Multi-dimensional Self-Tuning in Transactional Memory

Ricardo Neves

IST, Avenida Rovisco Pais 1, 1049-001 Lisboa

Abstract. TM is a powerful abstraction that promises to drastically simplify parallel programming. On the other hand, the efficiency of existing TM implementations can be strongly affected by the characteristics of the workloads generated by TM applications. This has raised interest in designing self-tuning solutions that adapt specific building blocks of existing TM implementations

This document first overviews the state of the art in the area of TM, analyzing the key trade-offs that affect the performance of these systems. Next it discusses existing self-tuning solutions for TM, highlighting a critical limitation of existing approaches: these only support the dynamic adaptations of *individual* building blocks of a TM. This prevents existing solutions from pursuing globally optimal configurations.

In the light of this critical analysis of the state of the art, the document proposes the development of Green-TM, a library supporting the dynamic and possibly simultaneous adaptation of multiple building block-mechanisms of a TM.

Keywords: parallel computation, machine learning, transactional memory, adaptivity, self-tuning, thread-mapping, contention manager, locking, optimistic concurrency, hardware

1 Introduction

Over the last years, processors manufacturers faced a problem with increasing the clock speed of a single processor. As a result, parallel processor architectures have entered the realm of mainstream computing: nowadays commodity processors adopt a multi-core organization, according to which computing capacity is expanded by increasing the number of cores, rather than enhancing the processor's clock speed.

However, having more cores is not equivalent to having a single more powerful CPU. In fact, in order to take advantage of the computing capacity offered by multi-core architectures, application developers are required to embrace a major paradigm shift, moving from sequential to parallel programming. Unfortunately, parallel programming is notoriously more challenging. One well-known, critical problem of parallel programming is how to synchronize concurrent accesses to shared data. In fact, the conventional approach to synchronization is based on locking, which is well known to suffer of severe issues like lack of composability, deadlocks, and livelocks.

Given the huge relevance that parallel programming has gained over the last years, the academic and industrial research community have invested a great effort to identify a synchronization mechanism that could represent a simpler, yet efficient alternative to locking. Currently, Transactional Memory (TM) is probably the most prominent proposal in this sense.

TM brings the familiar abstraction of transactions, used for decades with success in the area of database systems to the domain of parallel programming. By requiring programmers to only specify what should be executed atomically, and not how atomicity should be achieved, TM can drastically simplify the problem of correctly synchronizing concurrent computations. Further, by leveraging on scalable and efficient concurrency control mechanisms, possibly accelerated via dedicated hardware supports, TM has been shown to deliver performance comparable, or even superior, to the one achievable by complex and error-prone fine-grained locking mechanisms.

On the down side of the coin, the design space of a TM implementation is huge, and existing research has shown that no-one-size-fits-all design exists that can deliver optimal performance for all possible workloads [6, 14, 15, 35, 36]. To further complicate the matter, algorithmic designs are not the only factor influencing TM performance. The correct tuning of the parameters of a TM algorithm, as well the characteristics of the hardware platform are known to have a strong impact on the efficiency of existing TM libraries.

These considerations motivate the research on self-tuning mechanisms capable of dynamically adapting the various internal mechanisms encompassed by TM systems, in order to match the characteristics of application's workloads as well as of the underlying hardware infrastructure.

This document is structured as follows: Section 2 overviews the state of the art in the TM field, including existing approaches for self-tuning TM systems. Section 3 presents the objectives that I intend to pursue during the next phase of my dissertation. Section 4 focuses on how I plan to evaluate the work carried out in the context of my dissertation. Finally Section 5 contains a brief summary of the phases of my work.

2 Related Work

This section discusses and details the study conducted on the TM field. It covers several topics ranging from TM implementations (software, hardware, hybrid) to self-tuning techniques (Adaptive Thread Mapping, Tuning of internal TM parameters, etc.).

More precisely, it contains a overview of Transactional Memory, Software Transactional Memory (STM) and state of the art implementations, Hardware Transactional Memory (HTM) and current HTM support libraries, Hybrid Transactional Memory (HyTM), and lastly self-tuning schemes.

2.1 Transactional Memory

Transactional Memory is a lightweight concurrency control mechanism that many researchers believe to be the path to follow to achieve performance comparable (and under some conditions better) to fine-grained locking, providing at the same time the ease of use of coarse-grained locking.

As mentioned before, with TM the programmer only needs to specify a critical zone of code as atomic, and the underlying implementation will take care of correctly synchronizing it, removing loads of complexity from the programmer's shoulders.

This mechanism is based on the concept of transactions, which was successfully introduced decades ago in the context of database systems. As in classic database transactions environments, TM transactions preserves atomicity; as such, a transaction either has no effect (i.e., it is aborted), or appears to take effect instantaneously in some point in time (serialization point) within its start and completion.

Transaction memory enables several transactions to run in parallel with or without locks. The detection of invalid operations that would break consistency can be performed either while a transaction is running (eager) or at commit-time (lazy). This ensures that a given transaction has a consistent view of the shared memory.

TM was initially proposed as an extension for multi-processors cache-coherency protocol [22]. However, the lack of architectural support for transactions led researchers to focus on software-based solutions to further investigate the potentiality of the approach. Moreover, hybrid solutions have been proposed, which try to reconcile the software and the hardware.

2.2 Software Transactional Memory

Software Transactional Memory is the most common implementation of TM, since its implementation is not bound to the availability of any architectural support [1, 10, 16, 13, 18, 19].

At their basis, STM implementations rely on special data structures that track what a transaction read and write (read and write set). These sets are checked (either eagerly or lazily) to detect conflict.

Throughout the years, many STMs have been proposed, with the purpose of improving its performance. From the base design choices exploited, it is possible to identify the following alternatives:

1. word-based vs object-based - granularity level at which memory is accessed. Word-based means that the memory is accessed at the granularity of machine words or larger chunks of memory. Object-based implies that accesses to memory are done at object granularity and it requires the TM to be aware of the object associated in every access.
2. lock-based vs lock-free - whether or not a TM resorts to locks to correctly handle concurrency.

3. write-back vs write-through - how updates are written to memory. Write-through writes the updates directly to memory and previous values are stored in an undo log, providing lower commit-time overhead than Write-back. On the other hand, Write-back writes updates to memory upon commit, and stores it in a write log, granting lower abort overhead than the latter.
4. encounter-time locking vs commit-time locking - when conflict detection is performed either at commit (lazy) or during execution (eager).

