
The moment of truth: virtues and limitations of commodity Hardware Transactional Memory

Nuno Diegues  
INESC-ID/IST  
nmld@ist.utl.pt

Paolo Romano  
INESC-ID/IST  
romano@inesc-id.pt

Luis Rodrigues  
INESC-ID/IST  
ler@ist.utl.pt

December 2013
Abstract

The multi-core revolution that took place nearly one decade ago has turned parallel programming into a major concern for mainstream software development industry. In this context, Transactional Memory (TM) has emerged as a simpler, attractive alternative to lock-based synchronization, whose complexity and error-proneness are widely recognized. On the other hand, performance of TM has been a matter of controversial debate, as existing software implementations (STMs) need to incur in instrumentation overheads that can hinder their efficiency compared to fine-grained locking schemes. Recently, this landscape has been profoundly changed by the integration of Hardware TM (HTM) in the last generation of Intel processors, which raised a number of questions on the future of TM, such as: will the availability of HTM support turn TM into a mainstream paradigm? What role will STM play now that HTM is available in commodity architectures?

We seek answers to these questions by conducting the largest study on TM to date, comparing different locking techniques, hardware and software TMs, as well as different combinations of these mechanisms, from the dual perspective of performance and power consumption. Our study sheds a mix of light and shadows: on one hand, we identify workloads in which HTM clearly outperforms existing synchronization mechanism; on the other hand, current HTM implementations suffer of severe restrictions, which limit significantly the scope in which these can be more effective that state of the art STM solutions. We also highlight the urge for designing more effective approaches for using HTM and STM in synergy, as existing hybrid solutions suffer from excessive overheads that nullify their potential benefits.

Keywords: Multi-core, Synchronization Primitives, Transactional Memory, Locking, Evaluation study
1. Introduction

The advent of multi-core architectures has brought concurrent programming to the forefront of software development. For many years, locking has represented the *de-facto* standard approach to synchronization in concurrent applications. However, the inherent complexity and error-proneness of fine-grained locking schemes [27] has motivated intense research on alternative methodologies aimed at making parallel programming accessible to the mass of software developers.

Transactional Memory (TM) [22] is one of the most prominent proposals in this sense. In the TM paradigm, programmers are required only to identify which code blocks should run atomically, and not how concurrent accesses to shared state should be synchronized to enforce isolation (as with locks). The TM is then responsible for guaranteeing correctness, by aborting transactions that would generate unsafe histories.

Over the last decade a large body of TM research focused on software-based implementations (STM) [13, 14]. Unlike hardware-based implementations, however, STM needs to instrument reads and writes, to trace conflicts at run-time between concurrent transactions. These instrumentation costs can, in certain scenarios, introduce large overheads and hinder performance with respect to conventional fine-grained locking schemes [3]. HTM support is thus desirable, but its absence from commercial CPUs caused most research to be evaluated solely on simulators [20] (the only notorious exception being Sun’s Rock processor [12], which, however, was never commercialized). Recently, the maturing of TM research led to a breakthrough that changed drastically this scenario: over the last year, two major market players, IBM and Intel, have introduced HTM support in their latest generation of processors [32, 31, 3], targeting, respectively, HPC and commodity systems. This represents a significant milestone for TM, mainly due to the predictable widespread availability of Intel Haswell processors, which bring HTM support to millions of systems ranging from high-end servers to common laptops.

The advent of HTM in mainstream architectures raises a number of questions concerning the future of TM and concurrent programming: how competitive are available HTMs when compared with state of the art STMs? Will the performance achievable via HTM be sufficiently alluring to turn TM into a mainstream programming paradigm? What role will STM play now that HTMs are so widely available? How limiting are the architectural restrictions of existing HTM designs?

In this paper we seek an answer to these questions by conducting the largest study on TM-based synchronization to date. We compare, from the twofold perspective of performance and energy-efficiency, a range of synchronization mechanisms: 6 lock based approaches with different granularities; 4 state of the art STM implementations; Intel TSX’s implementation of hardware TM (HTM); and 2 Hybrid TMs (HyTM) that use STM and HTM mechanisms in synergy. We study highly heterogeneous applications, encompassing i) STAMP, a *de-facto* standard suite of benchmarks for TM, ii) Memcached [30],
a popular in-memory object caching system that was recently ported to use TM, and iii) concurrent data structures that are widely used as building blocks of parallel applications (yet, hard to parallelize efficiently). The results of our study allow us to draw two main conclusions:

**Lights and shadows for HTM:** Approaches based on TSX yielded outstanding performance in workloads characterized by small transactions, such as concurrent data structures and Memcached, but only with two of the STAMP benchmarks. TSX is highly dependent on the memory access patterns on the L1 cache, and long running transactions can lead to frequent cache capacity exceptions and spurious aborts. When transaction-intensity is medium, TSX is only the best choice for a limited degree of parallelism, and it is generally better on the energy side than on the performance side. The impact of its hardware limitations are highlighted by several STAMP benchmarks that generate long transactions, and in which TSX is outperformed by both locking and STM solutions. On the other hand, TSX shines as a synchronization primitive for concurrent data structures, for which it is by far the best choice in all workloads considered, with speed-ups up to $3.3 \times$ over the best alternative scheme.

**STM is still competitive:** Our study also shows that STM is quite competitive as an all-around solution across benchmarks, workloads, and parallelism degrees. Although STM was initially proposed as a prototyping alternative to actual hardware implementations of TM, its evolution throughout a decade of intense research has resulted in several highly-optimized mechanisms, achieving performance comparable to that of fine-grained locking. This does not mean that STMs embody a perfect solution; instead, this result highlights the current limitations of HTM support, which make STM still the most robust solution to date.

Further, the results of our study unveil a number of critical issues related with HTM performance and allow for identifying several research problems, whose timely solution could significantly enhance the chances for HTM to turn into a mainstream paradigm for parallel programming:

**HyTMs: a missed opportunity?** The outcome of our study for what concerns the efficiency of HyTMs, when employed in conjunction with Intel's TSX, is rather grim. The mechanisms currently adopted to allow for the simultaneous coexistence of HTM and STM induce high overheads in terms of additional spurious aborts. Our study highlights that these costs make HyTM generally less efficient than solutions based purely on STM or TSX+locking. This motivates further research in the design of architectural support (e.g., support for non-transactional memory accesses from transactions) capable of exploiting the potential synergies of HTM and STM.

