Synchronization Strategies on Many-Core SMT Systems

Agustín Navarro-Torres, Jesús Alastruey-Benedé, Pablo Ibáñez-Marín
Computer Architecture Group of Unizar (gaZ)
Universidad de Zaragoza
Zaragoza, Spain
{*agusnt,jalastru,imarin*}*@unizar.es*

Maria Carpen-Amarie
Zürich Research Center
Huawei Technologies Switzerland
Zürich, Switzerland
maria.carpen.amarie@huawei.com

***Abstract*—The complexity of efficient synchronization design increases with the continuous growth in the number of physical and logical cores on today's machines. Opinion is divided on which synchronization strategy is more powerful, opposing typical mechanisms, such as locks and atomic primitives, to emergent technologies, like transactional memory. We perform an extensive scalability study on many-core systems, evaluating most widely-used synchronization mechanisms in terms of application throughput and operation latency. We show that, from a performance perspective, current best-effort implementations of hardware transactional memory (HTM) are comparable to well-established locking or lock-free mechanisms. We also find that they scale better with the number of threads. We then showcase the ease-of-use of HTM in real-life applications. Finally, we analyze the impact of simultaneous multithreading (SMT) technologies on HTM performance. We propose a new cache replacement strategy that takes into account the transactional state of each cache line and aims to mitigate SMT-induced transactional overflow aborts.**

***Keywords*-Many-core systems, Scalability, Transactional memory, Intel Transactional Synchronization Extensions**

I. Introduction

Nowadays, multicore systems are the norm for virtually all kinds of commodity computing devices, ranging from mobile to large-scale server machines. The number of native threads can reach hundreds [1], [2], further growing with the increasing number of cores per socket. This paves the way for a higher degree of parallelism and, thus, for better performance across concurrent applications. However, efficient concurrency comes at a price, oftentimes paid in increased complexity for the synchronization mechanism. Complex synchronization, such as typical fine-grained locking, requires in-depth understanding of the concurrent program's structure and internal interactions, while it may still lead to unsound implementations, faulty executions, deadlocks, etc.

In the last decade, the transactional memory (TM) paradigm captured the research community's interest by proposing a simpler way of reasoning about concurrent programs and seamlessly performing all needed synchronization in the background. The concept of transactional memory was introduced almost 30 years ago by Herlihy and Moss [3]. However, since no hardware infrastructure was readily available to provide TM, software transactional memory (STM) was the first heavily studied category. Its usefulness in real-life scenarios was fiercely disputed [4], [5]. Once present in commodity processors, hardware transactional memory (HTM) quickly became a hot topic and was progressively adopted by all major hardware vendors.

There are various studies on the performance of synchronization mechanisms: e.g., comparisons between TM systems [6], between locking techniques [7] or even HTM versus classical locks and atomic primitives [8]. However, to the best of our knowledge, there is no study that evaluates the scalability of all synchronization mechanisms together on many-core systems and that takes into account the impact of simultaneous multithreading (SMT). Furthermore, while SMT is recognized as a limiting factor for HTM performance [9], its impact is not clearly quantified and no solution has been proposed to mitigate it.

We aim to bridge this gap with an extensive study on the scalability of various synchronization strategies, with a particular focus on the impact of SMT. We rely on concurrent data structures for a sound evaluation of scalability. We consider a hash-table and a binary search tree (BST), executing multiple workloads with varying ratios of lookup and update operations. These data structures are synchronized with: fine-grained locking, atomic primitives, HTM, and STM. We perform a detailed evaluation of throughput and operation latency. For HTM, we further analyze the breakdown of transactional commit and abort events and correlate it with the performance results. In addition, we briefly make the case of HTM's ease-of-use; we illustrate it with the PARSEC 3.0 benchmark suite, as a speed-up comparison between HTM and fine-grained locking. Finally, we propose a solution for reducing the negative impact of SMT on HTM performance.

This work makes the following contributions:

- **Extensive scalability analysis** for all classes of synchronization mechanisms. Our results show that HTM is a scalable alternative to atomic primitives, improving throughput with up to 60% and 37% on 88 threads, for the hash-table and BST, respectively. Overall, HTM maintains a comparable throughput with lock-free and fine-grained locking implementations. Moreover, our operation latency analysis shows that, regardless of aborts and rollbacks, HTM operations have tail-latency

This work was done while the first author was an intern at Huawei Zürich Research Center.

generally lower than or comparable to both fine-grained locking and lock-free versions. For the hash-table, HTM's tail-latency is up to 10x lower than that of classical synchronization mechanisms, depending on contention. STM stands out as having considerably higher overall operation latency: e.g., in the hash-table case, 99% of STM operations take one order of magnitude longer than the operations in the three other versions for the same interval. The results are similar for the binary search tree.

- **Observations regarding HTM adoption on many-core systems**. We showcase HTM's ease-of-use on the PARSEC 3.0 benchmarks, concluding that a basic, best-effort HTM implementation offers comparable scalability to highly optimized fine-grain locks, while coming for free with the glibc library.
- **In-depth evaluation of SMT impact** on synchronization performance, in particular for HTM. Our experiments with PARSEC 3.0 show that exploiting SMT always results in an increased number of transactional capacity overflow aborts. When executing on two hardware threads, we observe an up to 69x increase in the number of aborts in this category with respect to a single thread per core.
- **Hardware solution for improving HTM performance with SMT.** We present and evaluate *Transaction-Aware LRU* (TA-LRU), a cache replacement policy that achieves a 16x reduction of overflow aborts on a synthetic micro-benchmark executed in the gem5 simulator.

The rest of this paper is organized as follows: Section II presents the context of this work. In Section III, we describe in detail the scalability study on concurrent data structures. Section IV extends the evaluation with a comparison of HTM and fine-grained locking synchronization on a widely-used benchmark suite. Finally, we quantify the negative impact of the SMT feature on HTM capacity aborts and introduce our solution in Section V. Section VI concludes and presents future research directions.

II. Background

We evaluate the behavior and performance of four different synchronization mechanisms: locks, atomic instructions, STM, and HTM.

A. Synchronization Strategies

Classical strategies. Most parallel applications are implemented using locks or atomic instructions. A lock controls the access to data regions shared by multiple threads. Programmers usually use multiple locks (*fine-grain*) to increase the parallelism and performance of the applications. However, fine-grain locks require in-depth knowledge of the application and errors such as deadlocks or race conditions are not uncommon. Lock-free algorithms rely on atomic instructions, such as *compare-and-swap (CAS)*, to manage concurrency among threads. They do not suffer from lack of scalability, but require extensive knowledge on the underlying processor architecture and the program's structure.