As it was demonstrated, there exists a several number of different design choices, and internal parameters configurations to apply, and these decisions have a huge impact on the TM performance.

The best design and configuration can vary according to the architecture, depending on the CPU and the size of the caches. More important to note is that the efficiency that will result from these choices will hugely vary depending on the workload generated by the application.

Most of the researchers agree that there is no STM implementation that can always deliver the best performance independently of the workload [8, 14, 16, 35].

In this subsection, three STM implementations will be analyzed, namely TinySTM, JVSTM, and NOrec. These STMs explore different possibilities of the design space for what concerns the aforementioned design choices. Such different design choices, as hinted in Section 1, result into these STMs being optimized for different kinds of workload: TinySTM provides high scalability on write-intensive workloads, NOrec was conceived to minimize the overhead at low thread counts, and JVSTM excels at long-running read-only transactions.

It will be given focus to the design choices made by the authors of each one, as well as the auxiliary mechanisms incorporated, which led to these different specializations.

TinySTM

TinySTM is a STM implementation presented by Pascal Felber, Christof Fetzer and Torvald Riegel in 2008 [16].

This TM is a word-based implementation that resorts to locks to protect shared memory locations, and to an encounter-time locking (eager) scheme to guarantee that any transaction always read consistent states. It also allows the co-existence of both write-through and write-back designs for memory access.

The authors justify the adoption of encounter-time locking with the two following considerations:

1. Their experimental observations seem to indicate that if the conflicts are detected early, then the transaction throughput will increase because there will not be useless work done by transactions.
2. It allows to handle read-after-writes efficiently without requiring expensive mechanisms. Encounter-time locking enables a transaction to temporarily acquire the locks corresponding to the memory locations that it accesses. In combination with write-through it assures that the memory always contains the latest value written.

The validation of transactions at commit-time is implemented using a Hierarchical Locking scheme. This strategy uses a shared array of l locks, and a smaller "hierarchical" array of $h \ll l$ counters. Choosing l as a multiple of h,

$$l = 2^i, h = 2^j, i > j \tag{1}$$

it is possible to compute the index of the lock and the counter, for a given address:

$$lockIndex(addr) = hash(addr) mod(l) \tag{2}$$

$$counterIndex(addr) = hash(addr) mod(h) \tag{3}$$

This scheme allows transactions to determine whether locks have been acquired, and can be generalized to multiple levels of nesting.

TinySTM's internals are optimized for workload that exhibit the following characteristics:

1. update transactions read many locations.
2. few writes from concurrent transactions.

The authors tested, and believe that these conditions are often found in real applications, to consider it a useful strategy.

Finally, TinySTM implements self-tuning strategies to automatically adjust the values for this STM internal parameters. This aspect will be covered in the Self-Tuning Section.

JVSTM

Java Versioned Software Transactional Memory is a Java STM implementation (unlike most of STM on C) presented by João Cachopo, and António Rito Silva in 2006 [1].

It is actually used in a production environment on the IST's FénixEDU project as a replacement of the previous lock-based concurrency control mechanism. JVSTM ensures that object cache maintained by the middle-tier servers are consistently and atomically updated with the back-end database [4, 27].

The original version of this STM used commit-time locking with a single global lock, nested transactions, and versioned boxes.

A nested transaction is a transaction (child) that starts within the context of another transaction (parent). A versioned box is a container that keeps a sequence of values, and can be seen as a replacement for memory locations, and transactional variables.

It was the first known STM in literature to guarantee that read-only transactions would never conflict with concurrent ones, which improved the concurrency on applications consisting of long transactions containing considerably more reads than writes.

To minimize the overhead of running applications with a large fraction of read-only transactions (common scenario in many workloads) [2], JVSTM's default configuration speculates that any starting transaction is going to be read-only. This avoids tracking the transaction's read set and improve performance in

case the speculation is correct. If the transaction is detected to not be read-only, it is re-executed.

JVSTM employs two techniques to further reduce conflicts:

1. Delaying computations - avoids high-contention boxes, and re-executes the parts of the transaction that caused the conflicts. To effectively use this technique, it uses boxes that hold values private to a transaction.
2. Restartable transactions - allows the parts that caused the conflict to be re-executed.

Like stated before, the original version of JVSTM [1] used a global lock to handle mutual exclusion at commit-time. Later in 2011, Sérgio Miguel Fernandes and João Cachopo published a second, enhanced, implementation of JVSTM, which adopts a lock-free scheme [18].

The change was motivated for some reasons, e.g.: Poor performance in applications containing transactions that execute several writes in a short period. Thus, high contention on the global lock will significantly degrade performance.

The new commit algorithm proposed uses some concepts of the old one, e.g., read-set validation, write-back, commit visible to other transactions. While the original lock-based used snapshot validation as a validation technique, this new version uses incremental validation. The latter, consists in checking every write-set committed since the transaction started, in search for an intersection with its read-set.

The size of the write-set is application dependent, hence the list to iterate at validation depends on the number of write transactions, which hinders the incremental validation performance. To minimize this issue, Fernandes et al. [18] introduced a small modification to the validation procedure combining snapshot with incremental.

Before any validation, a transaction helps to write-back all pending commits already in a queue, returning the last record that it helped to commit, which is assigned to a variable named *lastSeenCommited*. After that, a snapshot validation is performed (without synchronization, being possible that concurrent commits occur). However, the set of committed versions for each transactional location contains at least all the commits up to the version of the *lastSeenCommited* record, performing validation up to that point.

One last validation still needs to be performed, checking for any newer commits, to grant a valid commit. This will be done through incremental validation, but only from the *lastSeenCommitted*.

This algorithm is slower for low transaction counts, but scales better because it keeps the list of write-sets small.

NOrec

No ownership records is an STM library presented in 2010, by Luke Dalessandro, Michael F. Spear, and Michel L. Scott [10].

NOrec is mostly known for being a TM that incorporates some desirable features (e.g., low fast-path latency, publication and privatization safety, livelock freedom, etc.).

NOrec is specifically designed to incur very low overhead at low thread counts, at the cost of exhibiting limited scalability. Its design is based on a low overhead STM algorithm, presented by Spear et al. named TML [33].