**Complexity of HTM tuning.** HTM performance can be significantly affected by the settings of several parameters and mechanisms. Without proper tuning, TSX can yield average throughput losses of 72% and of 89% in energy consumption. Also, the optimal configuration of these parameters can significantly vary depending on the characteristics of the application’s workload. These findings urge for novel approaches capable of removing from the shoulders of programmers the burden of manually tuning HTM parameters, delegating this task to middleware or compiler based solutions.

**Relevance of selective instrumentation.** Both TSX, as well as current GCC’s library for STM, basically trace every memory access performed within a transaction. We show that this can cause significant increases of the transaction footprint’s size, amplifying the instrumentation overheads in STM, and the chances of incurring in capacity exceptions in HTM. These results motivate research on cross-layer mechanisms operating at the compiler and at the hardware level, aimed to achieve selective instrumentation in a way that is both convenient for the programmer and efficiently implementable in hardware.
The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 overviews the synchronization mechanisms considered in the study. Section 4 describes our methodology, after which Section 5 presents preliminary experiments aimed at determining the optimal tuning for TSX. We then present our study in Sections 6–7. In Section 8, we identify several research questions suggested by the findings of our study. Finally, Section 9 concludes the paper.

2. Related Work

Transactional Memory was initially proposed as an extension to multi-processors’ cache coherence protocols (for instance, [22]). Due to the inaccessibility of rapidly prototyping in such environment, researchers resorted to STMs to advance the state of the art [13, 15, 14]. Throughout this decade, other hardware implementations have also been proposed, by relying on simulators to implement the ideas [20].

The concern for both performance and power consumption metrics has been only marginally explored in the scope of TM, and mostly relying on simulation studies that did not target Intel’s architecture. In both [18, 17] the authors resorted to a simulator to assess the behaviour of different HTM implementations; in particular, the latter focused on embedded systems. The approach was also taken by [2], where one STM was studied in a simulator to assess its power consumption. More recently, [19] was the first work to study power consumption and performance in a full fledged (non-simulated) environment. Yet, the authors considered a much narrower set of synchronization alternatives, i.e. only coarse locks (whereas we consider 6 locks and both fine/coarse grained schemes) and only one STM (whereas we consider 4 STMs and 2 HyTMs).

As already mentioned, major industry players have adopted HTM on their latest processors, namely Intel [32] and IBM [31, 3]. IBM processors, however, target high performance computing infra-structures that are not expected to be used in commodity systems. As such, we focus on the former (by Intel), for which our results show aspects and insights that were not highlighted by Intel’s study [32]. Before the recent release of Intel Haswell processors, researchers had already proposed some early theoretical improvements to best-effort HTMs [25, 1]. We integrate these mechanisms in our HTM-based runtime systems, and evaluate them for the first time using an actual HTM implementation.

Traditional techniques of synchronizing via locking have also been thoroughly studied throughout decades. In [16], the authors show that the power consumption of locking primitives can be improved by exploring a trade-off between processor deep sleeping states, frequency downsizing and busy waiting. We highlight a recent work [10], which studied the impact in performance of different lock designs and hardware architectures (without however considering TM).

Our work is clearly related to the body of literature on performance modelling of TM systems, which have relied on methodologies like analytical modelling [11, 21], as well as machine learning [29], which have often been applied to self-tune various TM parameters. Our results clearly indicate the importance of this line of research to make HTM more efficient in practice.

3. Synchronization Mechanisms Considered

In this comparative study we considered the several synchronization mechanisms listed in Table 1.

**Locks** — Decades of research on lock-based synchronization have resulted in a plethora of different implementations, many times trading off subtle changes with great impact in performance. We consider 6 different lock implementations [10] and both coarse and fine-grained locking strategies. Contrarily to the other approaches we used, fine-grained locking requires a per-application lock allocation strategy,
which is a non-generalizable, complex, and error-prone task [27].

**STM** — The key idea of STM is to instrument read and write accesses to trace accesses to shared memory, to detect conflicts among concurrent transactions. Clearly this instrumentation for concurrency control induces overheads that can have a detrimental impact on the efficiency of STM algorithms. Yet, the past decade resulted in many efforts to produce highly optimized STMs. For our study we selected four state of the art STMs, which are representative of different choice in the design space of TM algorithms. These include an STM optimized for validations only at commit-time (TL2 [13]); to maximize performance at low thread counts (NOrec [8]); in high contention scenarios (SwissSTM [14]); and to minimize instrumentation costs (TinySTM [15]).

**HTM** — HTM implements the concurrency control mechanisms in hardware, which avoids the overheads incurred by STM instrumentation. In our study we consider Intel TSX’s implementation of HTM, which is integrated in the family of Intel processors (Haswell) for commodity systems. One fundamental design principle of TSX (and, more in general, of HTM) is its best-effort nature: one cannot depend exclusively on TSX to synchronize accesses to data, since a transaction is not guaranteed to commit even in absence of contention. Briefly, TSX uses L1 cache to buffer transactional writes, and on the cache coherence protocol to detect data conflicts. A plausible reason for a transaction to fail executing exploiting hardware supports is because its data footprint exceeds L1 cache capacity. Further, hardware transactions are also subject to abort due to reasons like page faults and system calls.

As a result, a fallback software-based synchronization mechanism must always be provided in order to ensure progress guarantees in case the transaction cannot be committed using HTM. The software fallback mechanism must clearly co-operate with the hardware ones in order to ensure consistency. As we will see, the mechanisms used to coordinate the execution of the fallback mechanism have a crucial impact on the performance achievable by TSX. The simplest approach, as suggested by Intel’s optimization manual, is to use a single lock to protect atomic sections (we call it TSX-GL). When a hardware transaction aborts, it has the alternative to acquire the global lock instead. To ensure correct interplay with the fallback, hardware transactions must read the lock as free to guarantee correctness; the transactional semantics will guarantee that the transaction is successful when it does not conflict with a fallback transaction.

