Virtues and Limitations of Commodity Hardware Transactional Memory

Nuno Diegues, Paolo Romano, Luís Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{nmld, paolo.romano, ler}@tecnico.ulisboa.pt

ABSTRACT

Over the last years Transactional Memory (TM) gained growing popularity as a simpler, attractive alternative to classic lock-based synchronization schemes. Recently, the TM landscape has been profoundly changed by the integration of Hardware TM (HTM) in Intel commodity processors, raising a number of questions on the future of TM.

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 currently available commodity HTM: on one hand, we identify workloads in which HTM clearly outperforms any alternative synchronization mechanism; on the other hand, we show that current HTM implementations suffer of restrictions that narrow the scope in which these can be more effective than state of the art software solutions. Thanks to the results of our study, we identify a number of compelling research problems in the areas of TM design, compilers and self-tuning.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques - Concurrent Programming

Keywords

Empirical Study; Synchronization Techniques; Transactional Memory; Performance; Energy Efficiency

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 [32] has motivated intense research on alternative methodologies aimed at making parallel programming accessible to the mass of software developers.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.
Copyright 2014 ACM 978-1-4503-2809-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2628071.2628080 .

Transactional Memory (TM) [27] is one of the most prominent proposals in this sense. With the TM abstraction, programmers are required only to identify which code blocks should run atomically, and not how concurrent accesses to shared state should be synchronized (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) (e.g. [13, 17]). Unlike hardware implementations, however, STM requires software instrumentation of read and write memory accesses to trace conflicts between concurrent transactions. This instrumentation can, in certain scenarios, introduce large overheads and hinder performance with respect to conventional fine-grained locking [5]. HTM support is thus desirable, but its absence from commodity processors caused most research to be evaluated solely on simulators (e.g. [25]) — the only notorious exception being the Rock processor [12], which was never commercialized. Recently, the maturing of TM research led to a breakthrough that changed drastically this scenario: two major market players, IBM and Intel, introduced HTM support in their latest processors [37, 3, 36], 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 commodity processors 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 of HTM be sufficiently alluring to turn TM into a mainstream programming paradigm? What role will STM play now that HTM is 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 STMs; Intel TSX's implementation of HTM; and 2 Hybrid TMs (HyTM) that use STM and HTM mechanisms in synergy. We study highly heterogeneous applications, encompassing 1) STAMP, a *de-facto* standard suite of benchmarks for TM, 2) Memcached [35], a popular in-memory caching system that was recently ported to use TM, and 3) 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 performance is strongly dependent on the access patterns to 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 considered workloads, with speed-ups up to 3.3× 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, which achieve 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 of 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 concurrent 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 support 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 supports (e.g., non-transactional memory accesses from transactions) capable of exploiting the potential synergies of HyTMs.

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

Relevance of selective instrumentation. Both TSX and GCC library for STM 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 tuning properly 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 [27]. Due to the difficulty of rapid prototyping in hardware environments, researchers resorted to STMs to advance the state of the art [18, 17, 16, 14]. Simultaneously, hardware-based implementations have also been proposed, whose designs were validated using simulators [25].

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 (whose internals are only partially disclosed). In both [21, 20] the authors assess the behaviour of different HTM implementations via simulation (the latter focusing on embedded systems). The approach was also taken by [2], where the power consumption of one STM was studied via simulation. More recently, [22] studied both power consumption and performance in a non-simulated environment. Yet, this work considered a restricted set of synchronization alternatives focusing mainly on one STM.

As already mentioned, both Intel [37] and IBM [3, 36] have integrated HTM in their processors. IBM processors target high performance computing infra-structures that are not expected to be used in commodity systems. In this work, we focus on the former (by Intel), for which our results show aspects and insights that were not highlighted by Intel's paper [37]. Before the recent release of Intel Haswell processors, researchers had already proposed some theoretical improvements to best-effort HTMs [1, 30]. We integrate these mechanisms in our HTM-based runtimes, and evaluate them for the first time using an actual HTM implementation.

Traditional lock-based synchronization techniques have been thoroughly studied throughout decades. In [19], 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 also related to the body of literature on performance modeling of TMs, which have relied on methodologies such as analytical modeling [11, 26], as well as machine learning [34]. These proposals were applied to self-tune various TM parameters, and our results clearly indicate the potentiality and importance of this line of research also in the scope of HTM.

3. SYNCHRONIZATION MECHANISMS CONSIDERED IN THE STUDY

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 strategies. Contrarily to the other approaches we used, fine-grained locking requires a *per-application* lock allocation strategy, which is a non-generalizable and error-prone task [32].