TML uses a single global sequence lock, which has the advantage (over traditional reader-writer lock) that readers are invisible, and induce the coherence overhead of updating the lock. It uses eager conflict detection and in-place updates, where writes acquire the lock for writing, and reads check the lock version to ensure consistency.

TML scales poorly, since it uses a single global lock with eager conflict detection, meaning that only one writer can be active at a time.

NOrec design diverges from TML, in the sense that it tends to improve scalability over the latter. While TML is eager, NOrec uses a write-back strategy to write updates to memory, and performs conflict detection at commit-time.

The lazy conflict detection for concurrent speculative writers, enables to minimize the quantity of time that a writer holds the lock. Such, increases the probability that concurrent read-only transactions will commit. Reads are validated eagerly, in order to avoid a transaction to read inconsistent values and prevent erroneous behavior in transactions that are destined to abort.

Privatization problem [32] arises when an object is accessed by transactional and nontransactional code at the same time (e.g., private to a nontransactional thread), i.e., TM systems must avoid violation of atomicity, consistency and isolation in such scenarios to assure safety.

The serialization of writeback ensures privatization safety [32], and combined with value-based validation, ensures publication safety[24]. This allows for accessing shared data structures outside transactions, with no additional cost, as long as the program is transactional data race free.

It does also support *closed nesting* [26], enabling to abort an inner transaction without making the outer parent to immediately abort as well.

This STM also has high compatibility to legacy systems, and the authors think that would be a viable choice to use as a fallback software support for hardware, or hybrid TMs.

2.3 HTM

Hardware Transactional Memory is a more recent bet to build a superior TM.

In HTM, transactions are executed with the aid of specific architectural support and processor instructions.

Although the first TM proposal was hardware, the absence of HTM support in commercial processors, limited most of researchers to emulators to test their ideas, leading to a major focus on STM to advance the state of art.

Recently, both Intel and IBM[1] implemented support for HTM on some of their new processors. For this reason, real-world HTM implementations are garnering much attention in the last years.

Like the Herlihy and Moss's original TM proposal [22], these TMs rely on modified cache coherence protocols in order to achieve atomicity and isolation.

The main advantage of HTM over STM is that it requires no instrumentation comparing to STM solutions. Since it does not require to instrument reads and writes, it avoids most of the overheads incurred by STMs that could severely degrade the performance. On the other hand, a major drawback of HTM lies in the fact that it cannot guarantee that a transaction will ever succeed (even without concurrency) due to its limited nature, which has to keep the read-write set into l1 cache.

Taking this into account, one of the major concerns is to develop an efficient software fallback mechanism as an alternative synchronization mechanism to use whenever HTM restrictions may prevent transactions from ever committing.

Intel acknowledged that programmers must provide this software fallback path on the begin instruction, deciding in this way what should be done upon abort.

Example of restrictions are: long transactions, which contains lots of operations and can exceed the size of the l1 cache; interrupts; and context switches.

On the rest of this section, the two topics below will be addressed:

1. TSX
2. BlueGene/Q

It will be presented a brief overview of both HTM implementations (Intel TSX, IBM BlueGene/Q), and results of some evaluation done so far.

Intel TSX

Intel started including supports for HTM with the release of their Haswell (4rd gen.) processors, in 2012, and nowadays it is deployed on several machines, from tablets to servers.

The hardware support was made available by extending the instruction set for x86 with Transactional Synchronization Extensions (TSX), which is the first generation of Intel's commodity HTM. TSX provides two interfaces: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM).

HLE allows optimistic execution of a critical section through eliding the acquision of a lock when writing, so it appears as free to other threads.

TSX supports HLE by the use of two operates (XAQUIRE and XRELEASE) that can be placed in LOCK instructions. In Haswell processors those allow to elide the lock, assuring that it will be read but not written, enabling concurrent threads to execute the same section simultaneously. Data races are prevented

[1] AMD also planned a few years ago to build a next generation processor with support for HTM [12], but until now there have not been any more updates pointing towards a near future release.

by keeping track of the speculative accesses, rolling back the execution if the accesses made are invalidated by cache coherency. In such case, the thread re-execute the section but this time the lock is acquired and released normally. Concurrent elisions of the same lock are aborted, since the lock is part of their transactional footprint.

RTM offers more flexibility, offering instructions such as $XBEGIN$ and $XEND$, that map directly to the usual construction beginning and committing of a transaction. Another advantage is that it allows the application of more fallback strategies upon abort of the hardware transaction, unlike HLE that gives up immediately. On the other hand, determining the best fallback strategy is not an easy task, as it ultimately depends on the workload.

The provided hardware support is best-effort nature, and, due to inherent HTM limitations, it cannot guarantee if a transaction will ever be able to commit, not ensuring progress guarantees.

Intensive evaluations done on Intel TSX were recently published by Diegues et al. [15]. The evaluation was done comparing this HTM to several state of arts STMs, and coarse/fine-grained locking schemes. Amongst other conclusions, the study demonstrates that TSX can clearly outperform any synchronization scheme in some workloads (e.g., short and infrequent transactions) but performs poorly in others (e.g., long and contention-prone transactions, where STM and fine-grained locking perform better).

IBM Blue Gene/Q

Blue Gene/Q is one of the first implementations of HTM for commercial processors, and it was made available by IBM.

Being a HTM it partially solves some of the main problems of STM solutions, such as: the high overheads from starting and committing a transaction, and the instrument and monitoring of memory references inside a transaction. But like expected from its nature, it does not ensure forward progress, and like Intel TSX it can benefit from software support to improve its performance.

The BG/Q provides the following to support transactional execution:

1. Buffering of speculative states - writes made during a transaction execution store a speculative state, that is buffered on the L2 cache, and becomes visible to the other threads after commit, in an atomic way.
2. Hardware conflict detection - hardware detects write-read, read-write, and write-write between concurrent transactions, or when a transaction is followed by a non-transactional access to the same address.

Hardware support is primarily implemented on the L2 cache that acts as a point of coherence, and efficiently supports TM enabling a considerably large speculative state.

The HTM alone shows good results on several benchmarks [34]. Unfortunately, there are no extensive works that compare Blue Gene with STMs. However, due to its HTM best effort nature, it is likely to exhibit limitations and

trade-offs similar to the TSX case. Like the Intel TSX an efficient software support is desirable to assure forward progress.