An obvious extension of this idea is to use multiple, finer-grained locks (TSX-FL). As the TM paradigm is motivated by the need of relieve programers from the complexity of designing fine-grained locking schemes, the usage of fine-grained locking schemes as fallback mechanism for HTM sounds somewhat contradictory. However, this choice allows us to assess to what extent the use of simplistic fallback strategies using a single lock can hinder parallelism. Also, limited fine-grained locking may be automatically crafted with recent techniques based on static analysis [24].

**HyTM** — Another mechanism proposed in the literature is to use STM as fallback for HTM, a.k.a. Hybrid TM (HyTM). HyTM the advantage of allowing concurrent execution of hardware transactions...
and software ones the fallback path, spring from the complexity of identifying fine-grained locks. However, STMs typically manipulate some metadata associated with the memory addresses modified transactionally, for which reason both software and hardware transactions have to play along to ensure correct executions. In our study we considered two state of the art Hybrid TM proposals [25,7], which are evaluated for the first time on a commodity HTM (such as TSX). These HyTMs embed the idea of reduced hardware transactions: normally transactions execute mainly in hardware, with a pure software fallback; the idea of these HyTMs is to have an intermediary mode where the software fallback still relies partially on hardware speculations to boost performance, namely during the commit in the fallback.

4. Methodology and Testbed

We consider in our study several parallel applications using atomic blocks for synchronization. First we use the STAMP suite, a popular set of benchmarks for TM [4], encompassing 8 applications representative of various domains that generating highly heterogeneous workload domains. We excluded the Bayes application given its non-deterministic executions, and used the standard parameters for each application.

We also consider a red-black tree and a hashmap, as examples of concurrent data structures, which represent important building blocks of parallel applications and that have two interesting characteristics: they are very hard to parallelize efficiently using locking schemes, and are challenging for STMs given that they generate extremely short transactions that suffer from the relatively large instrumentation overheads. Finally, we also used a recently crafted transactional version of the popular Memcached [30], an in-memory web cache used to help scale web page servers (widely used for instance at Facebook [26]).

Each experiment is the average of 20 executions to derive statistically meaningful results. We often show speedup results, which are relative to the performance of sequential, non-instrumented executions, unless stated otherwise. Power consumption metrics were collected using Intel RAPL [9].

Our target machine is equipped with an Intel Haswell Xeon E3-1275 3.5Ghz processor with 32GB RAM. This choice fulfils our requirement of using a processor with TSX, which is limited for now to 8 virtual cores (4 physical, with hyper-threading). We always pin threads to physical cores in a round-robin fashion; for instance, 4 threads will be allocated uniformly, one per core. As a result, hyper-threading is only used when 5 or more threads are used. We used GCC 4.8.1 with all compiler optimizations enabled and Ubuntu 12.04. All our synchronization approaches and benchmarks are organized in a framework where backends can be pluggable for evaluation. The software will be open-sourced, but we omit references to the source code due to double-blind review constraints.

5. Tuning TSX Fallback Path

Before comparing the considered synchronization mechanisms, we conduct a set of preliminary experiments evaluating several alternative configurations of the coupling between TSX and its fallback mechanism, which, as we will see, can have significant impact on TSX’s efficiency. The settings identified thanks to this preliminary study will be adopted in the remainder of the paper, to ensure that the comparison is performed using an appropriately tuned HTM system.

We begin in Section 5.1 by comparing the performance and energy efficiency of 6 locks in the fallback path of TSX. This shall allow us to narrow down the multitude of combinations of TSX and lock implementations that we will assess in our study. Next, in Section 5.2 we optimize TSX-GL with a recently proposed technique [5,6] aimed at reducing spurious hardware aborts. This shall provide some
Table 2. Overhead (%) of each lock (as fallback of TSX) with respect to the optimal choice in each execution.

<table>
<thead>
<tr>
<th>Lock</th>
<th>Performance Overhead (%)</th>
<th>Rank</th>
<th>Power Overhead (%)</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ticket</td>
<td>1.0</td>
<td>1.75</td>
<td>1.1</td>
<td>1.75</td>
</tr>
<tr>
<td>MCS</td>
<td>2.4</td>
<td>2.62</td>
<td>1.2</td>
<td>2.25</td>
</tr>
<tr>
<td>CLH</td>
<td>2.9</td>
<td>3.62</td>
<td>2.4</td>
<td>3.38</td>
</tr>
<tr>
<td>RW</td>
<td>14.2</td>
<td>4.89</td>
<td>17.4</td>
<td>3.88</td>
</tr>
<tr>
<td>TTAS</td>
<td>15.2</td>
<td>5.00</td>
<td>17.4</td>
<td>4.88</td>
</tr>
<tr>
<td>Spin</td>
<td>16.4</td>
<td>5.00</td>
<td>17.5</td>
<td>4.88</td>
</tr>
</tbody>
</table>

insights about its actual practical effectiveness, as it was never evaluated before. Lastly, we investigate when it is best to give up on hardware and trigger the fallback path, in Section 5.3.

5.1. The Impact of Locks on the Fallback

The simpler way to use TSX is by relying on very coarse-grained locks on the fallback, or simply a single one. We considered the six lock listed in Table 1 to be used in the fallback. These implementations are representative of different design choices, and our goal is to understand if there is some implementation that consistently performs good enough across all considered parallelism degrees and benchmarks.

Table 2 shows a comparison of TSX’s behaviour according to the lock implementation that is used in the fallback path. We show the average overhead with respect to the best performing lock in each experiment, considering both time to complete the benchmark as well as power consumed. The reported overhead is the average across all STAMP benchmarks and thread counts (1 to 8). Using this metric, we can see that the Ticket, MCS and CLH locks perform best.

We additionally ranked the locks in every experiment, according to performance and power consumption, and then averaged every rank obtained by each lock. This shows that no lock implementation is always the best or worse. However, we can see that the Ticket lock is consistently ranked higher, for which reason we shall rely on it from now on whenever we require locking (both standalone, or in the fallback of TSX).

