due to capacity aborts.

While this limitation motivates for a different granularity for the self-tuning – for instance, atomic blocks that consider also the call-stack – we note that this kind of behaviour does not occur in a measurable way on the reference benchmarks for TM systems that are used in the literature (and in our evaluation). Namely, it is often the case that an atomic block is invoked from the same code location, generating identical workloads in subsequent invocations. Even if the workload changes, as is the case for some STAMP benchmarks [33], this happens gradually and not with concurrent inter-leaved requests exhibiting different workload characteristics as was the case for the example given above.

A possible solution to address this issue is to use the approach proposed by Dice et al. [52], which consists of labelling atomic blocks with context identifiers that capture invocations from different contexts in the code. This allows a given static atomic block – i.e., at the source code level – to be seen as different instances at run-time. The integration of this technique with TUNER is perfectly viable, and would enable different optimizations of the same atomic block depending on the context in which it is invoked.

7.2. Bootstrapping the self-tuning process

When presenting TUNER we argued for the use of thread-local variables to maintain the metadata necessary for the self-tuning process. This approach has two main advantages. First, it allows threads to perform self-tuning in an independent fashion. This can be beneficial for applications in which threads generate heterogeneous workloads and, consequently, have different optimal configurations. Second, our distributed self-tuning approach is designed to avoid any inter-thread synchronization.

However, for applications that often spawn new threads, this poses the problem that such threads basically restart the learning process by collecting statistics from scratch. For such cases, which are not encompassed in our evaluation test-bed, we propose a simple idea borrowed from G-TUNER: we can periodically gather statistics across the threads into a centralized metadata, and a new thread may use that to bootstrap its self-tuning.

7.3. Workload changes

The self-tuning approach taken here is theoretically tailored to stationary workloads. This is because the UCB solution, which we employed to decide what to do upon capacity aborts, assumes a constant reward function. In fact, some of the STAMP benchmarks that we evaluate our solution with, have dynamic workloads that change over time. As we shall see, our solution performs efficiently (compared to any static solution) even in those cases.

Regardless, it is worth highlighting other alternatives that have the potential to perform as good (or even better) with more suitable theoretical guarantees. One alternative would be to use a different solution to the bandit problem that accounts for possible changes in the workload [53,54]. The idea there is to consider the data collected on the levers of the bandit over a sliding window of time. Alternatively, we could use a workload change detector, such as CUSUM [55], and reset the UCB statistics upon a workload change.

8. Evaluation study

We now present our final experimental study, in which we assess the effectiveness of our self-tuning proposal when applied to Intel RTM. We shall consider both RTM-SGL and RTM-NOrec, which were described in Sections 2.2 and 2.3, enhanced with both TUNER and G-TUNER, as described in Sections 3 and 6.

We note that we have only implemented the RTM-SGL approaches (both for TUNER and G-TUNER in libitm in GCC). For approaches with RTM-NOrec, instead, we resorted to a macro-based library that the applications invoke directly. The main reason being the lack of support in libitm for HybridTMs as they require two instrumentation paths—besides the non-instrumented one. We highlight that our RTM-SGL implementations yield equivalent performance both when used in libitm in GCC and also in a macro-based library. As such, we believe that the same would be possible for RTM-NOrec.

We resort to the baselines that were described throughout this paper, together with a state of the art approach that was designed in the scope of STMs:

- GCC: corresponding to Algorithm 1, which is the implementation available in libitm in GCC 4.8.2.
- ADAPTIVELOCKS: proposed to decide between locks and TM for atomic blocks [50]; 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.
Fig. 7. Overhead of the self-tuning algorithms. For each algorithm, we used a stripped down version in which all the profiling and optimizations are performed, but the RTM configuration used is always the same. We compare the performance with a baseline that uses that same static configuration and show the geometric mean speedup across all benchmarks (data-structures and STAMP).