2.4 HyTM

Hybrid Transactional Memory is a TM that enables both hardware and software transactions to be executed in conjunction within the same TM.

After the first real HTMs like TSX and BlueGene/Q were released, an augmented interested on HyTMs solutions emerged. Due to the best-effort nature of HTMs, it is desirable to find a scalable STM to be used in the fallback path to ensure progress, instead of coarse lock-based schemes.

HyTM schemes aim to deliver the best of both TM types: HTM's efficiency and modern STM's scalability. Some TM researchers argue that HyTM has plenty of potential to be exploit [3, 9, 15, 23], but there is still lots of work to be done.

One of the latest results worth to mention on this field is Invyswell, a HyTM released this year, which will be addressed in detail on this section.

Invyswell

Invyswell is a recent published HyTM presented by Calciu et al., in 2014 [3]. This HyTM is a scalable hybrid implementation that uses a HTM along with a STM fallback, to guarantee forward progress.

It uses hardware transactions from Intel RTM, and software transactions from a modified version of an STM called InvalSTM [19]. Like any HyTM, Invyswell pursues the objective of enabling both hardware and software transactions to be concurrently executed, with the objective of delivering a solution that is effective for all transactions sizes and contention levels.

The STM used provides scalability and performance for large transactions with considerable contention, and has a distinctive difference compared to other STMs, i.e., it performs commit-time invalidation [19].

Commit-time invalidation allows the STM to have full knowledge of all conflicts between a committing transaction and others in-flight transactions, allowing it to make decisions on how to best mitigate contention. This design choice is extremely useful on HyTMs as it allows to mitigate the conflicts between software and hardware transactions. Another important aspect, is that it allows read-only transactions to commit without incurring serialization overhead, being transparent to RTM's faster executing hardware transactions.

This STM aims at optimizing performance against large transactions workload, complementing RTM, which excels with short transactions. Regarding this interplay, it is important to note that the STM chosen, does not serialize read-only transactions, enabling RTM to execute without interfering with the STM when the latter executes read-only transactions.

In this implementation, Haswell's hardware transactions are instrumented to track their read and write set via Bloom Filters. InvalSTM can then leverage

on the information maintained by these bloom filters for detecting conflicts with concurrently executing hardware transactions.

To manage the shared-memory between HTM and STM, Invyswell performs the conflict detection between a hardware and a software transaction after the hardware transaction has committed. This is unavoidable, since any write operation performed by an hardware transaction is buffered until commit time (TSX does not support non-transactional operations from within a transactional context). RTM does not support escape actions, hence when a hardware transaction conflicts with a software one, it aborts. By combining invalidation and conflict detection after a hardware transaction, this scheme minimizes the chance to abort a hardware transaction due to an in-flight software one.

The need for assuring progress guarantees, and adaptability to different heterogeneous workloads, lead Invyswell to support five transactions types:

1. SpecSW - software speculative transaction that uses private Bloom filters to track accessed memory locations. Performs invalidation after committing, which makes it compatible with hardware transactions that can invalidate in-flight SpecSWs.
2. BFHW - hardware bloom filter-based transaction. Used to handle conflict with software transactions, by preventing software transactions (SpecSWs) from committing or reading, while is still committing. After commit, it performs post-commit invalidation, marking as invalid all in-flight SpecSWs.
3. LiteHW - lightweight hardware transaction executed without read or write annotations. Can only commit if there are no in-flight software at the beginning of their commit-phase. Optimal for small transactions.
4. IrrevocSW - direct update software transaction, which acquires the commit lock as soon as it execution begins. Used to assure progress guarantees.
5. SglSW - direct update software transaction that does not allow concurrent execution of other software transactions. Used for small transaction that contain instructions not supported by RTM, assuring progress guarantees.

The transactions are scheduled in a performance descending order: first the high-risk hardware transactions, then the low-risk software transactions. The transitions between the types are decided at runtime, based on an application independent heuristic.

Finally, Invyswell relies on a Contention Manager that can be used by software transactions, which decides how to handle conflicts based on information such as the priority of each conflicting transactions and the size of their read and write sets.

The experimentation done with Invyswell shows that this HyTM delivers a better performance on a range of STAMP benchmarks [25] than the state of the art NOrec[10], and the authors believe that the hybrid mechanisms can become a first choice when Haswell platforms with more cores are released.

2.5 Self-Tuning

The success of a transactional memory largely depends on the fact that it greatly simplifies the task of writing and maintaining parallel applications. However, as

previously shown, the performance of a TM strongly depends on the affinity between its design and the workload characteristics.

Taking into account such considerations, self-tuning is needed to deliver optimal performance across all workloads, by reconfiguring the TM at runtime depending on the currently workload characteristics. This mechanism, in an ideal scenario, can permit the TM system to predict changes, adapting the system in a dynamic way to the configuration that will provide the best performance.

A self-tuning scheme needs to answer the following:

1. When to adapt?
2. How to adapt?

Regarding when to trigger the adaptation, the self-tuning mechanism can either be designed to react to workload changes (reactive), or to try anticipate them (proactive).

Reactive schemes determine the need for reconfiguration based on the recent observations of the workload. Proactive schemes try to anticipate the need for reconfiguration by predicting future workloads, and their effectiveness highly depends on the precision of the mechanisms chosen to predict these workloads.

Upon detection or prediction of the workloads changes, the self-tuning mechanism has to decide how to adapt to this change, i.e. which adaptation should be triggered, if any. The proper identification of the optimal configuration for a given workload is performed by the use of performance models, which allows the prediction of the system's performance in the available configurations to trigger.

Couceiro et al. [6] recently classified the performance models that can be used in self-tuning systems, in three categories:

1. White-box modelling
2. Black-box modelling
3. Grey-box modelling

White-box modelling leverages on the available knowledge of the systems and applications, and codifies it into a model (e.g., analytical or simulative), to capture how the system configuration and workload characteristics translate to performance. These models typically do not require training, still can benefit from a minimal sampling phase to get value for some parameters. The drawback of white box is that it relies on approximations, and assumptions, that may hinder the performance in certain situations [6].