STM — With STM, reads and writes to shared memory (inside atomic blocks) are instrumented to detect conflicts between transactions. This instrumentation induces overheads that can have a detrimental impact on the efficiency of STMs. Yet, much research has been devoted over the last years to reduce STM's overheads. For our study we selected four state of the art STMs, which are representative of different choices in the design space of TM. These include an STM optimized for validations at commit-time (TL2 [13]); to maximize performance at low thread counts (NOrec [7]); in high contention scenarios (SwissTM [17]); and to minimize instrumentation costs (TinySTM [18]).

HTM — HTM implements a concurrency control scheme in hardware, avoiding the overheads of STM instrumentation. In our study we consider Intel TSX's implementation of HTM, which is integrated in the family of Intel commodity processors (Haswell). One fundamental design principle of TSX (and 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 the L1 cache to buffer transactional writes, and relies on the cache coherence protocol to detect conflicts. A plausible reason for a transaction to fail in hardware is because its data footprint exceeds the L1 capacity. Hardware transactions are also subject to abort due to reasons like page faults and system calls.

As a result, a fallback software synchronization mechanism must be provided to ensure progress in case a transaction cannot be committed with HTM. The software fallback mechanism must co-operate with the hardware in order to ensure correctness. As we shall see, the mechanisms used to coordinate the execution of the fallback mechanism have a crucial impact on the performance of TSX. The simplest approach, as suggested by Intel's optimization manual, is to use a single lock to protect atomic blocks (we call it TSX-GL). When a hardware transaction aborts, it has the alternative to acquire the global lock instead. To ensure a correct interplay with the fallback, hardware transactions must read the lock as free to guarantee correctness; the transactional semantics will guarantee that the transaction commits only when there is no ongoing fallback execution.

An obvious extension of this idea is to use fine-grained locks (TSX-FL). As the TM abstraction is motivated by the need of relieving programmers from the complexity of designing locking schemes, the usage of fine-grained locks as a fallback for HTM sounds somewhat contradictory. However, this choice allows us to assess to what extent a simplistic fallback (using a single lock) can hinder parallelism. Also, fine-grained locks may be automatically crafted, to some extent, by using recent techniques based on static analysis [29].

Table 1: Mechanisms used for synchronization in our study.

Mechanism	Description
Locks	Coarse/fine-grained locking [10]: TTAS, Spin, RW (pthreads), MCS, CLH, Ticket
STMs	TL2 [13], TinySTM [18], SwissTM [17], NOrec [7]
HTMs	TSX-GL [37] (global lock), TSX-FL (fine locks)
HyTMs	TSX-TL2 [30], TSX-NOrec [8]

HyTM — Another mechanism proposed in the literature is to use an STM as fallback for HTM, also known as Hybrid TM (HyTM). Its main advantage is to allow concurrent execution of hardware transactions and software ones, used in the fallback. However, during their concurrent execution, both software and hardware transactions have to play along in order to preserve correctness. In our study we considered two state of the art Hybrid TM proposals [30, 8], which are evaluated for the first time on a commodity HTM. We exploit 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 generate highly heterogeneous workload domains. We excluded the Bayes application given its non-deterministic executions, and used standard parameters for each application. STAMP contains manually instrumented reads and writes inside atomic blocks to invoke the STM-based synchronization. Naturally, this is not relevant for the case of pure HTM approaches. We shall additionally present results for compiler based instrumentation in Section 8.

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 relatively large instrumentation overheads. Finally, we also used a recent TM-based porting of the popular Memcached [35], an in-memory cache, widely used for instance at Facebook [31].

Each experiment is the average of 20 executions. We use the geometric mean whenever we show an average of normalized results. We often show speedup results, which are relative to the performance of sequential, non-instrumented executions, unless stated otherwise. The reported measurements of power consumption were obtained via the Intel RAPL [9] facility and are restricted to the processor and memory subsystems. Recent studies ([23, 24]) show that the model used by Intel RAPL estimates quite accurately the power consumption, when compared to a power meter attached to the machine.

Our machine is equipped with an Intel Haswell Xeon E3-1275 3.5GHz processor and 32GB RAM. This choice is dictated by the requirement of using a processor equipped with TSX, which is limited for now to 4 cores (and 8 hyper-threads). 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.

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. As we shall see, this 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.

We begin in Section 5.1 by comparing the performance and energy efficiency when using six locks implementations to implement the fallback mechanism of TSX. This shall allow us to narrow down the multitude of combinations of TSX and lock implementations assessed in our study. Next, in Section 5.2, we optimize TSX-GL with a recently proposed technique [1] aimed at reducing spurious hardware aborts. This shall provide some 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 above the average across all parallelism degrees and benchmarks.

Table 2 shows the performance of TSX given the backing lock implementation 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.

For each benchmark and thread count, we additionally sorted the considered lock implementations according to either their performance or power consumption, determining in this way their rank for that benchmark/configuration. 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 fallback path [1].