Transactional memory. According to the literature [10], transactional memory seems to be a promising mechanism for synchronizing processes. TM enables the programmer to mark a section of code, specifying that it has to be executed as a transaction. The TM system guarantees that transactions are executed atomically. TM presents two main benefits over classical synchronization strategies: **(1)** ease-of-use, while also avoiding well-known concurrency issues, such as race conditions; **(2)** optimistic execution, allowing for a higher degree of parallelism.

Transactional memory comes in three flavors: software (STM), hardware (HTM), and hybrid (a combination of both software and hardware). In general, STM libraries [11], [12], [13], [14], [15] instrument applications' code to detect and solve memory conflicts. The high flexibility given by its software implementation is tarnished by its instrumentation performance overhead. HTM [16], [17], [18], [19], on the other hand, aims to address this performance concern, but its hardware constraints limit the flexibility of the solution. In 2009, Sun Microsystems was the first company to announce a multicore processor with HTM support, codenamed Rock [20]. Later, HTM support was implemented in commercial processors by other hardware vendors, such as Intel (starting with the Haswell family) and IBM (POWER [21], Blue Gene/Q [22], the zSeries [23]).

Intel's HTM implementation goes by the name of Transactional Synchronization Extensions (TSX). It has two different interfaces: *Intel Hardware Lock Elision (HLE)* and *Intel Restricted Transactional Memory (RTM)*. Intel TSX uses the cache hierarchy to track write and read-sets [24], and the coherence protocol to detect conflicts at a cache-line granularity. It is a best-effort TM system. More precisely, it does not guarantee that a hardware transaction will ever commit, thus introducing the need for a software fallback to provide progress using another synchronization mechanism. Most often, a global lock is used inside the fallback path, but more complex fine-grained locking schemes can be exploited for added efficiency [25], [26]. There are several considerations when providing a transactional fallback-path. On the one hand, the abort reason needs to be taken into account when deciding whether the transaction should be restarted, depending on the chances of commit on a retry. On the other hand, the number of retries needs to be well-balanced: if there are too few retries, the transaction may be serialized prematurely; if there are too many, the eventual commit may not amortize the cost of the rollbacks. Finally, a back-off time may be inserted between retries, in order to reduce conflicts.

B. Related Work

There is extensive literature on synchronization mechanisms scalability. Guiroux et al. [27] compare the performance of 27 lock algorithms on 35 real-world large-scale applications on many-core systems. On the same note, Rico et al. [28] present an extensive scalability study on 4 STM libraries. Similarly, Brown et al. [29] analyze HTM performance on a many-core NUMA system (up to 72 threads), formulating guidelines on efficient use of HTM in a many-socket setup. All these works focus on a single synchronization strategy and make a deep-dive into its scalability performance and issues. By contrast, our work compares all major classes of synchronization mechanisms in order to provide a broader picture on their performance in a many-core context (up to 88 threads).

Only a few studies perform a direct comparison between synchronization methods. Park et al. [8] and Schindewolf et al. [30] experiment with HTM, locks and atomic primitives on a 64-thread setup, both using a synthetic microbenchmark suite that emulates HPC applications. Our work also brings STM into the equation and evaluates scalability on widely-used concurrent data structures. Yoo et al. [31] address the same synchronization mechanisms as us, but on a very low thread count. In addition, we analyze the evolution of HTM events with the increasing number of threads and provide a detailed breakdown on commits and various abort types.

Nakaike et al. [32] and Dice et al. [20] provide an in-depth characterization of HTM for different HTM implementations. While their analyses go into great architectural detail, they do not study the SMT impact on the working-set size limit of transactions. Hasenplaugh et al. [24] and Wang et al. [9] briefly look at capacity aborts in SMT systems, but do not go as far as proposing a solution to mitigate SMT effects on HTM performance. The recent work of Cai et al. [33] aims to shed some light on the way in which transactional structures are tracked in hardware and on the impact of the replacement policy on capacity aborts. They find that flushing or warming the cache maximizes the read-set capacity of a transaction. They do not investigate the impact of SMT in this context.

III. Scalability Analysis of Synchronization Mechanisms

We focus on two widely-used concurrent data structures, a hash-table and a binary search tree, representing typical building blocks in the development of large-scale systems. We implement them with four different synchronization mechanisms: atomic primitives [34], [35], fine-grain locks [36], STM (TinySTM engine), and HTM (Intel RTM). All implementations are optimized, avoiding typical parallel applications issues such as false-sharing, and are lock-free for lookup operations[1].

A. Experimental Setup and Methodology

Table I: Main characteristics of the workstation used in the evaluation.

Processor	2×Intel Xeon Gold 6238T
Threads	2 × 44 (88)
Family	Cascade Lake
Speed	1.9 GHz
Cores	2 × 22 (44)
TurboBoost	No
L1D	32 KiB 8-way
L2	1 MiB 16-way
LLC	22×1.375 MiB 11-way
Mem	192 GiB 6 channels
OS	Ubuntu 20.04
Kernel	5.4.0
NUMA policy	Default
Governor	Performance
Compiler	GCC 9.3.0

Table I shows our experimental setup. The threads are distributed so that they occupy as many cores as possible. In experiments on up to 44 threads, we pin each thread to the first logical core on each NUMA node in turn. Above this thread count, we move to the second logical core and continue pinning the threads in the same order as before on each NUMA node.

We implement the HTM version using C/C++ intrinsics, a global spin-lock without back-off as fallback, and a maximum of 10 retries before jumping to the fallback path. This is the simplest and most common fallback path implementation. This strategy gives us a lower-bound on HTM performance. In addition to this, we follow the recommendations laid out by Intel [37] and Bonnichsen et al. [38] to increase the chances of commit: small transactions, no system calls inside a transaction, and small memory footprint.

The lock-based version uses standard Pthread mutex locks to synchronize its critical sections. According to the extensive study on lock algorithms done by Guiroux et al. [27], Pthread locks are amongst the best performing locking structures for a variety of applications. We thus decide to rely on this implementation, rather than incurring extra overhead from a lock interposition library such as LiTL [39] or introducing unneeded complexity from a home-brewed locking algorithm.