Black-box modelling does not require any prior knowledge on the internal dynamics of the target system, and it relies on a training phase, observing the inputs (e.g., workload characteristics) and outputs (e.g., KPIs like throughput or energy consumption) of a system, with the objective to infer a statistical model (e.g., based on ML techniques) that captures the observed relations between inputs and outputs. It relies in inferring the performance function by observing the output (e.g., throughput, read/writes ratio) corresponding to an input (e.g., workload with small read-only transactions, using a configuration for HTM to retry 5 times). In comparison to White-box, this modelling provides adaptability

(e.g., initially unknown workloads), and tends to achieve a better precision, but could require a big amount of samples to build an efficient performance function.

Black-box can be further divided into off-line and on-line approaches. Offline approaches rely on a controlled training phase, during which the system experiments with different input/outputs to infer the performance function. Online approaches, instead, aim at finding a configuration that maximizes performance by exploring different configurations at runtime.

Grey-box modelling is a hybrid approach that uses both white and black modelling, attempting to benefit from the best that the two can offer: minimal training (White-box), and enhanced accuracy via retraining (Black-box).

With the available knowledge, the self-tuning implementation has to decide what should be the best decision for a system, for it to get the best possible performance. The literature on the self-tuning TM has explored a wide number of alternative techniques that are heterogeneous both in the employed modelling technique and in the target self-tuned aspect (e.g., TM parameters, TM switching, Thread Mapping Strategy, Contention Manager).

This section will consist of an overview of some of these self-tuning schemes: TM internal parameters, Adaptive thread mapping, Contention Manager adaptation, TM switching. Lastly I will describe some honorable mention self-tuning techniques for Distributed TM, which have some aspects that will used on the proposed solution (Section 3).

TM parameters

Research on TM has shown that a variety of parameters have a huge impact in the delivered performance. The tuning of the internal parameters of a TM has been subject of study by various works [14, 16, 29, 30]. Again, the dynamic change of these parameters is motivated by the acknowledge of the TM research community that there is no configuration that can fit better than all the others independently of the workload characteristics.

An example of a mechanism that focus on tuning the retry policy of an HTM is Tuner. Tuner is a self-tuning mechanism for Intel TSX, presented by Nuno Diegues and Paolo Romano, in 2014 [14].

It is an innovative approach that makes use of learning techniques, and profiling at run-time in order to identify the optimal TSX configuration in a workload.

Diegues and Romano show that properly handling HTM retry parameter, can provide a huge boost on performance.

Tuner deals with optimizing the retry strategy of TSX by tuning two internal HTM parameters:

1. Number of retries before giving up to software
2. How to spend the number of retries

Regard point one, it has been recently reported that 5 appears to be the best all round value for the maximum number of attempts in hardware [14, 15, 36]. Still

static configuration delivers suboptimal performance, due to the heterogeneity of the workloads generated by applications.

As for point two, the budget of retries can decrease linearly, divide by two, or set immediately to zero upon detecting a capacity abort in a transaction.

An important feature, is that it enables to individual tune the parameters of each application's atomic block, instead of using a single global configuration. This feature is relevant in programs that contain transactions with different characteristics, leading to heterogeneous workloads, benefiting from different configurations.

The final solution applied Upper Confidence Bounds (UCB) as a learning technique, considering each atomic block as a UCB instance. This technique estimates which one, among the available options for a given parameter, has the highest reward, and its used to optimize the consumption of the attempts upon capacity aborts.

To optimize the configuration of the number of attempts for each atomic block, Gradient Descent Exploration (GRAD) is used, which is an exploration technique similar to hill climbing.

These two techniques work in conjunction in the final algorithm, with a hierarchy between the two, so they don't overlap, allowing UCB to force GRAD to just explore in another direction, or avoiding "ping-pong" optimizations between the two [14]. The authors acknowledged that the joint strategy provided results that were always better than using only one of the approaches alone.

Still, in the context of self-tuning TM parameters, TinySTM has a scheme to tune the lock granularity.

This STM, which was already discussed at the STM Section, has a self-tuning mechanism aimed at optimizing the following parameters:

1. The hash function to map a memory location to a lock.
2. Number of locks.
3. Size of the array for the hierarchical locking.

The employed strategy keeps the throughput corresponding to each tested configuration, where a configuration is a triple formed by number of shifts, the number of locks, and the size of the hierarchical locking array.

It relies on a multi-dimensional hill climbing algorithm with memory, forbidden areas, and eight possible moves, as so, it works by making a move and then verifying its effectiveness during the next period.

Thread-Mapping

Multicore processors usually possess complex memory hierarchies of different levels of cache to reduce accesses to main memory. However, this can increase the time to access memory and degrade bandwidth usage if threads are not correctly placed on cores.

Thread-Mapping problem consist into allocating a thread onto the core that minimizes memory latency, and reduces memory contention through data locality.

Devising an optimal thread mapping strategy for a TM application is cumbersome because of two main causes:

1. TM applications are characterized by irregular behaviors, given by phased workloads and by its speculative nature.
2. Performance of a thread mapping strategy for a TM application not only depends on the target application, but also on the underlying TM backend.

One of the latest works on this matter focused on TM, was made by Castro et al.[5]. This study proposes four adaptive thread mapping strategies: two of which do not require any prior knowledge from TM applications.

In order to increase performance, the adaptive strategies must select the appropriate mapping for different TM applications, and STM configurations. The decision of the thread mapping is dynamic, since the workload may change during the execution. The adaptive thread mapping strategies require different information from the workload, to decide the mapping to apply, and are divided in two categories:

1. Single metric-based
2. ML-based

Like the name suggests, the single metric based approach relies on monitoring at runtime a single metric to characterize the behavior of a workload and to perform the self-tuning choice. For this, the authors propose two adaptive thread mapping strategies making use of two reference metric: conflict level (Conflict), and the execution time (Test).

The ML-based approach, instead, leverages on an offline trained ML algorithm to drive the self-tuning process. The authors applied two ML algorithms to develop the following two ML-based strategies: ID3, i.e., a Decision Tree, and Apriori algorithm, i.e., an association rule learner.

Concluding this topic, this recent paper [5] shows that the performance of TM applications can be improved by better exploiting the memory hierarchy of multicores, through the use of Thread-Mapping, presenting different adaptive strategies to find the best thread mapping for TM applications. An important note, is that during the evaluation process, the authors found that on average ML-based adaptive strategies deliver a better performance than single metric-based ones.