5.2. Improving the Single-lock Fallback

The recommended fallback mechanism for TSX relies on a global lock (as explained in Section 3.). In this section we evaluate for the first time on TSX the efficiency of an alternative technique proposed to reduce the situations under which best-effort HTMs have to follow this pessimistic approach [1].

The idea is that it may not be safe for transactions to execute speculatively in hardware an atomic block if at the same time some transaction is executing pessimistically by acquiring the global lock. A pessimistic execution cannot restart so its accesses must be consistent when faced with concurrency. For this reason, TSX’s usage guide points out that the lock has to be read as being free during the course of a hardware transaction. Then any transaction that activates the fallback path has to first acquire the lock, and cause every hardware transaction to abort. This can cause a chain effect, also known as lemming effect [12], where the aborted hardware transactions also try to acquire the lock, preventing hardware speculation from ever resuming. In [1], the authors use an auxiliary lock to prevent the lemming effect. The idea is to guard the global lock acquisition by another lock. Aborted hardware transactions have to acquire this auxiliary lock before restarting speculation, which effectively serializes them. However, this auxiliary lock is not added to the read-set of hardware
transactions, which avoids aborting concurrent hardware transactions. If this procedure is attempted some times before actually giving up and acquiring the global lock, then the chain reaction effect can be avoided, as the auxiliary lock serves as a fallback manager preventing hardware aborts from continuously acquiring the fallback lock and preventing hardware speculations.

In Table 3 we compare TSX using a single-lock (TSX-GL) and the auxiliary lock. We report the average across either benchmarks or threads, and show the relative improvement in time, energy, and their combination with the Energy Delay Product (EDP) of the optimized approach normalized with respect to TSX-GL. Naturally, we can see that there is no difference with 1 thread because there is no concurrency and hence no problem resuming speculative execution. But beyond that, and in particular at larger concurrency levels, this technique helps consistently to improve the EDP. Some benchmarks do not show any difference because there are very little aborts (SSCA2) or TSX is not able to execute speculatively most of the time (Labyrinth). Yada’s workload is conflict-intensive, for which reason the non-optimized approach is slightly better due to its inherent pessimism in following the fallback path — that pays off due to the high conflict probability limiting the effectiveness of optimistic transactions.

5.3. Retry Policy for the Fallback

Given that TSX must always have a fallback due to its best-effort nature, an important decision is when to trigger that path. Upon a transaction abort, TSX provides an error code that informs about the reason of the abort. An abort due to a capacity exception is typically a good reason to trigger the fallback path. However, hardware transactions may abort for various micro-architectural conditions that are less deterministically prone to happen upon transaction re-execution, and even capacity exceptions may not always be deterministic. Also, there is the case that data contention exists and transactions abort due to conflicts. In these situations one may aggressively trigger the fallback, or opt to insist on speculating using HTM.

As we will shall discuss in more detail in Section 8., the optimal choice of the retry policy can vary significantly across workloads and degrees of parallelism. As it is impractical to assume that the retry policy is ad-hoc tuned by programmers for each and single workload/application, we use in our study the configuration that performs best on average. Recent works with TSX have reported that retrying up to 5 times is the best all-around figure [32, 23]. We have confirmed that with TSX-GL on our testbed, by considering that transient failures (such as conflicts) do not decrease the retries left. For the HyTMs, 4 times was the best number of retries on average.
6. STAMP Benchmark Suite

In this Section we rely on the STAMP benchmark suite to assess the efficiency of all the synchronization mechanisms listed in Section 3., namely HTM, STMs, HyTMs, and locking. As already mentioned, we shall always include the TSX optimizations discussed in the previous section.

We start by summarizing our results in Table 4. There, we list the STAMP benchmarks sorted by two important characteristics of their workloads: the percentage of time spent inside transactions (i.e., atomic blocks), and the contention level. We then identify the synchronization mechanism that takes the least time to complete and which one consumes the least power, given the averaged results across threads.

This summarized perspective allows to highlight an interesting fact. It is possible to distinguish three categories in which TSX behaves differently, according to the transaction’s characteristics. Kmeans and SSCA2 represent workloads with small transactions, medium frequency and low contention; here, TSX-GL performs consistently better than the alternatives across all threads. Then, Intruder and Vacation exhibit medium profiles for what concerns the time spent in transactions and contention; in these cases, TSX-GL results in the best performing solution using up to 4, resp. 2, threads, and the most energy efficient up to 5, resp. 4, threads. Finally, the other benchmarks spend almost all the time in transactions, encompassing both low and high contention scenarios. In these settings, TinySTM emerge as the most robust solution, both from the perspective of energy and performance.

This analysis allows to draw a set of guidelines to select which synchronization to use, at least when considering applications having analogous characteristics to those included in the STAMP suite. A simple approach with TSX-GL is desirable when transactions are small, generate low/medium contention, and the workload is not all the time under transactions. When contention increases, or the frequency of transactions is high, TSX-GL is competitive up to a medium degree of parallelism. In the remaining cases, STM is normally the best choice, even when compared with fine-grained locking. The considered HyTMs perform poorly compared to the alternatives, never clearly outperforming the competing schemes in any benchmark. In the following Sections 6.1.-6.2. we present in more details our experiments in STAMP. We then consider additional benchmarks and fine-grained locks in Section 7.