The idea is that it may not be safe for transactions to execute speculatively in hardware if, at the same time, some transaction is executing pessimistically after acquiring the global lock. A pessimistic execution cannot restart, and hence 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: the auxiliary lock serves as a manager preventing hardware aborts from continuously acquiring the fallback lock and preventing hardware speculations.

In Table 3 we compare TSX using the auxiliary lock against a single-lock (TSX-GL). For this, we report values for time, energy, and Energy Delay Product (EDP), normalized with respect to TSX-GL (analogously to traditional speedup metrics). We report the average across either benchmarks or threads. 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

Table 2: Overhead (%) of each lock (as fallback of TSX) with respect to the optimal choice in each execution.

-	Performance		Power	
Lock	Overhead (%)	Rank	Overhead (%)	Rank
Ticket	1.0	1.75	1.1	1.75
MCS	2.4	2.62	1.2	2.25
CLH	2.9	3.62	2.4	3.38
RW	14.2	4.89	17.4	3.88
TTAS	15.2	5.00	17.4	4.88
Spin	16.4	5.00	17.5	4.88

Table 3: Normalized performance of auxiliary lock [1] over TSX-GL across benchmarks and threads (higher is better).

Avg across Benchmarks				Avg across Threads			
	time	energy	edp		time	energy	edp
genome	1.58	1.6	2.54	1	1.00	1.00	1.01
intruder	1.80	1.95	3.52	2	1.08	1.06	1.14
kmeans	1.20	1.17	1.40	3	1.14	1.12	1.28
labyrinth	1.01	1.01	1.01	4	1.29	1.26	1.62
ssca2	1.00	1.00	1.00	5	1.26	1.25	1.57
vacation	1.52	1.48	2.25	6	1.26	1.23	1.55
yada	0.96	0.96	0.92	8	1.26	1.23	1.55

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 since the high conflict probability limits 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, of course, transactions may abort due to data contention. In these situations one may aggressively trigger the fallback, or opt to insist on 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 set the number of retries to 5, which is the configuration reported to deliver best all-around performance with TSX [37, 28] (a result that we have confirmed with TSX-GL on our testbed). For the HyTMs, 4 times was found to be 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. In the following, 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 contention level between transactions, and the percentage of the workload that is transactional. We then identify the 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. 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 emerges 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. TSX-GL is desirable when transactions are small, generate low/medium contention, and the application does not spend all the time executing 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 often 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 Sections 6.1-6.2 we present our experiments with STAMP. We will 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 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× 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 com-

Table 4: Summary of results according to the workload characterization of the STAMP suite (xt = number of threads).

	Time in Tx (%)	Contention	Best Performing	Least Power Consumption
kmeans	low (7)	low	TSX-GL	TSX-GL
ssca2	low (17)	low	TSX-GL	TSX-GL
intruder	medium (33)	high	TSX-GL $\leq$ 4t; TinySTM $\geq$ 5t	TSX-GL $\leq$ 5t; TinySTM $\geq$ 6t
vacation	high (89)	low	TSX-GL $\leq$ 2t; TinySTM $\geq$ 3t	TSX-GL $\leq$ 4t; TinySTM $\geq$ 5t
genome	high (97)	low	TinySTM	TinySTM
yada	high (99)	medium	SwissTM	TinySTM
labyrinth	high (100)	high	STMs (except TL2)	STMs (except TL2)

petitive 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 poorly in this benchmark, this also drags TSX-TL2 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 it.

Vacation: Once again we see that the performance of TSX-TL2 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 lazily at commit-time. This results in TL2 doing useless work more often, whereas SwissTM and TinySTM restart the speculation faster when reacting to conflicts. On the other hand, NOrec is 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. Analogous results are achieved for what regards power consumption. It is interesting to note that TSX-GL performs worse than TSX-NOrec at 8 threads, but the two consume approximately the same power. This is a result of the power savings that are achievable with the lock acquisition in TSX-GL.