Contention Manager

Most of STM systems guarantee weak progress property, named obstruction-freedom, which, consists in assuming that a transaction that runs for a long period without overlapping with any transaction will eventually commit [2, 20, 32].

Since this property does not exclude livelock or starvation, stronger properties can be provided by a module called contention manager (CM).

A CM is a mechanism that resolves conflicts between transactions, by deciding which transaction to abort and when and how to reschedule the execution of the aborted transaction. The CM affect liveness, not safety, and can be evaluated by the number of transactions committed per time.

Throughout the years, many different contention managers have been proposed [10, 3, 21, 31]. Evidence suggest that no CM outperforms all the others for every workload.

On the lights of this consideration, Guerraoui et al. proposed a self-tuning mechanism for TM, named Polymorphic Contention Manager [20], which, allows transactions to have different CMs, in a tentative to provide the CM that provides the best throughput for a given transaction according to the situation. After an extensive experimentation, the authors of this study, concluded that there is no CM that performs best independent of the circumstances.

Polymorphic CM is a module that allows not only to switch CMs across workloads, but also across concurrent transactions in a single workload, and even between phases of a single transaction.

One of the problems of mixing CMs is how can CMs of different classes interact in a useful way. A hierarchy of contention manager classes was implemented, taking into consideration the cost associated to each CM class, and generalizing groups of CM classes. The cost associated has to do, with the quantity of information that a CM keeps, being the less costly, those who do not keep any information, and the most costly those that maintain a lot of information about the current transaction, other transactions, etc.

A last note on the topic, is that this study goes one step further, by allowing nested transactions, and allowing different CMs for these.

TM Backend

As already discussed, the design of each TM implementation is optimized for a target workload. As such, there is no single TM implementation that delivers the best performance across all possible workloads. For this reason, there is a recent proposal by Wang et al. [35] that implements methods to construct policies with the available knowledge and dynamically choose the most appropriate TM that maximizes performance.

To select the best STM for a workload, this work relies on information gathered through static analysis and dynamic measurement. The adaptivity policy framework can be activated during program execution by four events:

1. number of consecutive aborts exceeds threshold,
2. long delays when attempting to begin a transaction,
3. thread creation and destruction,
4. commit rate below defined threshold.

The profiling triggered by the events above, uses a simple custom STM named by the authors as ProfileTM [35] to sample per-transaction characteristics.

When one of these conditions is met, a profiling phase is performed. The profiling process is the following: the library blocks new transactions, and waits for all in-flight transactions to either commit or abort. Then the TM is switched to ProfileTM, and N transactions are ran, one at a time. After profiling, the system changes the TM to the one recommended.

If the recommended is the same as the current active TM library, the system stores the total number of commits and aborts, until the next trigger of the framework. By storing this information, if the same TM is chosen again, then it will only remain if there has been forward progress, and the abort limit for causing another trigger will be doubled.

The adaptivity policies for selecting the most suitable algorithm, can either be created by a programmer, generated by an ML system, or by a collaborative process between the programmer and the learning tool.

The authors divided these policies in two general types: Expert policies and ML-based policies.

Expert policies are written by a programmer to satisfy some requirements, like for example the intuition that the best algorithm depends on the maximum number of active threads.

ML-based policies are automatically created through machine learning techniques.

This is the first paper in literature to present an ML-based adaptivity system for synchronizing TM programs. The experimentation results showed that ML-based adaptivity offers great performance, maintainability, and flexibility, making TM switching using policies automatically generated by ML, an attractive approach to maximize performance of TM applications, despite heterogeneity and complexity of those.

Although, this work only supports switching between STMs. The proposed solution (Section 3) takes a leap further by aiming to support HTM too, since current HTMs outperform state of art STMs in some workloads [15, 36].

Still in the topic of TM Switching, to the issue of how to perform the switch, Lev et al. presented PhTM [23].

The main idea of this mechanism relates to being not safe to change modes without waiting for some conditions to be verified, i.e., in some cases it is better to delay changes. The delay allows to avoid trashing phenomena in which the TM is continuously switching between modes, resulting into poor performance.

The changing of TM backend (mode) is based on a single global variable named *modeIndicator* that contains 6 fields, being the most relevant:

1. current mode
2. number of transaction that must complete before switching
3. next mode
4. number of transactions that will be switched to the next mode

In certain modes, some transactions cannot be completed, e.g., functionality is not supported. To tackle this problem, a thread joining a mode can access the *modeIndicator*, changing the needed field, to assure correctness.

Distributed TM self-tuning

To conclude the topic of Self-tuning, I will briefly present two more works, morphR [8] and polycert [7].

Despite being focused on Distributed TM, they present interesting concepts, which will be applied on the system to develop (Section 3).

MorphR is a framework that supports generic adaptations, and introduces the concepts of fast-switching vs stop-and-go [8]. This can be extremely useful in a system composed by multiple reconfigurable components, where the reconfiguration process is a major part of the overall performance of the solution.

Fast-switching transitions are non-blocking and can be used when there is knowledge that both states can safely coexist, allowing for a faster reconfiguration process.

On the other hand, Polycert supports the coexistence of multiple protocol schemes of the same family [7].

Similar to what I will develop, this system relies on Oracles, which use forecasting in this case to determine the optimal certification scheme. The protocols used coexist naturally, since all of them rely on a common phase, during which they establish global serialization. Each message is tagged with a label that specifies which is being used for each transaction.

It is possible to create an analogy between tagged messages, and a possibility for tagged transactions, allowing for different TMs to coexist, something that could be explored at the proposed solution (Section 3).

Still, the more interesting aspect is the knowledge that grants that they can coexist, which applied with Morphr fast-switching, can enhance the process of reconfiguration.

These concepts will be taking into high consideration on the proposed solution, which will be described in detail on the next section.

3 Proposed Solution

After performing a survey on TM's state of the art, it is safe to consider self-tuning TMs as an appealing option towards building a superior TM. Still, the state of the art lacks in some interesting aspects, e.g.:

1. No solution is capable of encompassing simultaneous tuning of multiple configuration parameters,
2. There is almost no work focused on tuning HTM [14, 29],
3. Only a minority of the solutions focus on energy efficiency (most consider solely performance).