6.1. Performance Study

In Fig. 1 we show, for each benchmark and while varying the parallelism level, the speedup of all the considered synchronization schemes (with the exception of schemes based on fine-grained locking, which shall be presented in Section 7.) with respect to a sequential, non-instrumented execution, and the power consumption during the execution (in Kilo Joules). This allows us to discuss in detail the differences between the synchronization mechanisms in different workloads.
**Kmeans:** This benchmark yields the biggest gap in performance between a TSX variant and STMs. Namely, TSX-GL reaches $3.5\times$ speedup over a sequential execution, beating every other alternative both performance-wise and in terms of energy-efficiency. An interesting trend concerning energy-efficiency is that the power consumption with TSX (and, to some extent, also for all other synchronization schemes but GL) tends to slightly decrease as the parallelism level grows, which is a symptom of efficient utilization of the available architecture resources achievable using TM-based solutions. If we consider TSX-TL2 and TSX-NOrec, they are still competitive and better than the corresponding STMs, but they are far from TSX-GL in both metrics. It is worth noticing that the small and rare atomic blocks of this benchmark allow the GL approach to scale up to 3 threads. This explains the considerable success of TSX-GL in this benchmark, as a transaction that resorts to the GL is still able to run concurrently with other threads that are not under an atomic block at that time.

**SSCA2:** This benchmark shows a similar trend between TSX variants, but with the significant difference that all STM approaches scale better as the degree of parallelism increases. Here, TSX-GL is only slightly better than the best STM, and this is consistent across all the thread counts. Also interestingly, TSX-TL2 improves little and fares rather bad on the energy side. This, however, is not the case for TL2 or TSX on their own, and as such is an artefact of the hybrid implementation integration. Finally, the reduced time within atomic blocks still allows the GL approach to scale up to 2 threads, which justifies the advantage of TSX-GL. However, this effect is smaller than in Kmeans, which also matches the fact that TSX-GL achieves less improvements over other approaches.

**Intruder:** Here TSX-NOrec (and TSX-GL to some extent) are competitive and even better (until 5 threads) than the best STMs (except for TL2). Since TL2 performs so poorly in this benchmark, this also drags TSX-TL2 a behind in both metrics. Interestingly, both TL2 and TSX-TL2 improve slightly performance with more threads, but TL2 consumes more power whereas TSX-TL2 slightly decreases the power consumed.

**Vacation:** Once again we can see that TSX-TL2’s performance is quite disappointing, as indeed TL2 itself performs poorly in this scenario. As we shall see throughout this study, TL2 is by far the worst STM among those considered, which is a result of having a similar algorithmic and synchronization complexity to that of SwissTM and TinySTM, while detecting conflicts only at commit-time. This fact results in TL2 doing useless work more often, whereas SwissTM and TinySTM are able to restart the speculation faster when reacting to conflicts. On the other hand, NOrec is much simpler, both in algorithmic as well as synchronization terms, reducing its instrumentation overheads and maximizing its performance at low thread counts. With regard to the other approaches, TSX-GL and TSX-NOrec are competitive with STMs until 4 threads. At higher parallelism degrees, their performance degrades due to contention on L1 caches caused by hyper-threading. The perspective on power consumption follows up closely on these results. It is interesting to note that TSX-GL performs worse than TSX-NOrec at 8 threads, but they consume approximately the same power. This is a result of the power savings that are achievable by the deep sleep state induced during the single global lock acquisition in TSX-GL.

**Genome:** In these three last benchmarks we have either transaction-heavy or high-contention workloads, characterized by large transaction foot-prints. These conditions are clearly a much more favourable playground for STMs. In this case, we can see a clear (and consistent across benchmarks) distinction between TL2 and NOrec, as these two lag behind in both metrics particularly at higher thread counts. Interestingly, we can see that TSX-TL2 performs best among the TSX variants at a higher concurrency degree, which is a singularity among all benchmarks. This benchmark also shows a clear trend when the 5th thread is used: all approaches stabilize (or even decrease) performance at that point, due to the usage of hyper-threading. Interestingly, this effect is not so noticeable on the
Figure 1. Speedup (relative to non-instrumented sequential execution) and Energy Consumption (in Kilo Joules) while varying the number of threads (horizontal axis) in STAMP benchmarks.

energy side, as STM approaches are still able to reduce the power consumed as parallelism increases. This highlights an interesting trade-off of hyper-threading: it allows sub-linear speed-ups only, but
Table 5. Transactional abort rate (%). For STM we show the lowest and highest values obtained (across all considered STMs).

<table>
<thead>
<tr>
<th>benchmark</th>
<th>kmeans</th>
<th>ssc2</th>
<th>intruder</th>
<th>vacation</th>
<th>genome</th>
<th>yada</th>
<th>labyrinth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STM</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 0</td>
</tr>
<tr>
<td>TSX-GL</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>49</td>
<td>11</td>
<td>46</td>
<td>95</td>
</tr>
<tr>
<td>TSX-TL2</td>
<td>0</td>
<td>0</td>
<td>36</td>
<td>94</td>
<td>35</td>
<td>19</td>
<td>53</td>
</tr>
<tr>
<td>TSX-NOrec</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>40</td>
<td>6</td>
<td>19</td>
<td>53</td>
</tr>
<tr>
<td>4 threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STM</td>
<td>10 - 34</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 6</td>
<td>1 - 51</td>
<td>5 - 58</td>
<td>4 - 13</td>
</tr>
<tr>
<td>TSX-GL</td>
<td>26</td>
<td>0</td>
<td>22</td>
<td>69</td>
<td>31</td>
<td>48</td>
<td>100</td>
</tr>
<tr>
<td>TSX-TL2</td>
<td>50</td>
<td>74</td>
<td>74</td>
<td>100</td>
<td>45</td>
<td>84</td>
<td>60</td>
</tr>
<tr>
<td>TSX-NOrec</td>
<td>31</td>
<td>46</td>
<td>29</td>
<td>66</td>
<td>17</td>
<td>31</td>
<td>55</td>
</tr>
<tr>
<td>8 threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STM</td>
<td>25 - 54</td>
<td>0 - 0</td>
<td>3 - 57</td>
<td>0 - 10</td>
<td>0 - 1</td>
<td>7 - 65</td>
<td>8 - 23</td>
</tr>
<tr>
<td>TSX-GL</td>
<td>42</td>
<td>1</td>
<td>33</td>
<td>72</td>
<td>48</td>
<td>47</td>
<td>100</td>
</tr>
<tr>
<td>TSX-TL2</td>
<td>60</td>
<td>99</td>
<td>92</td>
<td>100</td>
<td>53</td>
<td>92</td>
<td>69</td>
</tr>
<tr>
<td>TSX-NOrec</td>
<td>44</td>
<td>88</td>
<td>62</td>
<td>99</td>
<td>69</td>
<td>39</td>
<td>60</td>
</tr>
</tbody>
</table>


it also consumes little additional power. This fact is more favourable to STMs, as TSX approaches generate many more transactional aborts when hyper-threading is used, due the higher contention on the L1 cache.