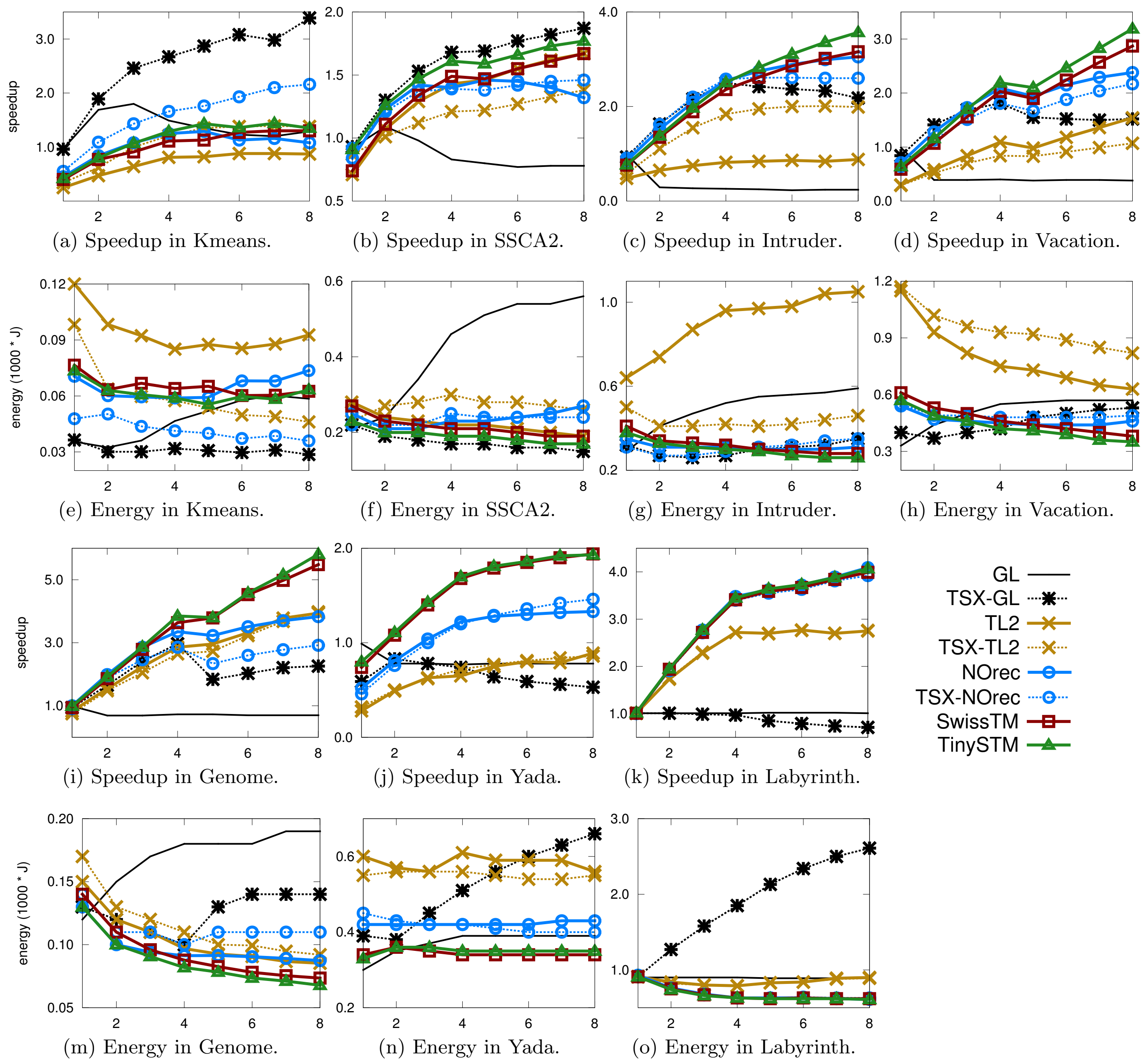
Genome: In the three last benchmarks we have either transaction-heavy or high-contention workloads, characterized by large transaction foot-prints. These conditions are clearly a much more favourable playground for STMs. In this case, we see a clear (and consistent across benchmarks) distinction between 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 5^{th} thread is used: all approaches stabilize (or even decrease) performance at that point, due to hyper-threading. Interestingly, this effect is not so noticeable on the 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 it also consumes little additional power. This fact is favourable to STMs, as TSX approaches generate 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 the energy-efficiency, whose trends are highly correlated with the performance.

Labyrinth: Here we 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 (at least when used with TSX), or if it

Table 5: Transactional abort rate (%). For STM we show the lowest and highest values obtained (across all considered STMs).

	benchmark	*kmeans*	*ssca2*	*intruder*	*vacation*	*genome*	*yada*	*labyrinth*
1 thread	STM	0 - 0	0 - 0	0 - 0	0 - 0	0 - 0	0 - 0	0 - 0
	TSX-GL	0	0	7	49	11	46	95
	TSX-TL2	0	0	36	94	35	19	53
	TSX-NOrec	0	0	4	40	6	19	53
4 threads	STM	10 - 34	0 - 0	0 - 0	0 - 6	1 - 51	5 - 58	4 - 13
	TSX-GL	26	0	22	69	31	48	100
	TSX-TL2	50	74	74	100	45	84	60
	TSX-NOrec	31	46	29	66	17	31	55
8 threads	STM	25 - 54	0 - 0	3 - 57	0 - 10	0 - 1	7 - 65	8 - 23
	TSX-GL	42	1	33	72	48	47	100
	TSX-TL2	60	99	92	100	53	92	69
	TSX-NOrec	44	88	62	99	69	39	60



(a) Speedup in Kmeans. (b) Speedup in SSCA2. (c) Speedup in Intruder. (d) Speedup in Vacation.