- **GiveUp-5**: corresponding to Algorithm 2, which embodies the best practices described in the literature to statically tune RTM with heuristics. We applied this baseline to both RTM-SGL and RTM-NOrec.
- **Stubborn-10**: unlike GiveUp-5, this heuristic insists on using RTM for 10 times independently of the exception type that caused the transaction to abort. Then, it switches to using the fall-back path. Also in this case we consider two fall-back paths, namely SGL and NOrec.
- **Best Static**: an idealized upper bound on the best result possible, obtained by picking the best configuration among all those possible 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 specifically tailored for each workload/benchmark. We show the results using this ideal variant both for the case of RTM-SGL and RTM-NOrec.

Similar to all the results shown, we shall present speedup values that are computed with respect to a sequential, non-synchronized execution of each benchmark. Every experiment was repeated ten times to achieve statistically meaningful results.

We begin by summarizing our findings across all benchmarks in Section 8.1. Then, in Section 8.2, we study the performance of the different solutions in a set of concurrent data-structures, which are widely used to study TM algorithms. Afterwards, in Section 8.3, we extend our study to a variety of popular TM standard applications. Finally, we also evaluate the algorithms from an energy-efficiency perspective in Section 8.4.

8.1. Summary of evaluation

To ease the interpretation of all our results in this extensive study, we begin by summarizing our findings across all the benchmarks that we use (namely, concurrent data-structures in Section 8.2 and the STAMP benchmarks in Section 8.3), whose detailed description is provided later in the following sections.

First of all, we sought to understand the overhead of our self-tuning approaches. For that, we created variants of our Tuner and G-Tuner (applied to both RTM-SGL and RTM-NOrec) that executed all the profiling and optimization procedures, even though the atomic blocks were always executed with a static configuration. We compared the performance of these stripped down tuning algorithms with a baseline that used that same static configuration and had none of the added procedures. The results, which illustrate the overheads of our proposed solution, are shown in Fig. 7.

This experiment shows that the overhead of G-Tuner is slightly less than Tuner, although in any case the values amount at most to an average of roughly 5%. This is an inspiring low value, which we hope to capitalize with the self-tuning produced by these algorithms.

We then present the average speedup across all benchmarks, for each approach, relative to that of Best Static. As such, values closer to 1 are optimal. The results for RTM-SGL and RTM-NOrec are presented, respectively in Tables 5 and 6. The main result that we highlight here is two-fold: (1) there is a large gap between the performance of the baselines and that of the Best Static variant, regardless of the used fall-back path; and (2) both Tuner and G-Tuner are capable of effectively minimizing that gap, yielding an average performance that is very close to that of the Best Static variant.

Note that the worst performing approaches (namely, GCC and AdaptiveLocks) tend to have higher standard deviation, because in some benchmarks they perform closer to the optimal (e.g., Labyrinth and SSCA2) and in others they can perform much worse
(e.g., Vacation and Intruder). In contrast, the best performing approaches tend to perform more consistently throughout the benchmarks.

In fact, for 8 threads, we can see that our self-tuning proposals achieve performance within 5% of the Best Static across both types of fall-backs. The data appears to suggest that TUNER performs slightly better in RTM-SGL and G-TUNER in RTM-NOrec. We argue that this may depend on the inherent trade-offs that exist between the two proposed solutions:

- In RTM-SGL, bad configurations for triggering the fall-back are more catastrophic, because dictating the usage of the fall-back imposes heavy serialization on the single global-lock. As such, it pays off more to have a finer-grained optimization with TUNER, even though the overhead of doing so is higher.
- In RTM-NOrec, the fall-back is less aggressive in constraining concurrency as it uses an STM rather than a global lock. As such, it is better to have a more coarse-grained (and lighter) optimization, by relying on G-TUNER instead.

Although these antagonistic design choices pose a trade-off, the aggregated performance values are fairly close to each other. In the following sections we shall delve into detailed results for each benchmark, where the differences become noticeable. Furthermore, it is important to recall that TUNER allows for self-tuning workloads that are heterogeneous across threads, which does not happen in these typical TM benchmarks. We have verified this in synthesized benchmarks: for instance, one benchmark where some threads manipulate a low contended hash-map and others a contended linked-list; in such case TUNER obtained about 20% improvement over G-TUNER for 8 threads. However, for cases where this heterogeneity between threads is not expected, then the simpler algorithm for G-TUNER provides approximately the same performance, which makes it more appealing.