We evaluate the concurrent data structures with three different workloads: **(1)** 100% lookup operations; **(2)** 80% lookups and 20% updates (10% insertions and 10% deletions); and **(3)** 50% lookups and 50% updates (25% insertions and 25% deletions). For brevity, we call these workloads *AllLookup*, *Update20*, and *Update50*, respectively. The workloads consist of 2^{26} predefined operations. To minimize execution variability, the sequence of operations is the same in all experiments, regardless of synchronization mechanism. The data structures are populated before each experiment with previously-generated random elements: 2^{18} for the hash-table and 2^{17} for the binary search tree.

For the analysis we rely on three metrics: **throughput**, measured in operations per second; **latency**, defined as the time it takes for an operation to be executed; and **HTM events** as the percentage of transactions committed

[1] The code is available at: https://github.com/agusnt/Synchronization-Strategies-on-Many-Core-SMT-Systems

and aborted, the abort rate being further split according to the abort reason. For the evaluation of throughput and HTM events, we execute each experiment <synchronization method, workload> 11 times and take the median of all executions. The latency analysis is based on all latency data points per workload execution, on 88 threads. For simplicity and readability, we present the data from a single run per implementation, but note that the results are consistent over multiple runs.

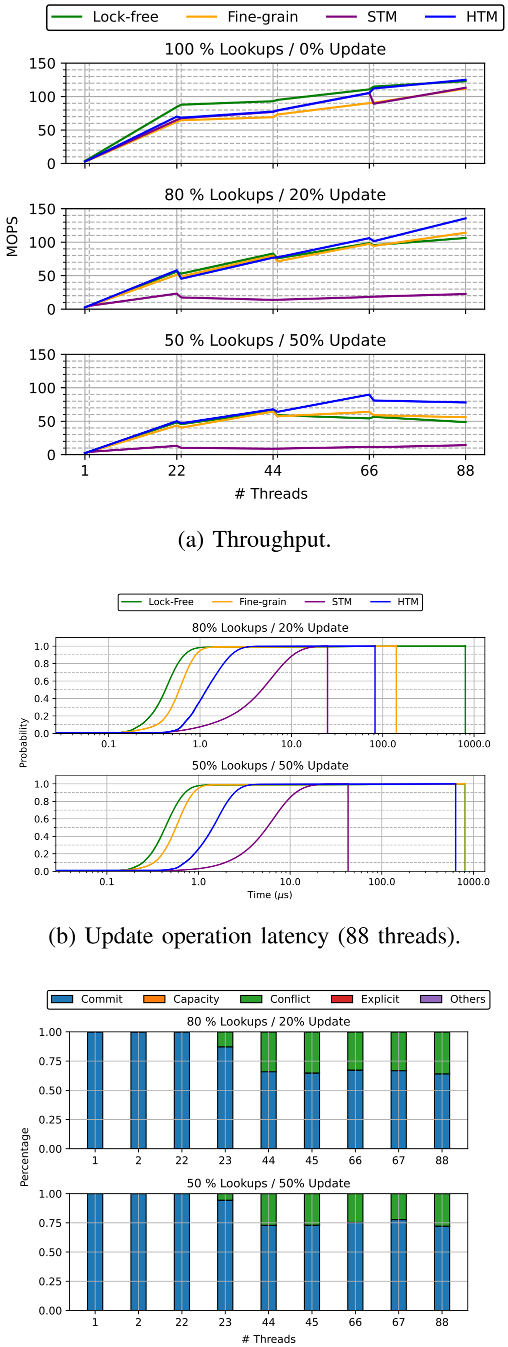
B. Concurrent Hash-Table

We implement a fixed-size hash-table with 2^{17} buckets. In order to solve potential key conflicts, each bucket contains a sorted linked-list of keys.

Throughput. We first compare the four different hash-table implementations in terms of throughput. Figure 1a shows the throughput in *Million Operations Per Second (MOPS)* on the Y axis and the number of threads on the X axis. The vertical lines mark the following transitions of interest: **(1)** from 22 to 23 threads, i.e., from using one processor to two processors; **(2)** from 44 to 45 threads, i.e., from no SMT to using SMT on the first processor; and **(3)** from 66 to 67 threads, i.e., from no SMT on the second processor to using SMT on both processors. The relative standard deviation is 5.4% on average across all workloads, contention levels and implementations, with a maximum of 18%. Only 6 datasets out of 108 present outliers (one dataset being composed of all 11 results of an experiment). More precisely, in all cases there is at most one outlier, which we believe is due to an anomaly on the machine where experiments were running. The outliers were computed with the `zscore` function.

From a performance perspective, there is no clear winner across all implementations, workloads and number of threads. For the *AllLookup* workload, all versions behave similarly, since they are all equally lock-free. The slight quantitative dissimilarity between the lines in the graph is explained by the inherent differences in the implementation of the concurrent algorithms.

The *Update20* workload shows comparable throughput for the lock-free, fine-grained locking, and HTM implementations. All three synchronization mechanisms scale with the number of threads. Starting with 45 threads, HTM performs better than the classical methods. On 88 threads, HTM achieves 18% and 27% higher throughput than the fine-grained locking version and the lock-free implementation, respectively. STM is the only synchronization mechanism not able to scale further than 22 threads. On 88 threads, the use of STM leads to a performance degradation of more than 5x on average compared to the other versions, mainly because of the overhead of instrumenting all memory accesses. We observe a performance drop (9% on average) across all implementations when we start using the second processor (in both transition points, 22 – 23 threads and 66 – 67 threads, respectively). The communication between the two processors and its impact on the cache can explain this behavior. Using the second set of logical cores (i.e., 45 threads and more), on the other hand, has a visible impact only on the lock-free and locking implementations.



(a) Throughput.

(b) Update operation latency (88 threads).

(c) HTM events.

Figure 1: Synchronization mechanisms evaluation on a concurrent hash-table.

In contrast, the more contended workload, *Update50*, has a bigger impact on performance. Up to the point where the second logical core starts being used, all synchronization mechanisms except STM follow the same trend with similar throughput. While HTM continues to scale on more than 45 threads, the lock-free and fine-grained locking versions stop scaling at 44 threads and their throughput starts decreasing when running on more than 66 threads. When the second logical core on the second NUMA node is used (transition point **(3)**), the performance of the HTM-based implementation has a drop of $\sim 11\%$ and the throughput starts decreasing. As with the *Update20* workload, the STM version does not scale after 22 threads. In addition, this experiment reveals the same performance drops at the transition points of interest, with the HTM version having also a negative spike at transition point **(2)**. We attribute this to the increased contention between transactions that have to share the same core.