(e) Energy in Kmeans. (f) Energy in SSCA2. (g) Energy in Intruder. (h) Energy in Vacation.

(i) Speedup in Genome. (j) Speedup in Yada. (k) Speedup in Labyrinth.

(m) Energy in Genome. (n) Energy in Yada. (o) Energy in Labyrinth.

Figure 1: Speedup (relative to non-instrumented sequential execution) and Energy Consumption (in Kilo Joules) when varying the number of threads (horizontal axis) in all the STAMP benchmarks

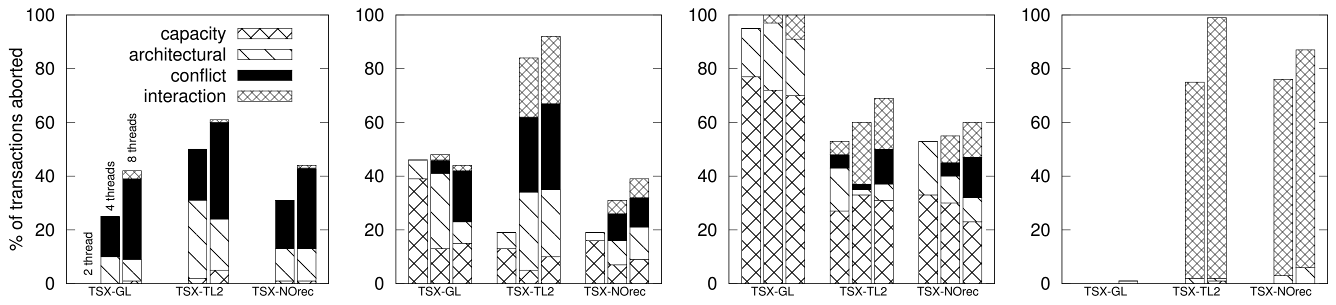
would be preferable to adaptively employ the most promising technique (TSX-GL or an STM) based on the workload.

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 and SwissTM, whereas TL2 yields the maximum abort rate.

Once again, we structure the table considering the different categories of workloads. As we move right (more contended or transaction-intensive workloads) and down (higher degree of parallelism), TSX approaches increase 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 distinguish aborts caused by exceeding the capacity of L1 cache; micro-



(a) Kmeans. (b) Yada. (c) Labyrinth. (d) SSCA2.

Figure 2: Breakdown of reasons causing transactional aborts for TSX variants.

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' breakdown shows that, as expected, as concurrency increases, also abort rates increase due mainly to data conflicts. It is worth mentioning that Kmeans is the benchmark with the least average aborts for TSX variants. Half of the aborts are due to conflicts, 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. In TSX-GL, instead, the fallback executes non-speculatively 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 HyTMs have enormous abort rates, dominated by the interaction with the fallback path.

This motivates to better understand the usage of the fallback path in the HyTMs. Table 6 shows the percentage of transactions that were executed in the fallback (i.e., not purely in hardware). We also show the percentage of transactions in the fallback that are able to execute in a fast mode, i.e., a mode in which the transaction executes in software but the commit is boosted by using a reduced hardware transaction [30] (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, a higher thread count typically 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 in Section 6: 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 in [8] that, in order to support efficient HyTMs, it is desirable to have hardware support for selective non-transactional memory accesses in 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 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 and energy, but also in terms of architectural intrusiveness.

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).

	TSX-TL2		TSX-NOrec	
	fallback	Fast	fallback	Fast
kmeans	< 1 - 77	92 - 32	< 1 - 78	100 - 23
ssca2	< 1 - 99	91 - 2	< 1 - 86	95 - 14
intruder	33 - 88	98 - 39	3 - 55	100 - 52
vacation	94 - 100	43 - 3	38 - 99	100 - 89
genome	50 - 100	97 - 71	6 - 67	100 - 94
yada	18 - 78	50 - 34	17 - 32	99 - 82
labyrinth	58 - 100	14 - 2	54 - 98	10 - 3

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

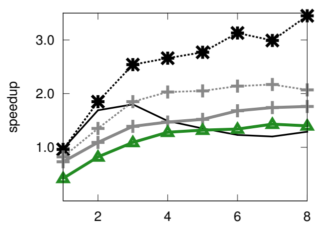
As already mentioned, implementing a fine-grained locking strategy is a complex task for most of the STAMP benchmarks. We were, however, able to devise fine-grained locking strategies for three of the STAMP benchmarks, whose results we report in Fig. 3. Besides fine-grained locks (FL), we also show results 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.