Finally, we do not specifically aim to adapt between the different fall-backs – namely between using both the SGL and NOrec – for which reason it is out of scope to compare directly their performance here. In fact, others have proposed and evaluated the trade-offs between such approaches in the scope of Hybrid TMs [26,41,52,56].

### 8.2. Concurrent data-structures

We tested a set of four representative data-structures that are typically used to study the performance of TM algorithms [21–23,57]. The implementations of these data-structures were based on those of Synchrobench [58], a recent work that studies the performance of TM algorithms (among others) in concurrent-data structures. We used two workloads for each data-structure, respectively with low and high contention. The parameters used are described in Table 7, and were chosen such that the low (and respectively high) contention across data-structures generated approximately the same ratio of aborted transactions (less than 10% for low contended, and more than 50% for high contended).

We present our results concerning RTM-SGL in Fig. 8 and RTM-NOrec in Fig. 9. The highlight of most of our evaluation is that our self-tuning approaches are generally the closest ones to the idealized Best Static variant; not only that, but the gap between the optimal performance and our solution is typically very small.

In general, the static baselines can perform significantly worse than the best variant. This is an expected result, as we have already shown how difficult it is for a static approach to perform optimally across workloads. The performance of AdaptiveLocks, on the other hand, is largely unsatisfactory due to the large, constant, overheads that it entails. This approach, in fact, requires

<table>
<thead>
<tr>
<th>Table 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup of each approach in RTM-SGL relative to the Best Static (with standard deviation) averaged across all benchmarks.</td>
</tr>
<tr>
<td>Speedup to best static Threads</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>GCC</td>
</tr>
<tr>
<td>AdaptiveLocks</td>
</tr>
<tr>
<td>GiveUp-5</td>
</tr>
<tr>
<td>STUBBORN-10</td>
</tr>
<tr>
<td>TUNER</td>
</tr>
<tr>
<td>G-TUNER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup of each approach in RTM-NOrec relative to the Best Static (with standard deviation) averaged across all benchmarks.</td>
</tr>
<tr>
<td>Speedup to Best Static Threads</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>NOrec</td>
</tr>
<tr>
<td>GiveUp-5</td>
</tr>
<tr>
<td>STUBBORN-10</td>
</tr>
<tr>
<td>TUNER</td>
</tr>
<tr>
<td>G-TUNER</td>
</tr>
</tbody>
</table>
Fig. 8. Speedups relative to sequential execution in the data-structures benchmarks when tuning RTM-SGL with different approaches.
Fig. 9. Speedups relative to sequential execution in the data-structure benchmarks when tuning RTM-NOrec with different approaches.
manipulating shared memory positions to store metadata associated with the atomic blocks. These manipulations require costly synchronization that hampers performance significantly and shadows the gains achievable via dynamic adaptation.

If we consider the two heuristics GiveUp-5 and Stubborn-10, we can see that they usually have contrasting results. That is, whenever one is closer to the Best Static, the other performs worse, and vice versa. This is a consequence of their antagonistic heuristics for budget of attempts and to deal with capacity aborts.

If we compare our two tuning approaches, we can see across all the plots that they perform very similarly in general. There are some sporadic advantages towards G-Tuner that yield a geometric mean improvement of 5% over Tuner, which happens for two main reasons. Firstly, there are three atomic blocks only in these data-structures, one for each type of operation (and two of which are very similar, mutation operations). As such, the G-Tuner approach to optimize the application globally does not lose much with respect to the specialization of Tuner because there are few and homogeneous atomic blocks. Secondly, these atomic blocks are very small compared to those of some applications that we test later. Consequently, Tuner’s main disadvantage, i.e., replicating the tuning procedure across different threads, becomes more noticeable. This is mainly a result of measuring the processor cycles spent, which adds a constant overhead of roughly 65 cycles in our machine, and is thus visible in such small and dominant atomic blocks.

While it is not our intention to compare the different RTM fall-backs, we can see between the two sets of plots that the scalability and performance trends are very similar. This is a result of the fact that the best performing algorithms allow both RTM approaches to rely on hardware transactions. We have ourselves assessed that concurrent data-structures are well suited for hardware transactions [59].