In conclusion, the implementation using HTM as synchronization benefits of more parallelism, in contended scenarios in particular. While the scaling slows down after 66 threads in our configuration, it still shows 30% to 60% higher throughput than the fine-grained locking and, respectively, the lock-free versions.

Latency. We measure the latency of each operation with the `rdtsc` instruction and remove outliers with the `zscore` function. We represent the latency of update operations on 88 threads with a CDF in Figure 1b. The Y axis shows the fraction of operations that execute in less than $x\,\mu$s (X axis, logarithmic scale). The vertical lines indicate the maximum measured value for a given implementation.

Lock-free, fine-grained locking, and HTM implementations present a long tail, more prominent for the update-intensive workload (*Update50*). HTM always has lower tail-latency than the classical synchronization mechanisms. The lock-free implementation consistently shows a considerably larger tail-latency than HTM and lock-based versions. Many threads spinning on the same atomic primitive (e.g., CAS) in order to perform update operations can explain this result. Thus, on the *Update20* workload, the tail-latency of the HTM version is 2x lower than that of fine-grained locking, and 10x lower than that of the lock-free implementation. The tail-latency for these three synchronization mechanisms becomes comparable on an update-intensive workload, such as *Update50*.

In contrast, the STM implementation has a short tail, but 99% of operations take one order of magnitude longer than the other three synchronization mechanisms for the same interval. We believe this is due to our STM configuration: transactions abort immediately on conflict, and the write-set implementation facilitates low-overhead aborts and high-overhead commits. Thus, in both workloads, less than 30% of the STM operations are executed in under $3\,\mu$s, as opposed to 97% for the other synchronization methods.

HTM events. Figure 1c shows the fraction of committed and aborted transactions on the Y axis and the number of threads on the X axis. The abort rate is further split into multiple abort causes: capacity aborts (physical limitation to the size of the transaction), conflict aborts (concurrent accesses to the same memory address), and other (explicit aborts when encountering an already-acquired lock or system-level interrupts, debug instructions, I/O operations). We only show these fractions for the mixed workloads (*Update20* and *Update50*) because the lookup operations alone do not perform any transactions.

Almost all transactional aborts are due to memory conflicts. Most of them are generated by the interaction with the global lock employed in the fallback path. More precisely, all transactions monitor the state of the global lock; if one transaction repeatedly aborts for any reason and takes the fallback path, it acquires the lock, changing its state; this state change conflicts with the monitored value in all other running transactions, causing them to abort. The ratio of conflict aborts increases with the number of threads in both workloads (e.g., from 12% on 23 threads to 35% on 88 threads for *Update20*). However, we observe a lower abort rate for the update-intensive workload than for *Update20*. We believe this is due to the fact that part of the delete operations will not hit an existing element in the hash-table, while the insert operations will keep adding elements. Iterating over a larger data structure favors more and longer transaction-free lookups and, thus, fewer opportunities for conflict.

There are two main factors that lead to infrequent aborts in the other categories for this application: first, we avoid overflows by carefully planning the contents of the transactional regions and taking into account HTM size limitations; second, our hash-table implementation follows closely the aforementioned Intel guidelines, thus avoiding aborts due to unfriendly instructions. We further discuss an optimization for the fallback algorithm.

HTM fallback discussion. To avoid restarting all running transactions every time a thread takes the fallback path, a more fine-grained approach can be implemented. This may represent a convenient trade-off between simplicity and performance, depending on the application. In the case of a concurrent hash-table, this optimization is easily achievable by replacing the global lock in the fallback path with a *per-bucket lock*. This reduces the set of conflicting transactions to only the potential few that are accessing the same bucket. An even finer-grained approach (e.g., per-object lock) would significantly increase the complexity of the code with negligible performance improvement.

We briefly evaluate the fallback path optimization. Figure 2 compares the throughput of the two approaches, using a global and a per-bucket lock, on the *Update50* workload. We observe up to 2x throughput improvement (on 22 threads) and 62% higher throughput on 88 threads. Moreover, while the performance of the two implementations follows the same trend, we note the lack of negative spike at transition point **(2)** for the per-bucket approach. Statistics on HTM events show a drastic reduction of conflict aborts, which are consistently under 1%.

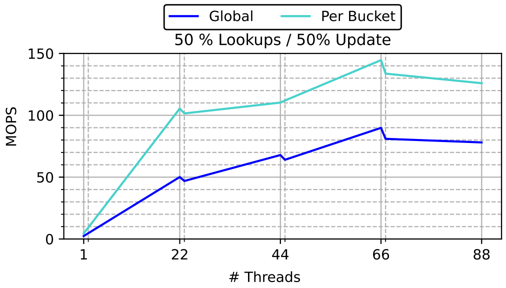


Figure 2: Global fallback lock vs. per-bucket fallback lock.

C. *Concurrent Binary Search Tree*

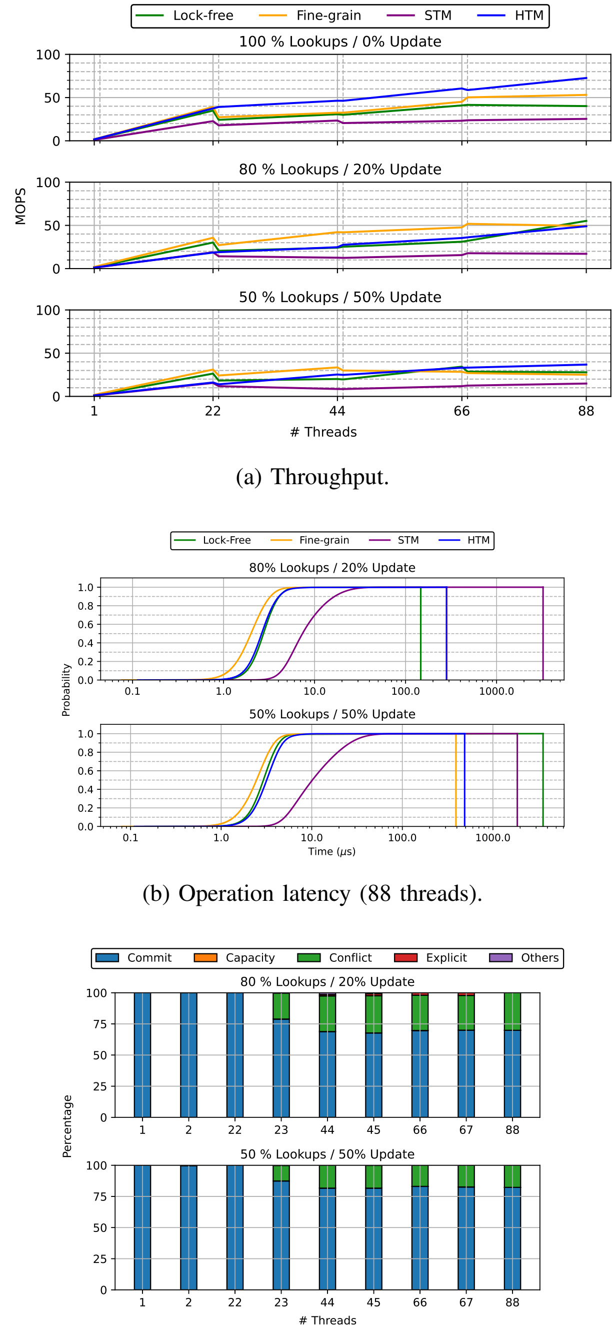
We evaluated the BST following the same methodology applied to the hash-table. The BST results confirm the findings from Section III-B and are illustrated in Figure 3.