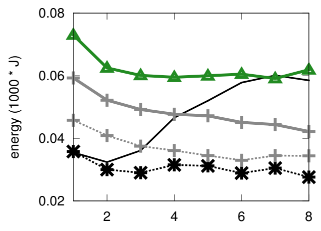
Recall that Kmeans and SSCA2 were the two benchmarks with workload characteristics more amenable to TSX-GL. This is justified by the low frequency of activation of the fallback path. As such, TSX-GL incurs minimal overhead thanks to the hardware speculation and to the avoidance of any software-based 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.

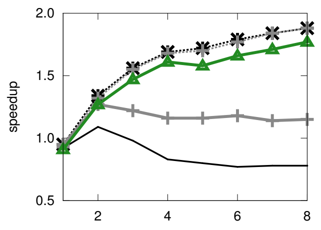
Finally, Intruder spends a large fraction of time within atomic blocks. As already discussed, this workload is more advantageous for STMs than for TSX. It is not surprising to see that TSX-GL is no longer the most competitive choice (although it still fares best until 3 threads). The interesting fact is that this kind of workload is more beneficial for 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 closer to TinySTM, following the trend of STMs that often fare worse on the energy side to obtain comparative levels of performance to the other approaches.



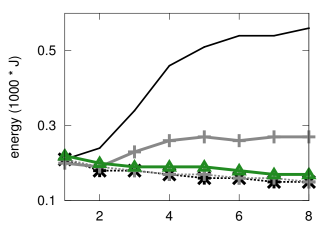
(a) Speedup in Kmeans.



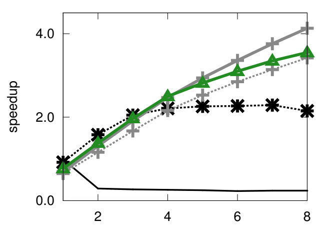
(b) Energy in Kmeans.



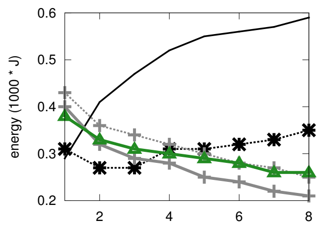
(c) Speedup in SSCA2.



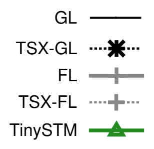
(d) Energy in SSCA2.



(e) Speedup in Intruder.



(f) Energy in Intruder.



Strategy	$10^3 *$ Operations/sec
GL	223
TSX-GL	467
FL	501
TSX-FL	481
TinySTM	329

(h) Throughput in Memcached.

Figure 3: Experiment similar to that in Fig. 1, but instead using fine-grained locking and Memcached.

7.2 Memcached

Memcached is a popular distributed object caching system [31]. In this study, we rely on a recent TM-based porting [35], and use the original Memcached as the basis for FL. We used the memslap tool, configuring the workload with 95% gets and 5% puts, 8 threads and a concurrency of 256.

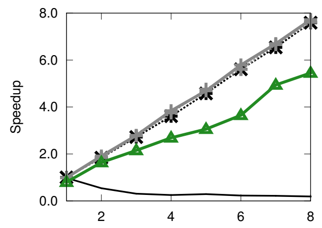
In Memcached it is not really possible to measure, for reference purposes, the performance of a sequential execution, because there is always concurrency due to the existence of a pool of maintenance threads. Hence, we present the peak throughput obtained using the maximum number of available hardware threads (see Fig. 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 fine-grained locks is considerably higher than using TSX-GL. Also, since FL is quite optimized, it is expectable that TSX-FL is not able to extract any further parallelism. Interestingly, with this benchmark, TinySTM is not competitive because the instrumentation overheads are amplified by the short and uncontended transactions.

7.3 Concurrent Data Structures

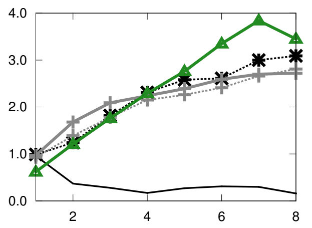
We now consider two concurrent data structures, namely a red-black tree and a hashmap, which represent particularly relevant use cases for TM given the complexity of designing efficient fine-grained locking strategies for these scenarios.

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 solutions.