As a last set of experiments with regard to the data-structures, we also present the configurations for which our Tuner approaches converged to. This is a particularly interesting opportunity to delve into such details because these data-structures have stationary workloads and a small number of atomic blocks when comparing to the benchmarks evaluated later.

Because each thread in Tuner may choose a different configuration, we present the configuration that is chosen by most threads in a given run. In practice, given the homogeneous workload of these micro-benchmarks, we noticed that most configurations were similar across the threads. Furthermore, we present the configurations that were used for the longest time in the benchmark, as our self-tuning algorithms regularly re-optimize, and may change their decision over time. Finally, these results correspond to the case of running with 8 threads, as that is the case where the gains were generally larger with our self-tuning proposals.

The results are shown in Tables 8 and 9, respectively for RTM-SGL and RTM-NOrec. Note that, as explained above, we show three configurations for Tuner corresponding to the three atomic blocks used in the data-structures (the canonical insert, remove and contains operations). In the case of G-Tuner, a single configuration is chosen globally for the application.

### Table 7
Parameters used in the data-structures tested. The low (and respectively high) contended workloads across data-structures were chosen in ways such that they generate approximately the same degree of contention.

<table>
<thead>
<tr>
<th>Benchmarks (contention)</th>
<th>Size</th>
<th>Insert (%)/remove (%)/contains (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked-list (low)</td>
<td>1000</td>
<td>1/1/98</td>
</tr>
<tr>
<td>Linked-list (high)</td>
<td>1000</td>
<td>37/37/26</td>
</tr>
<tr>
<td>Skip-list (low)</td>
<td>5000</td>
<td>1/1/98</td>
</tr>
<tr>
<td>Skip-list (high)</td>
<td>10,000</td>
<td>45/45/10</td>
</tr>
<tr>
<td>Red-black tree (low)</td>
<td>100,000</td>
<td>5/5/90</td>
</tr>
<tr>
<td>Red-black tree (high)</td>
<td>1000</td>
<td>45/45/10</td>
</tr>
<tr>
<td>Hash-map (low)</td>
<td>10,000</td>
<td>5/5/90</td>
</tr>
<tr>
<td>Hash-map (high)</td>
<td>100</td>
<td>45/45/10</td>
</tr>
</tbody>
</table>

### Table 8
Configurations to which Tuner and G-Tuner converge when using RTM-SGL and running with 8 threads.

<table>
<thead>
<tr>
<th>Benchmarks (contention)</th>
<th>RTM-SGL Tuner</th>
<th>RTM-SGL G-Tuner</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insert /Remove/Contains</td>
<td>Insert /Remove/Contains</td>
</tr>
<tr>
<td>Linked-list (low)</td>
<td>GiveUp-4/GiveUp-2</td>
<td>GiveUp-9/GiveUp-8</td>
</tr>
<tr>
<td>Linked-list (high)</td>
<td>GiveUp-6/Halven-4</td>
<td>Halven-5/Halven-7</td>
</tr>
<tr>
<td>Skip-list (low)</td>
<td>Halven-6/Halven-5</td>
<td>Stubborn-5/Halven-8</td>
</tr>
<tr>
<td>Skip-list (high)</td>
<td>GiveUp-8/GiveUp-6</td>
<td>Halven-15/GiveUp-15</td>
</tr>
<tr>
<td>Red-black tree (low)</td>
<td>Halven-7/Halven-7</td>
<td>Stubborn-6/Halven-8</td>
</tr>
<tr>
<td>Red-black tree (high)</td>
<td>GiveUp-8/GiveUp-8</td>
<td>Stubborn-8/Halven-9</td>
</tr>
<tr>
<td>Hash-map (low)</td>
<td>Halven-8/Halven-7</td>
<td>Stubborn-9/Halven-9</td>
</tr>
<tr>
<td>Hash-map (high)</td>
<td>GiveUp-6/GiveUp-6</td>
<td>Halven-5/GiveUp-8</td>
</tr>
</tbody>
</table>
### 8.3. Application benchmarks

For representative application benchmarks we relied on the STAMP suite [33], which was devised specifically for TM evaluation. We used the standard parameters for the STAMP benchmarks and show workloads for low and high contention when available.