Throughput. Overall, the measurements performed on the BST show more variability than those of the hash-table. The relative standard deviation is 8.5% in average across all workloads, contention levels and implementations of the tree data structure, with a maximum of 32%. We found and excluded a total of 3 outliers over all datasets with the `zscore` function.

As before, the STM implementation consistently shows the worst performance. For the *Update20* workload, the fine-grained locking implementation provides the highest throughput across the entire thread range. At transition point **(3)**, the locking implementation's scaling starts to slow down, while the HTM and lock-free versions continue to scale constantly. On 88 threads, all versions have comparable performance. By contrast, the update-intensive workload shows similar throughput for these three implementations throughout the experiment. The locking version stops scaling at transition point **(2)**, when the second logical core is enabled. Similarly, the lock-free implementation does not scale over 65 threads. The HTM version scales constantly until 88 threads, where it provides 53% better throughput than the lock-based version and 37% better than the lock-free one.

The state at transition points generally follows the same pattern as in the hash-table experiment. It is less prominent for the HTM-based implementation. The classical synchronization versions do not have a visible performance drop at transition point **(3)** for less contended scenarios.

Latency. Consistent with the results in the hash-table experiment, HTM exhibits tail-latency comparable to fine-grained locking, regardless of workload. The tail-latency for the lock-free implementation varies significantly with contention: it is 8x larger than that of the HTM version on the update-intensive workload, and 2x lower on the less contended one. We believe this variation comes from the nature of the lock-free algorithm. Since an updating thread is not blocking the portion of the tree it is working on, it may need to repeatedly return and restart its subtree traversal if other threads manage to change the values and the placement of the nodes before it applies its update. The update-intensive workload increases the chances that multiple back and forth iterations will take place for any update operation, thus resulting in the observed long tail.



(a) Throughput.

(b) Operation latency (88 threads).

(c) HTM events.

Figure 3: Synchronization mechanisms evaluation on a concurrent BST.

Another interesting behavior is presented by STM. Generally, when implementing concurrent data structures with

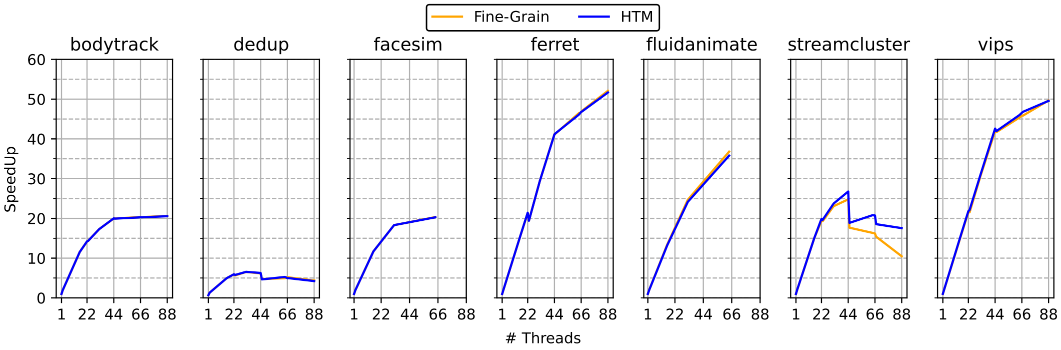


Figure 4: HTM and fine-grained locking evaluation on PARSEC 3.0 benchmarks.

STM, each operation is entirely encapsulated in a software transaction. Thus, operations have higher latency for a more complex tree structure, e.g., if a tree traversal is needed, than for a hash-table. As such, in this scenario, STM presents between 4x and 10x higher tail-latency than HTM, depending on the workload. Moreover, 90% of STM operations still take at least 3 times longer than those of other versions for the same interval.

HTM events. Figure 3c shows the ratios of transactional commits and aborts for different degrees of contention. The breakdown on abort types reveals a majority of conflict aborts. We explain this in the same way as for the hash-table. Similarly, we observe a lower abort rate for the update-intensive workload than for the *Update20* workload. This is consistent with our observations on the concurrent hash-table.

IV. Case Study: HTM Ease-of-Use

PARSEC 3.0 [40] is one of the most widely-used benchmark suites for evaluating multicore systems. It consists of 13 scientific real-world parallel applications. In this section, we compare the performance of fine-grain locks and HTM on realistic workloads.

The benchmarks being originally synchronized with a locking mechanism, the strategy we adopt for HTM synchronization is *lock-elision*, to allow for backward compatibility and minimal code modifications. The glibc library of GNU/Linux has a readily integrated version of HTM lock-elision, which can be enabled by setting the `GLIBC_TUNABLES` environment variable to `glibc.elision.enable=1`. The glibc implementation uses Intel RTM for performance and flexibility reasons [41]. This mechanism allows us to evaluate HTM in the context of a highly-optimized benchmark suite. We also modify the glibc library to retrieve the abort reason of failed transactions. We use the same experimental setup as in Section III.