Taking the above into consideration, the objective of this work is to build a TM library, named Green-TM, which has the capacity of monitoring and controlling a set of dynamically tunable TM-related components, including:

1. TM back-end (HTM vs STM vs HyTM)

2. Parameters of the HTM retry policy
3. STM algorithm
4. Parameters of the STM algorithm
5. Number of concurrently active threads
6. Contention manager scheme
7. Thread Mapping strategy

One of the desirable features of this library is that it will be extensible, enabling a way to easily add/remove components, providing high modularity and flexibility. In order to achieve this result, it will have a set of well-defined interfaces, which will have to be implemented by the various building blocks that compose a TM.

These interfaces shall allow the online monitoring and gathering of a set of performance-oriented metrics, as well as the dynamic adjustment of the mechanisms and parameters employed by the various components of the TM library. In order to enable the dynamic adaptation of complex mechanisms, like concurrency control algorithms, in an efficient way, the interfaces defined by Green-TM shall support both blocking transitions (e.g., when the initial and final configuration cannot simultaneously coexist, like in the PhTM approach [23]), as well as non-blocking transitions.

The library to develop shall not only define the mentioned interfaces, but also employ mechanisms, which can guarantee the correct and efficient orchestration of the transitions, by exploiting such interfaces. The support of multi-dimensional adaptations in this TM will be based on the work presented by Couceiro et al. [7], which focus on Protocol adaptation for distributed TM platforms. The main idea is to adopt and extend this approach, to the case of multi-dimensional adaptations in non-distributed shared-memory TM. Such an extension is essential, since a major goal, and unique characteristic, of Green-TM is precisely to be able to change multiple parameters at the same time, without endangering correctness.

Green-TM will support the modular interaction with external predictors, which will be responsible of determining *when* to trigger an adaptation, and *which* adaptation to trigger. To this end, Green-TM will expose well-defined interfaces, which will expose information regarding key performance indicators (KPIs), the current configuration of the TM library, and optionally metrics related to the workload characteristics (e.g., transaction duration, read/write ratio).

The system will encapsulate the prediction functionalities by means of an Oracle abstraction, which can be seen as an abstract component that relies on statistical knowledge to predict performance gains under a certain configuration. This component will rely on various ML-based techniques, which are being developed in parallel by other researchers at the Distributed Systems Group of INESC-ID.

In the remainder of this section, I will provide an overview on the envisioned architecture of Green-TM, of the mechanisms supporting dynamic adaptation, and of the work planned for the next semester.

3.1 Architecture

The proposed system architecture is illustrated in Fig.1, which depicts its main entities and the relations between them. The main modules are the Green-TM, the Reconfiguration Manager, and the Oracle.

Green-TM shall define two main APIs: the Monitoring API and the Switching API.

The Monitoring API will allow externalizing a set of KPIs (e.g., throughput), the current configuration of the TM library, and, optionally, metrics characterizing the current application's workload (e.g., read/write ratio). This API will be used to convey this set of information to the Reconfiguration Manager.

The Reconfiguration Manager has the role of coordinating the reconfigurations. It takes as input the new reconfiguration to apply, and decides in which order will be applied, based on the relations among the involved reconfigured components and the efficiency of the reconfiguration. A major challenge will be on how to extend the approach in MorphR [8] to handle multi-dimensional adaptations.

In order to uniformise the interactions between the Reconfiguration Manager and the TM, the plan is to define an additional API, called Switching API, which allows the Reconfiguration Manager to specify which configuration(s) should be adapted, while encapsulating the mechanisms necessary to achieve a correct and efficient transition.

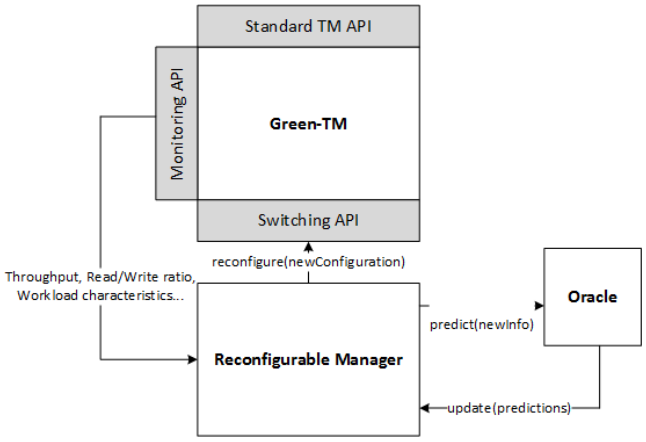

Fig. 1. Green-TM base structure.

Green-TM's internal structure is shown in Fig.2. At the lowest layer, the TM contains multiple tunable TM back-ends (HTM, STM), as well as different STM implementations. Besides the TM concurrency control mechanisms, Green-TM will also support the adaptation of additional mechanisms, such as Thread Mapping and Contention Management.

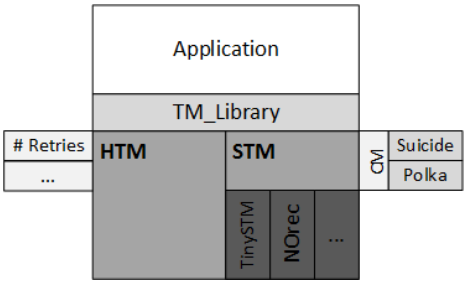

Fig. 2. Green-TM internal structure.

The entire set of TM algorithms, as well as, the auxiliary mechanisms to be incorporated in Green-TM are not yet fully defined. However, the plan is to support the reconfiguration of the following building blocks:

1. Active TM library - TinySTM, NOrec, HTM.
2. Contention Management - suicide, polka, exponential backoff.
3. HTM parameters - number of retries, technique to consume budget of attempts, technique to handle capacity aborts.
4. Maximum number of concurrently active threads.
5. Thread Mapping Strategy.

Since the library will adopt a modular and extensible design, it shall be easy to add/remove components.

3.2 Focus

The work that I will be doing on the following semester will be more focused on developing the Green-TM interfaces (Switching, Monitoring), and implement the strategies for supporting the dynamic reconfigurations of the various adaptable mechanisms encompassed by Green-TM.

Like mentioned before, the Oracles will be developed in parallel by others researchers at INESC-ID, and I will be using them to test their efficiency on the system.

It is important to mention that the planned work will include an intensive evaluation phase, aimed to assess the efficiency of the developed monitoring and reconfiguration strategies, as well as the accuracy of the prediction methodologies.