Once again, we present our study with respect to both RTM-SGL (in Fig. 10) and RTM-NOrec (in Fig. 11). In general this large set of experiments indicates a consistent 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 Intruder (in Fig. 10(e)) – 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 RTM. In short, these gaps in performance between the static alternatives and the best variant possible correspond exactly to the room of improvement that we try to explore with our self-tuning approaches in this paper.

In fact, both TUNER and G-TUNER are able to achieve performance improvements in all benchmarks with the exception Labyrinth, in which it yields roughly the same performance as the static approaches. Note that, in Labyrinth, transactions are always too large to execute in hardware, and the benchmark executes about 500 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.

We also note that the CPU considered in our study is limited in terms of hardware parallelism: in fact, sometimes going over 4 threads is not profitable as hyper-threading is not beneficial due to the extra pressure on L1 caches [59]. This, however, is an issue that has been tackled by related work (e.g., [60]) and whose importance shall be relatively diminished by the availability of new hardware to be released with a larger number of physical cores.

It is also interesting to see that there are some experiments where the self-tuning approaches perform better than what we call the best configuration. This is a consequence of the best configuration being devised from static configurations; we choose the one that performs best among all those that are possible. In contrast, our self-tuning approaches change the configuration during the execution, possibly obtaining a combination of configurations at run-time that performs best than any single configuration. We can observe this phenomenon in K-means, Vacation and Yada.

Finally, we present an example of the adaptation performed by TUNER in Fig. 12 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 RTM whereas the right one does not. This illustrates the advantages that we have previously mentioned for TUNER: (1) the adaptation allows heterogeneous

---

#### Table 9

Configurations to which TUNER and G-TUNER converge to when using RTM-NOrec and running with 8 threads.

<table>
<thead>
<tr>
<th>Benchmarks (contention)</th>
<th>RTM-NOrec</th>
<th>G-TUNER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Insert</td>
<td>Remove</td>
</tr>
<tr>
<td>Linked-list (low)</td>
<td>GiveUp-9</td>
<td>GiveUp-6</td>
</tr>
<tr>
<td>Linked-list (high)</td>
<td>GiveUp-6</td>
<td>GiveUp-6</td>
</tr>
<tr>
<td>Skip-list (low)</td>
<td>HALVEN-4</td>
<td>HALVEN-5</td>
</tr>
<tr>
<td>Skip-list (high)</td>
<td>GiveUp-1</td>
<td>GiveUp-6</td>
</tr>
<tr>
<td>Red-black tree (low)</td>
<td>GiveUp-6</td>
<td>HALVEN-6</td>
</tr>
<tr>
<td>Red-black tree (high)</td>
<td>GiveUp-6</td>
<td>HALVEN-6</td>
</tr>
<tr>
<td>Hash-map (low)</td>
<td>HALVEN-7</td>
<td>HALVEN-7</td>
</tr>
<tr>
<td>Hash-map (high)</td>
<td>GiveUp-16</td>
<td>HALVEN-7</td>
</tr>
</tbody>
</table>

---

4 Note that in Labyrinth (Fig. 10(f)) the GCC and Best Static lines are overlapping, as the GCC heuristic gives up very easily, making it a nice fit for the long running transactions of this benchmark that lead to regular hardware capacity aborts.
threads and atomic blocks to converge to different configurations; (2) an atomic block, such as that in Fig. 12(b), can still insist moderately on using RTM as long as capacity aborts do not occur, but react quickly in case they appear. In contrast, G-Tuner uses a slightly more efficient profiling approach (with less redundant work across the threads) at the cost of having a single optimizer for all the application. As such, this results in less performance in cases such as Yada, where it is possible to benefit from the heterogeneity of the atomic blocks.

To better show this duality among our self-tuning approaches we show, in Fig. 13, the throughput of the different algorithms over time in two benchmarks for a sample of time in their initial execution (with 8 threads). It is interesting to see that G-Tuner tends to stabilize quicker than Tuner, although possibly at a lower throughput rate. Tuner produces a spikier throughput due to the interactions among different threads and different atomic blocks exploring the different configurations, which has an impact in the stabilization of the algorithm. Nevertheless, this has the advantage that, in some cases, it may achieve an average performance that is higher than G-Tuner (as is the case for Vacation in Fig. 10(g)).