**Yada:** This benchmark shows one scenario where TSX-GL performs poorly, with slowdowns above 3 threads. HyTMs follow closely their fallback STMs’ performance, as TSX is not able to succeed. This is also a case where TinySTM and SwissTM perform better than the other two STMs. This benchmark presents no surprises in relation with energy-efficiency, whose trends are highly correlated with the performance.

**Labyrinth:** We can see STMs performing best and very alike each other. TSX-GL does not improve with thread count, simply because most transactions exceed the hardware cache capacity and, as such, eventually follow the fallback path which is a sequential bottleneck given the GL. For this reason, TSX-TL2 and TSX-NOrec obtain some improvements, exactly because the fallback allows for concurrency, contrarily to the global lock on TSX-GL. This scenario highlights, however, that HyTMs are capped by either TSX or the fallback STM — as such, it is dubious whether they are practical, and if it would be preferable to adaptively employ the most promising technique (TSX-GL or an STM) based on the workload.

### 6.2. Insights on TM Efficiency

In this Section we shed some additional light on the factors dictating the trends observed in the experiments. To this end, in Table 5 we report the average abort rate across benchmarks and threads for each speculative mechanism. This represents the percentage of speculations that do not complete. Since there are four STMs under evaluation, we show the minimum and maximum abort rates among them — typically the smallest abort rate belongs to TinySTM/SwissTM, whereas TL2 yields the maximum abort rate.

Once again, we structure the table considering the different categories of workloads. As we move to the right (more contended or transaction-intensive workloads) and down (higher degree of parallelism), TSX approaches increase their abort rates, which causes the loss of efficiency shown in the previous section. These results highlight that TSX has non-negligible aborts in many occasions where STMs abort very little.

In Fig. 2 we consider four different benchmarks, representative of scenarios that allow to derive insights on the efficiency of the considered TSX variants. In those plots we present a breakdown of the reasons
motivating transactional aborts, for each TSX mechanism. We identified aborts caused by exceeding the capacity of L1 cache; micro-architectural instructions or states forbidden by TSX, such as some system calls; data contention resulting in conflicts; and interaction between TSX and the fallback paths, such as checking if the GL is free in TSX-GL or more complex logic in the case of HyTMs.

Kmeans shows the expected breakdown, without aborts in single-threaded executions, and increasing aborts motivated mainly by data conflicts as concurrency increases. It is worth mentioning that Kmeans is the benchmark with the least average aborts for TSX variants. Half of the aborts are motivated by data conflicts from the workload, whereas the rest is motivated by a non-negligible percentage of aborts due to architectural instructions. This is something intrinsic to TSX, which is common throughout different benchmarks. The fact that these aborts occur less often in this benchmark allows TSX to obtain the most favourable results among all benchmarks.

In Yada and Labyrinth, instead, the workloads are much more transaction-intensive with non-negligible conflict rates. On top of this, the capacity of L1 caches is often exceeded by the hardware transactions (this is particularly visible in Labyrinth, where this phenomena dominates the aborts). This explains why the TSX variants followed up closely the performance of their fallbacks (with some constant overhead). HyTMs have a reduced abort rate because the fallback’s software transactions are also taken into account in these statistics, on top of the hardware transactions — since software transactions have little aborts due to the uncontended workload, they amortize the overall abort rate for the hardware transactions. In TSX-GL, instead, the fallback executes non-speculatively and serially due to the global lock, so we only count statistics for the hardware transactions there.

Finally, SSCA2 shows a completely different scenario, in which TSX-GL generates almost no aborts (in line with STMs’ behaviour), whereas the HyTMs have enormous abort rates, dominated by the interaction with the fallback path. Note that this problem does not occur with 1 thread, as there is no concurrency that allows transactions to simultaneously execute in hardware and in the fallback.

This motivates to better understand the usage of the fallback path in the HyTMs. Table 6 shows the
Table 6. Rate (%) of triggering the fallback on HyTMs and of executing it in fast mode. Intervals of values are shown, ranging from 1 (lower) to 8 threads (upper bound).

<table>
<thead>
<tr>
<th></th>
<th>TL2</th>
<th></th>
<th>NOrec</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fallback</td>
<td>Fast</td>
<td>fallback</td>
<td>Fast</td>
</tr>
<tr>
<td>kmeans</td>
<td>&lt; 1 - 77</td>
<td>92 - 32</td>
<td>&lt; 1 - 78</td>
<td>100 - 23</td>
</tr>
<tr>
<td>scca2</td>
<td>&lt; 1 - 99</td>
<td>91 - 2</td>
<td>&lt; 1 - 86</td>
<td>95 - 14</td>
</tr>
<tr>
<td>intruder</td>
<td>33 - 88</td>
<td>99 - 39</td>
<td>3 - 55</td>
<td>100 - 52</td>
</tr>
<tr>
<td>vacation</td>
<td>94 - 100</td>
<td>43 - 3</td>
<td>38 - 99</td>
<td>100 - 89</td>
</tr>
<tr>
<td>genome</td>
<td>50 - 100</td>
<td>97 - 71</td>
<td>6 - 67</td>
<td>100 - 94</td>
</tr>
<tr>
<td>yada</td>
<td>18 - 78</td>
<td>50 - 34</td>
<td>17 - 32</td>
<td>99 - 82</td>
</tr>
<tr>
<td>labyrinth</td>
<td>58 - 100</td>
<td>14 - 2</td>
<td>54 - 98</td>
<td>10 - 3</td>
</tr>
</tbody>
</table>

percentage of transactions that had to be executed in the fallback path (i.e., not purely in hardware). In addition to that, we also show the percentage of transactions in the fallback path that are able to execute in a fast mode, i.e., one in which the transaction executes in software but the commit is boosted by using a reduced hardware transaction [25] (as explained in Section 3.). For every table cell we show the percentage corresponding to 1 and 8 threads. Overall, the percentages vary linearly from 1 to 8 threads, for which reason we omit the intermediate values.