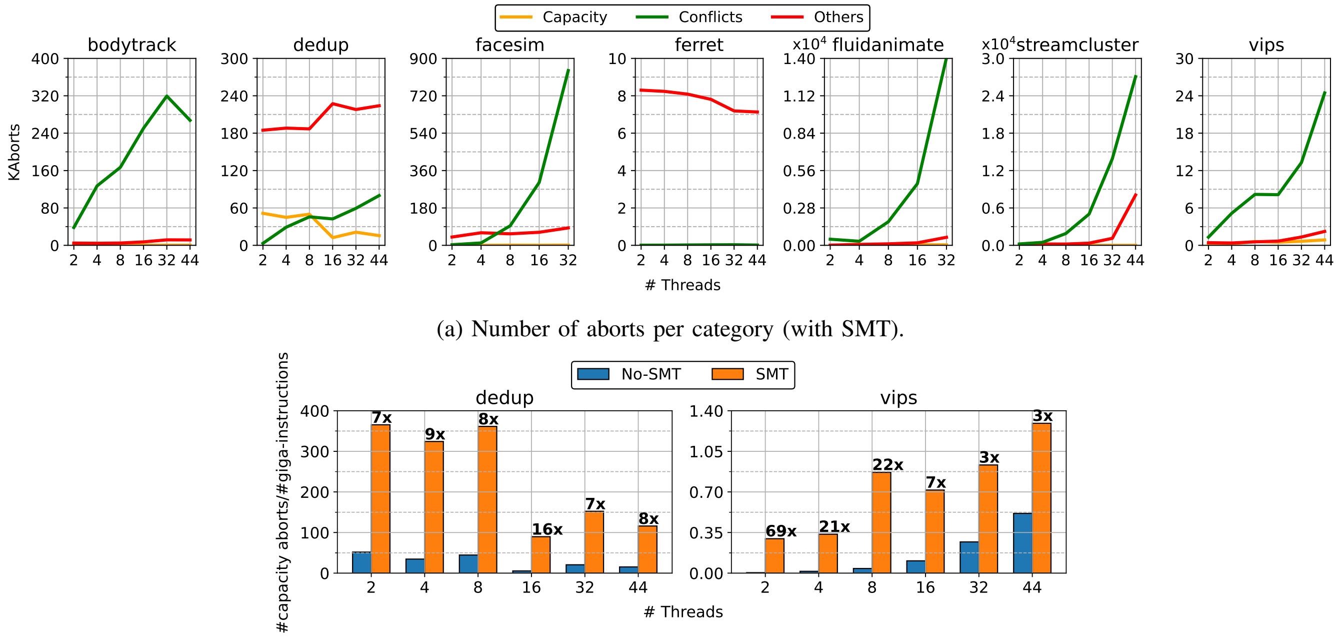
We select seven benchmarks from the PARSEC 3.0 suite to drive our experiments: `bodytrack`, `dedup`, `facesim`, `ferret`, `fluidanimate`, `streamcluster`, and `vips`. We choose this subset because the benchmarks use numerous lock-based synchronization points ($>$ 100) [42]. These locks are then being elided in the HTM version, providing us with sufficient information to analyze the HTM behavior. We exclude the `x264` benchmark due to runtime errors on our setup. All benchmarks are compiled with the recommended flags. We execute each benchmark three times using the native input files and take the median value.

Figure 4 shows the speed-ups achieved by our subset of benchmarks for different numbers of threads. We set the maximum value on the Y axis at 60 for all plots in order to facilitate a direct comparison between the graphs in the figure. All benchmarks are executed on up to 88 threads, except for `facesim` and `fluidanimate` which only allow powers of two as input for the number of threads. Overall, HTM performance is on par with highly-optimized fine-grained locking. Only three of the PARSEC 3.0 benchmarks in our subset show a non-negligible difference between the standard version (fine-grain locks) and the HTM version (lock-elision). More precisely, in `fluidanimate` fine-grain locks perform slightly better (around 2% better on 64 threads) than the HTM version. In contrast, the `vips` benchmark exhibits better speed-up (1% on 66 threads) when using HTM. `Streamcluster` shows a more acute difference in HTM's favor (7% better speed-up on 44 threads, going up to 71% on 88 threads).

V. SMT Impact on HTM Capacity Aborts

Most HTM implementations use the cache hierarchy to track their read and write sets. With SMT enabled, threads running on the same core share the private cache resources. When transactions are involved, this translates to a significant reduction in the working-set size limit of the transactions [9]. This is typically reflected by an increase in capacity aborts. Moreover, the impact is more prominent with the increasing number of hardware threads per core.

In this work, we aim to quantify the SMT impact on real-world applications. To this end, we measure the capacity aborts of PARSEC 3.0 benchmarks using lock-elision. As opposed to the applications in Section III, manually optimized with respect to the contents of the transactions, this analysis addresses transactional behavior in realistic scenarios, where



(a) Number of aborts per category (with SMT).

(b) Capacity aborts with and without SMT and their ratio.

Figure 5: SMT impact on HTM capacity aborts in the PARSEC 3.0 suite.

transactional contents cannot be controlled or planned. While the number of capacity and random aborts was negligible in the concurrent data structures execution, we expect an increase in the rate of these categories when running real applications. We start with the subset presented in Section IV. We use two SMT threads per core in our experiments. The applications are executed without and with SMT (we pin one and two threads per core, respectively). We analyze the distribution of aborts with the no-SMT and SMT versions.

A. *Evaluation*

Figure 5a shows the number of HTM aborts in the PARSEC 3.0 benchmarks. The abort reasons are split in three categories: capacity, conflict, and other. The latter contains explicit aborts and random aborts due to unfriendly instructions or OS interference. Most benchmarks are dominated by conflict aborts. Two applications show a significant amount of capacity aborts: `dedup`, with up to 21% and `vips`, with up to 11%, of all transactional aborts. These benchmarks have one particular property in common: they access large memory areas in the critical sections, i.e., inside transactions. More precisely, they process chunks of data for (de)compression and image transformation. In contrast, the other benchmarks in the suite employ HTM mainly for synchronizing access to shared flags and variables. As such, we will further focus on the subset of two applications that need large cache capacity to store their transactional working set.

Figure 5b shows the number of capacity aborts per giga-instruction for different numbers of threads, with and without SMT. We illustrate the ratio between the two versions (SMT and no-SMT) with numbers above the bars. When using SMT, both benchmarks show a considerable increase in capacity aborts regardless of thread count. For `dedup` the maximum impact is recorded on 16 threads, with 16x more capacity aborts when using SMT. `Vips` shows a dramatic 69x increase in capacity aborts on two threads for the SMT version. This ratio decreases with the increasing number of threads down to 3x. We believe this is due to increased thread contention that favors transactional conflict aborts before the tracking structures overflow.