In contrast, we can see that the static baselines have a steady performance, albeit far worse in absolute terms. AdaptiveLocks acts accordingly to an analytical model, whose impact in the configuration choice is visible as performance varies. However, its constant overheads are too high for those explorations to pay off.
8.4. Evaluating energy consumption

To conclude our evaluation study, we also assessed the energy efficiency of the different algorithms. So far we have focused on optimizing performance, but shifting the focus towards energy saving may, at least in principle, dictate a different optimization. To a large extent, the trade-off between optimizing for energy or time is a consequence of the trade-off between the usage of locks in the software fall-back and their absence in the hardware path.

However, optimizing energy consumption is not an easy task to implement. There is commodity facilities in recent Intel processors, called RAPL [61], which allow to measure the energy consumed at any point. Recent studies [62,63] show that the model used by Intel RAPL estimates quite accurately the power consumption, when compared to a power meter attached to the machine. Hence, this seems a promising tool to allow for practical energy consumption measurement and optimization.

One first problem is that the refresh rate of RAPL is not very high – of the order of milliseconds. Adding to this, RAPL uses model specific registers, which are exposed through the file system and require expensive calls to read and compute the energy consumed. As a result, these problems mean that measurements must be conducted over large periods of time, so that the
limitations are avoided and costs are amortized, when compared to the exploration/exploitation optimizations that we propose in this paper.

Given these challenges, which seem hard to overcome with current commodity hardware, we hypothesized to circumvent the problem: can we optimize a TM application in terms of performance and, as a side effect, also optimize the application in terms of energy efficiency?

To answer this question, we conducted a series of studies. As a first experiment, we took every execution in all our benchmarks and measured the distance correlation between the time to complete the benchmark and the energy spent in doing so. These executions encompass all the possible configurations, benchmarks and parallelism degree, which amounts to almost 25,000 runs. We used a state of the art distance correlation metric [64] in which two random variables are considered dependent if the distance is 1, and independent if the distance is 0.

In Table 10 we show the computed distance correlation between time and energy. The objective is to assess the extent to which these two variables are related. The results are shown for each benchmark, and averaged across all the configurations and degrees of parallelism. As a result, we obtain an average correlation of 0.81, with an outlier in Vacation high with 0.55 and all others above 0.70. This suggests a relatively strong correlation between the energy and performance achievable by any configuration considered in the study.

The data in Table 11 provides an alternative, interesting perspective from which to analyse the correlation between energy and performance. The experiment whose results are reported in this table was designed to answer the following question: how distant is energy consumption from the optimum in the configuration selected by (G)-Tuner, which, we recall, uses as target metric for the self-tuning process a performance-related metric?

For each benchmark and parallelism degree, we took the best performing configuration in terms of time (configuration T) and energy (configuration E), which are typically different (i.e., we verified that normally $T \neq E$). Then, we compare the relative
Table 10
Distance correlation between performance and energy consumption averaged over the runs with different number of threads for each benchmark. Values closer to 1 show dependence between performance and energy consumption.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Correlation</th>
<th>Benchmark</th>
<th>Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome</td>
<td>0.74</td>
<td>Linked-list low</td>
<td>0.91</td>
</tr>
<tr>
<td>Intruder</td>
<td>0.84</td>
<td>Linked-list high</td>
<td>0.87</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>0.82</td>
<td>Skip-list low</td>
<td>0.94</td>
</tr>
<tr>
<td>K-means high</td>
<td>0.76</td>
<td>Skip-list high</td>
<td>0.81</td>
</tr>
<tr>
<td>K-means low</td>
<td>0.92</td>
<td>Hash-map low</td>
<td>0.98</td>
</tr>
<tr>
<td>SSCA2</td>
<td>0.97</td>
<td>Hash-map high</td>
<td>0.72</td>
</tr>
<tr>
<td>Vacation high</td>
<td>0.55</td>
<td>RBT-low</td>
<td>0.95</td>
</tr>
<tr>
<td>Vacation low</td>
<td>0.74</td>
<td>RBT-high</td>
<td>0.73</td>
</tr>
<tr>
<td>Yada</td>
<td>0.77</td>
<td>Average</td>
<td>0.81</td>
</tr>
</tbody>
</table>