We start by highlighting in SSCA2 how both HyTMs are able to execute purely in hardware with 1 thread (they trigger the fallback < 1% of the transactions). However, notice how a higher thread count results in executing in the fallback mode almost all the time, which matches the idea conveyed by Fig. 2(d). In particular, for this benchmark, the ability to rely on hardware to speed up the software fallback path is reduced from above 90% to 14% or even less.

These results for HyTMs show that TSX-TL2 triggers the fallback more often, and is able to execute in the fast mode less frequently than TSX-NOrec. This justifies the advantage of TSX-NOrec, which fared better across all benchmarks as shown in Section 6.: notice how TSX-NOrec executes the fallback software transactions in fast mode for most of the time. The reason is that the (much) simpler design of NOrec allows for a (much) easier integration with TSX in a HyTM.

Ideally one may want to also rely on more scalable STMs, like TinySTM or SwissTM, in the fallback of TSX. However, due to the higher complexity of their algorithms, coupling them efficiently with HTM is a challenging task, and, in fact, we are not aware of any proposal in this sense in literature.

Finally, it has been pointed out [7] that, in order to support efficient HyTMs, it is desirable to have hardware support to issue selective non-transactional memory accesses from the scope of transactions. Such a feature is not currently supported in TSX, whereas its inclusion was, e.g., planned in AMD’s HTM proposal [6] (which, however, was never commercialized). Hence, an interesting research direction suggested by this study is to investigate the impact of supporting non-transactional accesses, not only in terms of performance/energy, but also in terms of architectural intrusiveness.

7. Benchmarks Using Fine-grained Locking

Most of the STAMP benchmarks have an irregular nature, which makes it very challenging to derive fine-grained locking schemes. In this section we focus on benchmarks for which it is possible to use (possibly very complex) fine-grained locking approaches. We start, in Section 7.1, by focusing on a subset of three STAMP benchmarks, for which we could craft an ad-hoc fine-grained locking strategy. We then present results for Memcached in Section 7.2 and for two concurrent data structures in Section 7.3.
7.1. Fine-grained Locking in STAMP

As already mentioned, implementing a fine-grained locking strategy is a complex task for most of the STAMP benchmarks. We were however able to devise fine-grained locking strategies for three of the STAMP benchmarks, whose results we report in Fig. 3. Besides results using fine-grained locking (FL), we show also data for TSX-FL, which combines hardware transactions with a fallback path that relies on FL. Naturally, the combination of both schemes in TSX-FL requires hardware transactions to read all necessary locks as being free. We then compare these two approaches with TSX-GL and TinySTM, which were the best mechanisms in our previous experiments, and remove the others to improve the readability of the plots.

TSX-FL presents one advantage over TSX-GL, in that the fallback path allows for threads to proceed in parallel if they require different locks (which is highly likely if there is little data contention). However, this has the drawback that more locks have to be checked (during speculative executions) or acquired (during the fallback executions). Hence, there is a clear trade-off that is subtle and difficult to manage.

Recall that Kmeans and SSCA2 were the two benchmarks with workload characteristics more amenable to TSX-GL. This is justified by the low frequency of activation of the fallback path. As such, TSX-GL incurs minimal overhead thanks to the hardware speculation and to the avoidance of any software-based instrumentation. Therefore, it is not a surprise that fine-grained locking is of no advantage in this scenario: each lock acquisition represents a synchronization point, whereas for TSX-GL there exists only explicit synchronization at the hardware level when a transaction attempts to commit. Note, however, that FL is consistently better than the best STM (TinySTM). This fact is even more relevant from the energy perspective, where the gap between FL and TinySTM is larger. Since the TSX fallback is not triggered often, then TSX-FL goes through the additional verifications over more locks that are useless most of the time (to ensure a correct integration of the fallback with hardware transactions), which explains its lower performance in this kind of workload.

In SSCA2 we see a different behaviour as both TSX variants perform quite similarly. This is explained by the fact that the fine-grained scheme is not very efficient: its locks are relatively coarse, which induces unnecessary serialization. This has the side-effect of making TSX-FL competitive with TSX-GL, because both have a similar effort in checking the locks in the speculative executions to ensure correct integration with the fallback. Notice how the FL scheme still performs better than GL, which is a consequence of the higher degrees of parallelism achievable by reducing lock granularity. This confirms an expectable trade-off concerning lock granularity: the more fine-grained, the best the fallback performs; however this can have an impact on the performance of the speculative executions as we saw for Kmeans.

Finally, Intruder spends a large fraction of time within atomic blocks. As already discussed, this workload is more advantageous for STMs than for TSX. It is not surprising to see that TSX-GL is no longer the most competitive choice (although it still fares best until 3 threads). The interesting fact is that this kind of workload is more beneficial towards FL. With more threads, TinySTM degrades its scalability, and is surpassed by FL. From an energy perspective, it is even clearer that FL is the best choice comparing to TinySTM, as it is almost always consuming less energy. TSX-FL suffers from the overheads of checking additional locks, until 3 threads, for which reason it is not as good as TSX-GL. However, at that point TSX triggers the fallback more often, which justifies the use of fine locks and allows TSX-FL to perform substantially better than TSX-GL. On the energy side, TSX-FL is much closer to TinySTM, also following the trend of STMs spending more energy to obtain a given performance level.
7.2. Memcached

Memcached is a popular distributed object caching system [26]. We adapted a recent port to TM [30] and use the original Memcached as the basis for FL. We used the memslap tool [30], configuring the workload with 95% gets and 5% puts, 8 threads and a concurrency degree of 256.