The evaluation plan, including the metrics used to this end, and the employed benchmarks, will be described in further detail in the next section.

4 Evaluation Method

The prototype developed during this dissertation will compare the efficiency (in terms of performance and energy consumption) of the proposed system (Sec-

tion 3) against state of the art TMs, e.g., TL2, NOrec, SwissTM, TSX, etc., and coarse/fine-grained locking solutions.

The key metrics to consider on this evaluation, are the following:

1. Throughput (committed transactions per second)
2. Abort rate (aborted transactions per second)
3. Energy consumption (Joule), available, e.g., via the Intel's RAPL interface [11]
4. Energy Delay Product (Execution time * Joule consumed during execution)
5. Reconfiguration latency
6. Monitoring Overhead (impact on throughput)
7. Computational cost for generating a prediction (at the Oracle side)

The system will be tested in a wide variety of scenarios: in workloads with contention ranging from low to high, with low or high percentage of time spent in transactions, with different number of threads, and using different hardware architectures (e.g. AMD vs INTEL).

The following benchmarks will be used to test the system:

1. STAMP: this is a standard suite of benchmarks for TM [25] that encompasses 8 different realistic applications, which generate very heterogeneous workloads
2. Memcached: this is a popular distributed object caching system, recently ported to use TM [28].
3. Microbenchmarks: concurrent data structures, such as linked list, redblack tree, skip list, which are parallelized using transactions and are frequently adopted in the TM literature [15]. Despite their simple and synthetic nature, these microbenchmarks have the advantage of generating easily predictable workloads that allow for stressing, in a controlled way, different building blocks of the TM system.

5 Work Plan

For the next semester, the work to be realized is scheduled as follows:

1. January 9 - May 31, 2014: Design and implementation of Green-TM backend support.
2. June 1 - June 31: Complete experimental evaluation.
3. July 1 - July 31: Finish writing the paper for the Green-TM project.
4. August 1 - September 31 - Conclude writing the dissertation.

6 Acknowledgements

I want to thank my advisor Professor Paolo Romano, for all the support and attention provided during the elaboration of the first part of the dissertation.

I also want to thank Diego Didona, who gave me a huge support during the conception of the written report, and to Shady Aala, who helped me get into the TM field at the very beginning of the thesis.

This work is supported by FCT GreenTM (EXPL/EEI ESS/0361/2013).

References

1. João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.
2. João Manuel Pinheiro Cachopo. *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Universidade Técnica de Lisboa, 2007.
3. Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: a hybrid transactional memory for haswell's restricted transactional memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 187–200. ACM, 2014.
4. Nuno Carvalho, João Cachopo, Luís Rodrigues, and António Rito Silva. Versioned transactional shared memory for the fénixedu web application. In *Proceedings of the 2nd workshop on Dependable distributed data management*, pages 15–18. ACM, 2008.
5. Márcio Castro, Luís Fabrício W Góes, and Jean-François Méhaut. Adaptive thread mapping strategies for transactional memory applications. *Journal of Parallel and Distributed Computing*, 74(9):2845–2859, 2014.
6. Maria Couceiro, Diego Didona, Luís Rodrigues, and Paolo Romano. Self-tuning in distributed transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, pages 418–448. Springer, 2015.
7. Maria Couceiro, Paolo Romano, and Luis Rodrigues. Polycert: Polymorphic self-optimizing replication for in-memory transactional grids. In *Proceedings of the 12th International Middleware Conference*, pages 300–319. International Federation for Information Processing, 2011.
8. Maria Couceiro, Pedro Ruivo, Paolo Romano, and Luis Rodrigues. Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation. pages 1–12, 2013.
9. Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L Scott, and Michael F Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. *ACM SIGPLAN Notices*, 46(3):39–52, 2011.
10. Luke Dalessandro, Michael F Spear, and Michael L Scott. Norec: streamlining stm by abolishing ownership records. 45(5):67–78, 2010.
11. Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: memory power estimation and capping. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on*, pages 189–194. IEEE, 2010.
12. Advanced Micro Devices. Proposed architectural specification. *45432 Revision: 2.1*, Mar 2009.
13. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. pages 194–208, 2006.
14. Nuno Diegues and Paolo Romano. Self-tuning intel transactional synchronization extensions.
15. Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 3–14. ACM, 2014.
16. Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

17. S Fernandes and Joao Cachopo. A scalable and efficient commit algorithm for the jvstm. In *Proc. of the 5th ACM SIGPLAN Workshop on Transactional Computing, Paris, France (April 2010)*, 2010.
18. Sérgio Miguel Fernandes and João Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, volume 46, pages 179–188. ACM, 2011.
19. Justin E Gottschlich, Manish Vachharajani, and Jeremy G Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 101–110. ACM, 2010.
20. Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In *Distributed Computing*, pages 303–323. Springer, 2005.
21. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.
22. Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.
23. Yossi Lev, Mark Moir, and Dan Nussbaum. Phtm: Phased transactional memory. 2007.
24. Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for java stm. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 314–325. ACM, 2008.
25. Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. pages 35–46, 2008.
26. J Eliot B Moss and Antony L Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.
27. Paolo Romano, Nuno Carvalho, Maria Couceiro, Luís Rodrigues, and Joao Cachopo. Towards the integration of distributed transactional memories in application servers clusters. pages 755–769, 2009.
28. Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using gcc and memcached. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 399–412. ACM, 2014.
29. Diego Rughetti, Paolo Romano, Francesco Quaglia, and Bruno Ciciani. Automatic tuning of the parallelism degree in hardware transactional memory. In *Euro-Par 2014 Parallel Processing*, pages 475–486. Springer, 2014.
30. Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Analytical/ml mixed approach for concurrency regulation in software transactional memory. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 81–91. IEEE, 2014.
31. William N Scherer III and Michael L Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005.
32. Michael F Spear, Virendra J Marathe, Luke Dalessandro, and Michael L Scott. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 338–339. ACM, 2007.

33. Michael F Spear, Arrvindh Shriraman, Luke Dalessandro, and Michael L Scott. Transactional mutex locks. In *SIGPLAN Workshop on Transactional Computing*, 2009.
34. Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 127–136. ACM, 2012.
35. Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. A transactional memory with automatic performance tuning. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):54, 2012.
36. Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 19. ACM, 2013.