Table 11
Relative energy of the best configurations, aimed for performance, with respect to the best configuration in terms of energy. Values closer to 1 show that optimizing for performance also optimizes for energy consumption. We show the geometric mean across different number of threads for each benchmark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Relative energy</th>
<th>Benchmark</th>
<th>Relative energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genome</td>
<td>0.99</td>
<td>Linked-list low</td>
<td>1.00</td>
</tr>
<tr>
<td>Intruder</td>
<td>1.00</td>
<td>Linked-list high</td>
<td>1.00</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>0.92</td>
<td>Skip-list low</td>
<td>1.00</td>
</tr>
<tr>
<td>K-means high</td>
<td>1.00</td>
<td>Skip-list high</td>
<td>0.98</td>
</tr>
<tr>
<td>K-means low</td>
<td>1.00</td>
<td>Hash-map low</td>
<td>0.99</td>
</tr>
<tr>
<td>SSCA2</td>
<td>1.00</td>
<td>Hash-map high</td>
<td>0.99</td>
</tr>
<tr>
<td>Vacation high</td>
<td>0.99</td>
<td>RBT-low</td>
<td>1.00</td>
</tr>
<tr>
<td>Vacation low</td>
<td>1.00</td>
<td>RBT-high</td>
<td>1.00</td>
</tr>
<tr>
<td>Yada</td>
<td>0.89</td>
<td>Average</td>
<td>0.98</td>
</tr>
</tbody>
</table>

energy obtained with $T$ with respect to that of $E$ (i.e., the optimal one). As such, we obtain the relative loss in terms of energy when optimizing for time compared to that if we optimized for energy.

The results, shown in Table 11, show with an outstanding consistency that the loss is negligible. The average relative energy consumption (i.e., a metric akin to that of speedup that we used earlier) shows a value of 0.98, which means that for the most part we obtain the optimal energy when focusing on performance alone.

This allows us to conclude also that our self-tuning proposals benefit both metrics of time and energy together, while only focusing on the former.

9. Related work

Transactional Memory was initially proposed as an extension to multi-cores’ cache coherence protocols [1]. Due to the inaccessibility of rapid prototyping in such environment, researchers resorted to software implementations (STM) to advance the state of the art [42]. These STMs require instrumenting the code (either manually or compiler-assisted) to invoke the TM software run-time 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 [24,65,66].

Full implementations in hardware (HTM) were always sought as desirable to avoid the overhead of software instrumentation. As a result, several proposals explored the design space in hardware via simulations (e.g. [57]). However, these proposals (and respective results) were typically obtained with different environments than those available in mainstream × 86 processors, for instance by simulating in-order processors.

Recently, HTMs became available in commercial processors delivered by major industry players. Beyond Intel, IBM also provided support for HTM [28], 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.

This current work extends on our preliminary pursuit for self-tuning RTM [67]. We have provided a novel study of the intricacies of Intel RTM, and statically optimized both RTM-SGL and RTM-N0rec with several contributions and state of the art techniques never before evaluated with this Best-Effort HTM. On top of this, we have presented in detail our preliminary TUNER algorithm, as well as the new G-TUNER, with an extended evaluation study and applied our self-tuning proposals to two fallback techniques for RTM. Besides our line of research, we are not aware of any other work that self-tunes RTM (or any similar Best-Effort HTM).
Works that studied the performance of RTM [14,27] obtained promising results, but relied on manual tuning and provided only brief textual insights as to how the decisions to configure it were taken. Similar studies were performed for IBM processors, also lacking workload-oblivious and application-independent mechanisms for tuning the HTM usage [28,68,69].

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 fall-back 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 RTM in SAP Hana database [27]. However, we also show that our proposal is general enough to tune RTM with a fall-back to an STM.

