Performance Modelling of Intel's Hardware Transactional Memory Implementation

Daniel Castro

Instituto Superior Técnico, University of Lisbon

Abstract. Transactional Memory (TM) is a recent alternative to traditional lock based synchronization mechanisms for parallel programming. This report analyses the state of the art in the area of performance modelling for transactional memory systems, as well as for concurrency control mechanisms for database management systems.
My analysis of existing literature in these areas highlights the existence of a relevant gap, which I aim to fill with my thesis work: the lack of performance models for hardware-based implementations of TM, also known as Hardware Transactional Memory (HTM).
More in detail, my dissertation will be aimed at building simulative and analytical models capable of capturing the performance dynamics of Intel's implementation of HTM.
In addition to defining the goals of my dissertation, this document also discusses initial ideas and preliminary results achieved so far.

Keywords: transactional memory, hardware, models, concurrency control

Table of Contents

1 Introduction

One of the main sources of complexity in parallel programming is related to the need of properly synchronizing accesses to shared memory regions. In fact, the traditional, lock-based approach to synchronize concurrent memory accesses is well-known to be error prone even for experienced programmers: on the one hand, using coarse-grained locking, e.g., synchronizing any concurrent access via a single lock can severely limit parallelism; on the other hand, while the usage of fine-grained locks can enable larger degrees of parallelism, it also opens the possibility of deadlocks and hinders a key property at the basis of modern software engineering, composability [1].

Recently, Transactional Memory (TM) has emerged as a simpler, and hence more attractive, alternative to lock-based synchronization: unlike locking, TM simply requires programmers to identify which code blocks should appear as executed atomically and not how atomicity should be achieved.

Over the last 20 years, several implementations of TM have been proposed, using either software, hardware, or a combination of both. Yet, at current date, performance of TM remains a controversial issue.

Cascaval et al. [2], for instance, harshly criticized Software-based TM implementations, arguing that such a generic concurrency control mechanism imposes

necessarily large overheads to track the memory regions accessed within transactions. Their results revealed that, in most tests, Software Transactional Memory (STM) behaves worse than sequential code (using 2-4 threads).

On the other hand, hardware implementations of TM (HTM) avoid the instrumentation costs incurred by STM, but their nature is inherently restricted and best-effort. In fact, commercially available HTM implementations (such as the ones provided by Intel Haswell or IBM P8 processors) rely on cache coherency protocols to keep track of the memory regions accessed by transactions. As such, they suffer of spurious aborts whenever relevant transactional metadata has to be evicted by the cache (e.g., due to cache capacity limitations).

These widely available HTMs, launched with recent processors, thus provide performance that varies significantly with the workload. However, the internals of their implementations are mostly undisclosed, for which reason it is hard to predict a priori how an HTM will perform for a given new application.

This work aims to develop tools for predicting the performance of HTM-based applications. This objective will be achieved in two steps:

1. First a discrete event based simulator will be developed. This simulator will focus on modelling the performance dynamics of the cache coherency protocol used by Intel Haswell processors, which, as it will be discussed in more detail in Section 2.3, represents the backbone used to implement the HTM provided by Intel.
2. Next an analytical model of Haswell's HTM will be developed. The analytical model will rely on queuing theory and probabilistic techniques (extending prior literature in the area of performance modelling of concurrency control algorithms in STMs and Database Management Systems (DBMSs) literature [3–7]) and will be validated by means of the event based simulator.

The remainder of this document is structured as follows. In Section 2 the related work is discussed, with emphasis on the different implementations of TM both in software and hardware, as well as on works that aim to model STMs and DBMSs. In section 3, the proposed solution is discussed. Besides discussing the architecture of the analytical and of the simulation model, this section also presents the results of preliminary work aimed at characterizing the behaviour of Haswell's HTM. In Section 4 the evaluation methodology is presented to explain how the obtained results will be corroborated for the hypotheses established in this thesis proposal. In section 5, the calendarization of the proposed tasks is presented. Finally, in section 6, conclusions of this mid term report are given.

2 Related Work

During the past decade there has been an intense research in the area of TM, motivated by the current paradigm of multi-core hardware, which creates a great need for concurrency control mechanisms that are simple to use and yet provide scalable performance.

This Section is structured as follows. First, in Section 2.1, an overview of TM is presented. Section 2.2 discusses software-based TM implementations. Section 2.3 overviews hardware-based TM implementations, including both restricted TM implementations (such as Intel's one) and unbounded ones, which overcome the limitations of best-effort HTMs at the cost of additional complexity at the hardware level. Then, the combination of best-effort HTM and STM, also known as Hybrid Transactional Memory (HyTM), is discussed in Section 2.4. Section 2.5 overviews some of the most popular for TM libraries. In Section 2.6, HTM, STM and HyTM are compared with each other. The focus of the subsequent sections is on the modelling of performance of transactional systems. In Section 2.7 black-box and white-box techniques are surveyed. Approaches based on analytical modelling are presented in Section 2.8. In Section 2.9, simulation techniques are discussed and, finally, in Section 2.10 current research focused on increasing the performance of TM systems is presented.