In conclusion, we have observed that SMT can cause a significant increase in the number of aborted transactions due to lack of resources. This holds true even for carefully designed applications such as those of the PARSEC 3.0 suite. This issue can be much more critical in other areas. Wang et al. [9] have shown that for in-memory databases capacity aborts are a limiting factor.

B. *Transaction-Aware Replacement Algorithm*

We present *Transaction-Aware (TA)* replacement algorithm, a novel mechanism that mitigates the negative effects of SMT on capacity abort rate. We build our solution on top of the *Least Recently Used* (LRU) replacement algorithm and call it *TA-LRU*. As shown in Section V-A, SMT increases the number of capacity aborts. The main idea of TA-LRU is to protect the cache lines involved in transactions, and thus mitigate the SMT effect.

The classical LRU algorithm evicts the least recently used cache line when no more space is available in the cache. It does not take into account whether the selected cache line is used in a transaction. By contrast, our TA-LRU algorithm avoids evicting cache lines involved in a transaction unless it is strictly necessary. TA-LRU works as follows: let us assume that the cache contains lines L_1 to L_n, with L_1 being the most recently used and L_n, the least recently used. L_n is also used in a transaction. When a new line has to be brought into the cache, classical LRU would evict L_n, therefore causing a capacity abort for the running transaction. TA-LRU, on the other hand, will evict the least recently used line that is not used in a transaction, keeping L_n stored in the cache.

Our solution does not need any extra hardware. Typically, in HTM implementations, the cache lines are annotated with two bits to track read/write accesses. TA-LRU simply ORs the two bits. If the result of this operation is 0, i.e., none of the bits is set, it means that the current cache line is not involved in a transaction and can be evicted without any side effects on performance. If any of the bits is set, the algorithm moves to the next candidate in LRU order. Our solution can be integrated with any cache replacement algorithm.

While the TA-LRU selection process for an eviction candidate does not have any overhead compared to standard LRU, the algorithm may not be as effective at choosing the best victim. Some of the least-recently used entries will potentially be marked as transactional and avoided, leading to the eviction of more recent entries. In other words, some of the cache entries that may have been reused in the future with standard LRU, may be evicted with TA-LRU. In the worst case scenario, all lines can be involved in transactions, leading to the eviction of the least recently used transactional line and the subsequent abort of the corresponding transaction. We perform a preliminary overhead assessment in Section V-C and leave the complete evaluation on real-world applications as future work.

C. TA-LRU Prototype

We evaluate our proposal in gem5 [43] using the only HTM implementation currently available in the simulator, the Transactional Memory Extension (TME) for the ARM architecture. The incipient state of the TME implementation in gem5 significantly limits the performance of any executed transactional workload and hinders the evaluation. Presented with these limitations, we propose an initial TA-LRU prototype, with the goal of assessing its potential.

We implement and test our prototype in a one-core out-of-order superscalar ARM processor that can execute two threads simultaneously. We use the DerivO3CPU type. The threads share $32\,\mathrm{KiB}$ L1 data and instruction caches (8-way), a $1\,\mathrm{MiB}$ L2 cache (16-way), and a $2\,\mathrm{MiB}$ LLC (16-way). All cache levels are inclusive.

We evaluate our proposal on a synthetic microbenchmark designed to stress transactional capacity and generate overflows. The microbenchmark executes two concurrent threads that access two arrays: one smaller and, respectively, one larger in size than the L1 data cache. One thread starts a transaction, iterates over the smaller array, reading and updating the elements, and commits the transaction. The other thread simply iterates over the larger array, thus polluting the cache and affecting the working-set size limit of the transactional workload.

With TA-LRU we observe a 16x reduction in capacity aborts compared to classical LRU. More precisely, only 3% of the started transactions abort due to capacity overflow when using TA-LRU, as opposed to 53% for LRU. We further measure the performance in *cycles per instruction (CPI)*. The thread that executes the transactional workload shows 2% performance improvement with TA-LRU. We also look at the potential overhead of our solution, by measuring the slow-down of the non-transactional thread. We find that the performance loss is under 1%. Finally, we analyze the number of cache misses in L1, as indicator for the impact of TA-LRU on caching. We measure an overhead of 6.25% for TA-LRU.

VI. Conclusions and Future Work

This work presents an extensive study on the scalability of various synchronization mechanisms. It departs from the state-of-the-art by: including all major forms of synchronization, from typical locking schemes to emerging technologies like HTM; evaluating them in the context of many-core systems; and quantifying the impact of SMT on HTM performance.

For the scalability evaluation, we experiment with two concurrent data-structures, a hash-table and a binary search tree, and workloads with various degrees of contention. We find that, in terms of throughput and latency, STM is lagging behind because of its considerable instrumentation overhead. In contrast, HTM matches lock-free and fine-grained locking performance and scales better as the number of threads increases. Since our HTM-based implementation uses the simplest and most common version of a fallback path, relying on a global lock, these results represent a lower bound for HTM scalability. We note that careful planning of the fallback contents can substantially reduce transactional conflicts and boost the performance.

Going further, we compare the performance of fine-grain locks and HTM on a widely-used benchmark suite consisting of real-world scientific applications, namely PARSEC 3.0. At the same time, we make the case of HTM adoption, showing on the PARSEC benchmarks how easy it is to obtain comparable performance to that of a highly-optimized locking scheme by simply flipping the value of a flag in the glibc library.

Finally, we analyze the impact of SMT on HTM performance. We find that enabling SMT for applications that access large blocks of memory inside their critical sections, considerably affects HTM commit-rate. SMT reduces the available resources per transaction, resulting in repeated capacity overflow aborts. We propose *Transaction-Aware LRU (TA-LRU)*, a novel cache replacement algorithm that aims to mitigate the negative effects of SMT on transactional capacity abort rate. Our prototype reduces aborts in this category by a factor of 16.

As future work, we plan to evaluate TA-LRU on real-world benchmarks and to enhance this work with results on even higher core-count setups, employing various types of hardware, compilers and data-structure implementations. Another potential future direction is an extensive comparison of main synchronization mechanisms on many-core systems from an energy perspective.

Acknowledgments

This work was supported by Huawei Zürich RC Computer Systems Lab, MINECO/AEI/ERDF (grant PID2019-105660RB-C21 / AEI / 10.13039/501100011033), and Government of Aragon (T5820R research group).

References

[1] L. Gwennap, "AMD Rome ruins Intel hegemony," *Microprocessor Report*, 2019.