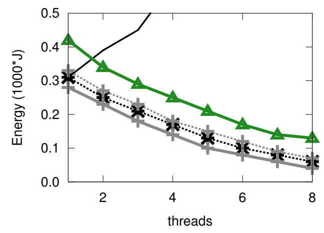
Table 4(e) shows a spectrum of workloads in red-black tree, by considering the normalized EDP of TSX-GL against the best alternative in each experiment. For this, we vary the size of the tree and the percentage of write transactions. The trend is clear in this table: TSX behaves best with light workloads,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



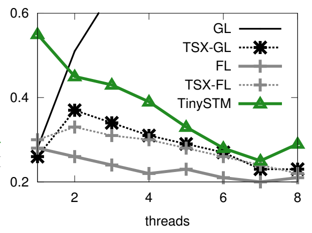
(a) Hashmap 10% writes.



(b) RB Tree 90% writes.



(c) Hashmap 10% writes.



(d) RB Tree 90% writes.

size and % of writes	10%	50%	90%
2^7 elements	3.30	1.79	1.21
2^{14} elements	2.96	1.58	1.11
2^{21} elements	1.86	1.33	1.06

(e) Normalized EDP of the best alternative to TSX-GL in Red-black Tree (higher is favourable to TSX-GL).

Figure 4: Data Structures varying contention level.

8. RESEARCH DIRECTIONS SUGGESTED BY OUR STUDY

We now identify some relevant research directions that emerged from the analysis of our experimental study:

• The overall performance of the tested HyTM solutions is quite disappointing. These findings contradict the simulation results published in several previous works, e.g., [30]. 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. [33]). 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 for adaptively tuning 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 [34] or analytical models [11]) is going to be strongly amplified in HTM settings because there exists no instrumentation as in STMs. In the light of these findings, we have concurrently obtained some initial results with regard to this research direction [15].

• Our study has used (selective) manual instrumentation when considering both STMs and HyTMs, i.e. 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

Table 7: Improvement of configuring TSX-GL for each workload compared to the single configuration used in our study.

Speedup %	kmeans	ssca2	intruder	vacation	genome	yada	labyrinth
4 threads	12	7	20	36	12	13	2
8 threads	5	8	80	21	2	55	39

obtained using the C++ TM extension integrated in GCC 4.8.2, show that non-selective instrumentations can impact performance by approximately 20% when using TinySTM. This is a consequence of the increase of the transaction footprint (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 support 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) and sufficiently non-intrusive to simplify integration in existing architectures.

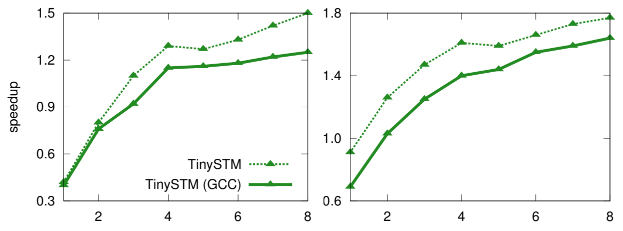
9. CONCLUSIONS

This paper analyzed extensively the performance and energy efficiency of several state of the art 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 synchronization alternative in a set of relevant workloads. On the other hand, it also identified some critical limitations of Intel TSX, and highlighted the robustness of state of the art STMs. These software implementations achieve performance competitive with fine-grained locking, and outperform HTM in workloads encompassing long and contention-prone transactions.

Furthermore we have shown that the performance of Hybrid TM, when used in combination with TSX, is normally quite 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, we believe, should be timely addressed to increase the chances of turning HTM into a mainstream paradigm for parallel programming.

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

This work was supported by funds from Fundação para a Ciência e Tecnologia under PEst-OE/EEI/LA0021/2013 and by GreenTM EXPL/EEI-ESS/0361/2013.



(a) Kmeans.　(b) SSCA2.

Figure 5: Impact of GCC instrumentation.

10. REFERENCES

[1] Y. Afek, A. Levy, and A. Morrison. Programming with hardware lock elision. In *Proc. of Principles and Practice of Parallel Programming*, PPoPP, pages 295–296, 2013.

[2] A. Baldassin, F. Klein, G. Araujo, R. Azevedo, and P. Centoducatte. Characterizing the Energy Consumption of Software Transactional Memory. *IEEE Comput. Archit. Lett.*, 8(2):56–59, July 2009.

[3] H. Cain et al. Robust architectural support for transactional memory in the power architecture. In *Proc. of International Symposium on Computer Architecture*, ISCA, pages 225–236, 2013.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. of International Symposium on Workload Characterization*, IISWC, pages 35–46, 2008.

[5] C. Cascaval et al. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):40:46–40:58, Sept. 2008.

[6] D. Christie et al. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *Proc. of EuroSys*, pages 27–40, 2010.

[7] L. Dalessandro, M. Spear, and M. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proc. of Principles and Practice of Parallel Programming*, PPoPP, pages 67–78, 2010.

[8] L. Dalessandro et al. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In *Proc. of Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 39–52, 2011.