Another recent work, called Adaptive Lock Elision (ALE) [52] proposed to regulate the choice of when to switch from HTM to software fall-backs by relying on an ad hoc adaptive approach that samples different configurations and interpolates the estimated performance of those not tested. In ALE, it becomes extremely important to perform such inferences correctly, because the authors adapt at the granularity of run-time atomic-blocks, i.e., by identifying not only the static source code site of the atomic block, but also labels that may be associated with them to indicate some context (as explained in Section 7). As such, a fundamental difference between the two approaches is that we explore the whole space of configurations, via reinforcement learning techniques that aim to identify a sweet-spot between exploration and exploitation. In fact, the space of exploration in ALE may be much larger, depending on how many contexts are associated with the atomic blocks, for which reason it may benefit from using techniques such as those proposed here. On the other hand, as mentioned above, the Tuner approaches could also be enhanced with the context-aware adaptation proposed in ALE.

Given the Best-Effort nature of this first generation of HTM in commodity processors, it is desirable to consider an efficient solution for the fall-back path in software. One interesting idea is to use STMs combined with HTM, the so called HybridTMs, so that transactions that are not successful in hardware can execute in software without preventing all concurrency, as is the case of the single global lock. In this scope, some work has obtained promising results with a simulator for HTM support from AMD [26,56]. More recently, the Reduced Hardware Transactions technique has been proposed for Intel RTM [41], but it was only evaluated in an emulated environment. We have taken these optimizations and studied them to understand their effectiveness in statically optimizing RTM-NoRec. Concurrently to this work, the authors of Reduced Hardware Transactions have also applied it to a version of Hybrid NoRec [70], obtaining similar gains to those of our baseline starting point for self-tuning.

Additionally, there have been other proposals for adaptation in TMs in software. AdaptiveLocks [50], VOTM [49], and Dynamic Pessimism [71] 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 AdaptiveLocks, 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 [51]. The main drawback of these kinds 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.

Finally, there is also a large body of work that has focused on adapting orthogonal aspects of TM algorithms, namely: the ideal mapping of locks to shared memory in the implementation [22]; the optimal parallelism degree, in terms of application threads, in both STM [72–74] and HTMs [75]; and searching for the optimal mapping of threads to processor cores, so that they benefit the most from the cache usage and avoid mutual interference with one another [76]. In contrast with these works, we focused on dealing with the emerging challenges posed by best-effort HTMs, such as Intel RTM, i.e., self-tuning the policies that govern the retry logic and the usage of the software fallback-path in best-effort HTMs.

In this context, one interesting open research question is how to integrate the solutions proposed in this article with the various self-tuning approaches published so far in the literature. The key challenge that we envision for pursuing this goal is related to the design of self-tuning algorithms capable of coping with the dimensionality explosion of the resulting configuration space.

10. Conclusions

This paper sheds light on one issue that can have a great impact on the performance of the recent Intel Restricted Transactional Memory (RTM): the interplay with the software fall-back that regulates how to cope with failed hardware transactions. We show that the optimal tuning of the software policy that regulates the re-execution of hardware transaction is strongly workload dependent, and that the relative difference in performance among the various possible configurations can be remarkable (up to 10 ×).

In order to ensure the validity of these findings, we first ensured the optimal tuning of RTM, by carefully optimizing orthogonal aspects, such as the implementation of two fall-back paths (i.e., a single global lock and the NoRec STM).

Motivated by these findings, we presented a novel self-tuning approach that combines reinforcement learning techniques and gradient-descent exploration-based algorithms to self-tune RTM in a workload-oblivious manner. This means that our proposal 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 proposed two designs, called Tuner and G-Tuner, which we integrated with the well known GCC compiler, thus achieving total transparency for the programmer. The two designs obtain similar average results, although our study highlighted interesting trade-offs that make Tuner best fit for RTM with a single global lock fall-back, and G-Tuner for RTM with an STM as the fall-back.
Our extensive evaluation study also shows consistent average gains of 60% over the Best Static alternative suggested by existing study to regulate the RTM's retry policy. Furthermore, the proposed solutions obtain results within 5% of the optimal performance of all possible configurations of RTM within the ranges that we considered. Finally, we concluded also that these approaches benefit equally energy-consumption, as energy efficiency and performance appear to be strongly correlated in the large majority of the workloads encompassed by our benchmarks' test-bed (composed by 9 realistic applications and 4 concurrent data structures).