[2] L. Gwennap, "IBM Power10 triples efficiency," *Microprocessor Report*, 2020.

[3] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, 1993.

[4] A. Dragojević *et al.*, "Why STM can be more than a research toy," *Commun. ACM*, vol. 54, no. 4, pp. 70–77, 2011.

[5] C. Cascaval *et al.*, "Software transactional memory: Why is it only a research toy?," *Queue*, vol. 6, no. 5, pp. 46–58, 2008.

[6] N. Diegues *et al.*, "Virtues and limitations of commodity hardware transactional memory," in *Proc. 23rd Int. Conf. PACT*, pp. 3–14, ACM, 2014.

[7] G. Southern and J. Renau, "Analysis of PARSEC workload scalability," in *Proc. ISPASS*, pp. 133–142, IEEE, 2016.

[8] J. Park and W. Baek, "Quantifying the performance and energy-efficiency impact of hardware transactional memory on scientific applications on large-scale NUMA systems," in *Proc. IPDPS*, pp. 804–813, IEEE, 2018.

[9] Z. Wang *et al.*, "Using restricted transactional memory to build a scalable in-memory database," in *Proc. 9th EuroSys*, pp. 1–15, ACM, 2014.

[10] V. Pankratius and A.-R. Adl-Tabatabai, "A study of transactional memory vs. locks in practice," in *Proc. 23rd ACM SPAA*, pp. 43–52, ACM, 2011.

[11] A. Dragojević *et al.*, "Stretching transactional memory," *SIGPLAN Not.*, vol. 44, no. 6, pp. 155–165, 2009.

[12] P. Felber *et al.*, "Dynamic performance tuning of word-based software transactional memory," in *Proc. 13th ACM Symp. PPoPP*, pp. 237–246, ACM, 2008.

[13] P. Felber *et al.*, "Time-based software transactional memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 12, pp. 1793–1807, 2010.

[14] A. Natarajan and N. Mittal, "False conflict reduction in the Swiss Transactional Memory (SwissTM) system," in *Proc. Int. Symp. IPDPSW*, pp. 1–8, IEEE, 2010.

[15] D. Dice *et al.*, "Transactional locking II," in *Proc. Int. Symp. DISC*, pp. 194–208, Springer, 2006.

[16] R. Rajwar and J. R. Goodman, "Speculative lock elision: enabling highly concurrent multithreaded execution," in *Proc. 34th MICRO*, pp. 294–305, IEEE, 2001.

[17] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *Proc. 10th Int. Conf. ASPLOS*, pp. 5–17, ACM, 2002.

[18] K. E. Moore *et al.*, "LogTM: log-based transactional memory," in *Proc. 12th Int. Symp. HPCA*, pp. 254–265, IEEE, 2006.

[19] L. Yen *et al.*, "LogTM-SE: Decoupling hardware transactional memory from caches," in *Proc. 13th Int. Symp. HPCA*, pp. 261–272, IEEE, 2007.

[20] D. Dice *et al.*, "Early experience with a commercial hardware transactional memory implementation," *SIGARCH Comput. Archit. News*, vol. 37, no. 1, pp. 157–168, 2009.

[21] H. Q. Le *et al.*, "Transactional memory support in the IBM POWER8 processor," *IBM J. Res. Dev.*, vol. 59, no. 1, pp. 8:1–8:14, 2015.

[22] A. Wang *et al.*, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *Proc. 21st Int. Conf. PACT*, p. 127–136, ACM, 2012.

[23] C. Jacobi *et al.*, "Transactional memory architecture and implementation for IBM System Z," in *Proc. 45th Int. Symp. MICRO*, pp. 25–36, IEEE, 2012.

[24] W. Hasenplaugh *et al.*, "Quantifying the capacity limitations of hardware transactional memory," in *Proc. 7th WTTM*, 2015.

[25] I. Calciu *et al.*, "Improved single global lock fallback for best-effort hardware transactional memory," in *Proc. 9th Transact Workshop*, ACM, 2014.

[26] R. Quislant *et al.*, "Enhancing scalability in best-effort hardware transactional memory systems," *J. Parallel Distrib. Comput.*, vol. 104, no. C, pp. 73–87, 2017.

[27] H. Guiroux *et al.*, "Multicore locks: The case is not closed yet," in *Proc. Conf. Usenix ATC*, pp. 649–662, USENIX, 2016.

[28] T. M. Rico *et al.*, "Energy consumption and scalability evaluation for software transactional memory on a real computing environment," in *Proc. Int. Symp. SBAC-PADW*, pp. 7–12, IEEE, 2015.

[29] T. Brown *et al.*, "Investigating the performance of hardware transactions on a multi-socket machine," in *Proc. 28th ACM SPAA*, pp. 121–132, ACM, 2016.

[30] M. Schindewolf *et al.*, "What scientific applications can benefit from hardware transactional memory?," in *Proc. Int. Conf. SC*, pp. 1–11, IEEE, 2012.

[31] R. M. Yoo *et al.*, "Performance evaluation of Intel® transactional synchronization extensions for high-performance computing," in *Proc. SC*, pp. 1–11, ACM, 2013.

[32] T. Nakaike *et al.*, "Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8," in *Proc. 42nd ISCA*, pp. 144–157, ACM, 2015.

[33] Z. Cai *et al.*, "Understanding and utilizing hardware transactional memory capacity," in *Proc. ISMM*, ACM, 2021.

[34] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proc. 15th Int. Conf. DISC*, pp. 300–314, Springer, 2001.

[35] S. V. Howley and J. Jones, "A non-blocking internal binary search tree," in *Proc. 24th ACM SPAA*, pp. 161–171, ACM, 2012.

[36] S. Heller *et al.*, "A lazy concurrent list-based set algorithm," in *Proc. Int. Conf. OPODIS*, pp. 3–16, Springer, 2006.

[37] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*. May 2020.

[38] L. F. Bonnichsen *et al.*, "Hardware transactional memory optimization guidelines, applied to ordered maps," in *Trustcom/BigDataSE/ISPA*, pp. 124–131, IEEE, 2015.

[39] H. Guiroux *et al.*, "LiTL source code and data sets." https://github.com/multicore-locks, 2016.

[40] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[41] "Lock elision in the GNU C library." https://lwn.net/Articles/535519/, 2021.

[42] C. Bienia *et al.*, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. PACT*, pp. 72–81, ACM, 2008.

[43] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.