[9] H. David, E. Gorbatov, U. Hanebutte, R. Khanna, and C. Le. RAPL: memory power estimation and capping. In *Proc. of International Symposium on Low power Electronics and Design*, ISLPED, pages 189–194, 2010.

[10] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proc. of Symposium on Operating Systems Principles*, SOSP, pages 33–48, 2013.

[11] P. Di Sanzo, B. Ciciani, R. Palmieri, F. Quaglia, and P. Romano. On the analytical modeling of concurrency control algorithms for software transactional memories: The case of commit-time-locking. *Perform. Eval.*, 69(5):187–205, May 2012.

[12] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proc. of Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 157–168, 2012.

[13] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of International Symposium on Distributed Computing*, DISC, pages 194–208, 2006.

[14] N. Diegues and J. Cachopo. Practical Parallel Nesting for Software Transactional Memory. In *Proceedings of International Symposium on Distributed Computing*, DISC, pages 149–163, 2013.

[15] N. Diegues and P. Romano. Self-Tuning Intel Transactional Synchronization Extensions. In *Proc. of International Conference on Autonomic Computing*, ICAC, 2014.

[16] N. Diegues and P. Romano. Time-Warp: Lightweight Abort Minimization in Transactional Memory. In *Proc. of Principles and Practice of Parallel Programming*, PPoPP, pages 167–178, 2014.

[17] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *Proc. of Programming Language Design and Implementation*, PLDI, pages 155–165, 2009.

[18] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. of Principles and Practice of Parallel Programming*, PPoPP, pages 237–246, 2008.

[19] C. Ferri, R. I. Bahar, M. Loghi, and M. Poncino. Energy-optimal Synchronization Primitives for Single-chip Multi-processors. In *Proc. of Great Lakes Symposium on VLSI*, GLSVLSI, pages 141–144, 2009.

[20] C. Ferri, S. Wood, T. Moreshet, R. Iris Bahar, and M. Herlihy. Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems. *J. Parallel Distrib. Comput.*, 70(10):1042–1052, Oct. 2010.

[21] E. Gaona, R. Titos, J. Fernandez, and M. Acacio. On the design of energy-efficient hardware transactional memory systems. *Concurrency and Computation: Practice and Experience*, 25(6):862–880, 2013.

[22] A. Gautham, K. Korgaonkar, P. Slpsk, S. Balachandran, and K. Veezhinathan. The implications of shared data synchronization techniques on multi-core energy efficiency. In *Proc. of Power-Aware Computing and Systems*, HotPower, pages 1–6, 2012.

[23] D. Hackenberg et al. Power measurement techniques on standard compute nodes: A quantitative comparison. In *Proc. of the International Symposium on Performance Analysis of Systems and Software*, ISPASS, pages 194–204, 2013.

[24] M. Hähnel, B. Döbel, M. Völp, and H. Härtig. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Performance Evaluation Review*, 40(3):13–17, Jan. 2012.

[25] L. Hammond et al. Transactional Memory Coherence and Consistency. In *Proc. of International Symposium on Computer Architecture*, ISCA, pages 102–113, 2004.

[26] A. Heindl and G. Pokam. An Analytic Framework for Performance Modeling of Software Transactional Memory. *Comput. Netw.*, 53(8):1202–1214, June 2009.

[27] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proc. of the International Symposium on Computer Architecture*, ISCA, pages 289–300, 1993.

[28] T. Karnagel et al. Improving In-Memory Database Index Performance with Intel TSX. In *Proc. of International Symposium on High Performance Computer Architecture*, HPCA, 2014.

[29] S. Mannarswamy, D. Chakrabarti, K. Rajan, and S. Saraswati. Compiler aided selective lock assignment for improving the performance of software transactional memory. In *Proc. of Principles and Practice of Parallel Programming*, PPoPP, pages 37–46, 2010.

[30] A. Matveev and N. Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 11–22, 2013.

[31] R. Nishtala et al. Scaling Memcache at Facebook. In *Proc. of Conference on Networked Systems Design and Implementation*, NSDI, pages 385–398, 2013.

[32] V. Pankratius and A.-R. Adl-Tabatabai. A Study of Transactional Memory vs. Locks in Practice. In *Proc. of Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 43–52, 2011.

[33] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proc. of the Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 53–64, 2011.

[34] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia. Machine Learning-Based Self-Adjusting Concurrency in Software Transactional Memory Systems. In *Proc. of Modeling, Analysis Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 278–285, 2012.

[35] M. Spear, T. Vyas, and Y. Ruan, Wenjia Liu. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 398–412, 2014.

[36] A. Wang et al. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proc. of Parallel Architectures and Compilation Techniques*, PACT, pages 127–136, 2012.

[37] R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *Proc. of High Performance Computing, Networking, Storage, and Analysis*, SC, pages 1–11, 2013.