In Memcached it is not really possible to measure, for reference purposes, the performance of a sequential execution, because there is always concurrency due to the existence of a pool of maintenance threads. Hence, we present the peak throughput obtained using the maximum number of available hardware threads (see Table 3(h)). The results show that FL has the best performance, but TSX-GL is only 7% behind. This is a significant achievement as the effort to devise such an optimized fine-grained locking scheme is considerably higher than using TSX-GL. Also, since FL is already quite optimized, it is expectable that TSX-FL is not able to extract any further parallelism. Interestingly, with this application, TinySTM is not really competitive because the instrumentation overheads are amplified by the short and uncontented transactions.
7.3. Concurrent Data Structures

We now consider two concurrent data structures, namely a red-black tree and a hashmap, which represent particularly relevant use cases for TM given the complexity of designing efficient fine-grained locking strategies for these scenarios.

Fig. 4 shows two different scenarios: we consider a small hashmap (512 buckets) with only 10% transactions performing writes (the rest are lookup operations), and a large red-black tree (1 million items) with 90% transactions performing updates. In the former case, TSX-GL achieves perfect linear scalability, which is a consequence of its negligible overheads and of the very reduced abort rate. With larger transactions, the gains achievable by TSX tend to diminish, although it still remains a very competitive solution.

Table 4(e) shows a spectrum of workloads in red-black tree, by considering the normalized EDP of TSX-GL against the best alternative in each experiment. For this, we vary the size of the tree and the percentage of write transactions. The trend is clear in this table: TSX behaves best with light workloads, and loses advantage when transactions become larger or write-intensive. This confirms the results of the workload analysis that we performed for the STAMP suite, given that, also in this case, TSX shines most when atomic blocks have little duration and the workload is not fully transactional.

8. Research Directions Suggested by our Study

We now identify some relevant research directions that emerged from the analysis of our experimental study:
Table 7. Improvement (%) when configuring TSX-GL for each application over the single configuration used in our study.

<table>
<thead>
<tr>
<th>Speedup %</th>
<th>kmeans</th>
<th>scca2</th>
<th>intruder</th>
<th>vacation</th>
<th>genome</th>
<th>yoda</th>
<th>labyrinth</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 threads</td>
<td>12</td>
<td>7</td>
<td>20</td>
<td>36</td>
<td>12</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td>8 threads</td>
<td>5</td>
<td>8</td>
<td>80</td>
<td>21</td>
<td>2</td>
<td>55</td>
<td>39</td>
</tr>
</tbody>
</table>

- The overall performance of the HyTM solutions we have tested is disappointing. These findings contradict the simulation results published in several previous works, e.g. [25]. Our analysis suggest that the root cause of the problem is related to the inefficiency of the mechanisms used to couple hardware and software transactions, which is generating a large number of spurious aborts. However, further research is due in order to understand what can be done to address such a problem. An interesting research question, in this sense, is whether the availability of support for enabling non-transactional memory accesses while executing hardware-assisted atomic blocks could indeed allow for more efficient interplay between HTM and STM (which has been assumed by other works in the area of HyTM, e.g. [28]). A related research question is how to support such a feature while minimizing the disruptiveness of the changes required at the hardware level — an aspect that cannot be overlooked given the complexity of modern processor architectures.

- As mentioned in Section 5.3., the performance of TSX is significantly affected by the retry policy (e.g., the settings of the number of retries upon abort, and the choice of how to react to capacity aborts). While in our study we used the configuration that performed best on average, as shown in Table 7, significant speedups (up to 80%) with regard to the configuration used in our study can be achieved by ad-hoc tuning the retry policy for the specific workload — even more could be achieved by considering the specific concurrency degree as well. Unfortunately, this is a tedious and error prone task that is not desirable to delegate to programmers. Hence, these findings highlight the relevance of devising solutions to adaptively tune these parameters in an automated manner. The key challenge is how to do it with minimal overhead, given that the cost imposed by self-tuning approaches targeting STMs (based on complex machine-learning [29] or analytical models [11]) is going to be strongly amplified in HTM settings (given that HTM avoids the instrumentation costs of STM).

- Our study used (selective) manual instrumentation when considering software based TMs (STMs and HyTMs), where only the relevant subset of memory locations accessed in atomic blocks have been traced. As an alternative, one could rely on the compiler to automatically instrument atomic blocks with calls to the TM runtime. The plots in Fig. 5, which were obtained using the C++ TM extension integrated in GCC 4.8.2, show that non-selective instrumentations can impact in performance by approximately 20% when using TinySTM. This is a direct consequence of the increase of the transaction footprint’s size (up to 3x larger with SSCA2) caused by the “blind” instrumentation performed by GCC. Not only these results unveil the possibility of optimizations in existing compiler’s support for STM, but also provide an additional compelling motivation to incorporate supports for selective instrumentation in HTM. Indeed, we have shown that capacity exceptions are one of the key sources of aborts with HTM. Hence, techniques capable of achieving noticeable reductions of the transactions’ footprint are expected to strongly benefit HTM’s performance. These considerations open interesting research avenues investigating cross-layer mechanisms operating at the compiler and architectural level, and aimed at supporting selective instrumentation in a way that is both convenient for the programmer (i.e., possibly fully transparent). However, all of this should be achieved while minimizing the cost/intrusiveness of its hardware implementation.
9. Conclusions

This paper analysed extensively the performance and energy efficiency of several state of the art of Transactional Memory (TM) systems. We compared different TM solutions (software, hardware and combinations thereof) among each other and against lock based systems. Our study demonstrates that the recent HTM implementation by Intel can strongly outperform any other considered synchronization alternative in a set of relevant workloads. On the other hand, it also identified some critical limitations of TSX, and highlighted the robustness of state of the art STMs. These software implementations achieve performance competitive with fine-grained locking schemes, and outperform HTM in workloads encompassing long and contention-prone transactions.

Furthermore we have found the performance of Hybrid TM, when used in combination with TSX, to be disappointing; we determined that the root cause of this surprising result lies in the inefficiency of the mechanisms used to couple software and hardware transactions. Finally, our study allowed to identify a set of compelling research questions, which should be timely addressed to increase the chances of turning HTM into a mainstream paradigm for parallel programming.

References


![Figure 5. Impact of GCC’s non-selective instrumentation.](image)