2.1 Transactional Memory

Firstly, the usage of TM will be motivated. When programming parallel applications, there are certain regions of code (i.e., critical sections) that must run atomically in order to guarantee the application state consistency. If two critical sections run concurrently without any synchronization, the application state may become inconsistent (i.e., a conflict occurred).

An easy way of completely avoiding conflicts is by introducing a global lock, allowing the critical section to only execute sequentially. As such approach degrades performance, there is the possibility of using fine-grained locking (read-/write locks, and each memory region may have its own lock). But identifying the optimal fine-grained locking scheme for each critical section is well known to be time consuming and error prone [8].

On the other hand, with the usage of TM, the programmer only has to annotate the critical section (i.e., transactional code block) with the `atomic` primitive.

The reference correctness criterion for TM is opacity [9]. Opacity is stronger than Serializability, the strongest correctness criterion typically enforced in a database management system. Serializability requires that the (concurrent) execution of committed transactions lead to a state that could be obtained by running these transactions in some sequential order. Roughly speaking, opacity enforces two additional guarantees: i) the serialization order of committed transactions has to preserve the real-time order of execution of transactions (as in strict serializability [10]); ii) aborted transactions should observe a state producible by executing transactions in some sequential order (possibly different from the serialization order imposed to committed transactions).

As a matter of fact, in most real-life applications, conflicts are not very common. Therefore, avoiding them is not the most efficient approach. TM will run each critical section speculatively and, then, if no conflict is detected, the modifications are committed. Nevertheless, if a conflict in fact occurs, one of the conflicting transactions may have to abort and restart later.

Ideally, the TM library would take care of each memory access without calling any read/write routines (i.e., implicit instrumentation), support arbitrarily large transactions (i.e., unbounded) and not need a priori awareness of which memory region are going to be accessed within the transaction (i.e., dynamic).

However, for research purposes and optimization (where the flexibility of changing the implementation in prototypes matters), the library may allow the usage of transactional read/write routines within the critical section (i.e., explicit instrumentation), and for simplification purposes, transactions that accesses too many memory regions (large footprint) may not be allowed (i.e., bounded), as well as, only a certain part of the memory may be accessible within the transaction (i.e., static).

For example, in program 1.1, variable `a` will never be less than 0, and the only change, regarding the non synchronized version, was the usage of `__transaction_atomic` to identify the critical section. The advantage in comparison with a global lock is that transactional code blocks can run truly in parallel and the locked critical section must be executed sequentially.

To compile a program using TM in GCC, the following command can be used: `gcc -Wall -fgnu-tm -O3 -o a.out test_program.c`. Nevertheless, the programmer can change the TM library to other than the included in GCC by default. The accesses to memory regions within `__transaction_atomic` will be translated by the compiler with respect to the provided library, which can be STM, HTM or a hybrid solution.

Program 1.1: TM example in C language.

```
1  int a = 2, b = 0;
2
3  void T1() {
4    __transaction_atomic { if(a >= 2) { a -= 2; b += 2 } }
5  }
6
7  void T2() {
8    __transaction_atomic { if(a >= 2) { a -= 2; } }
9  }
10
11 int main() {
12   // launch two threads to execute T1 and T2 in parallel
13 }
```

2.2 Software Transactional Memory

STMs were deeply investigated because they allow to build very flexible prototypes, as the full concurrency control is implemented in software, allowing to quickly prototype many different design choices.

Under the hood, STM implementations may use different memory regions access granularities (i.e., could be each memory address, chunk of memory or

object). Most implementations have structures that identify which memory regions are being accessed by each transactions (i.e., ownership records) [11, 6, 2]. Nevertheless, in Dalessandro et al. [12], NoREC is proposed as an alternative to ownership records.

Regarding the conflict detection, STM implementations may be eager, i.e., they perform synchronization immediately during the speculative execution upon each access to shared memory, or lazy, i.e., at commit time. Eager conflict detection may provide better performance, since transactions immediately abort upon detecting a conflict. This approach can be implemented by detecting if one transaction is writing some memory region that is also being accessed by other transaction. On the other hand, eagerly aborting a transaction may lead to aborting more transactions that strictly necessary, e.g., T conflicts with T', eager approaches would immediately abort T; then T' aborts, lazy approaches would let T continue.

Correctness, i.e., opacity [9], is generally, achieved by creating versions of the accessed memory regions and running the transaction speculatively. Versions may be maintained using timestamps and storing the modified values locally (in a redo log) before committing them to the shared memory. Alternatively, versions may be written in place, storing the old values in an undo log that is used to restore a consistent state in case the transaction has to be aborted. Other lock-based implementations will use fine-grained locking to protect the memory regions and some deadlock detection mechanism to restart blocked transactions. As expected, whatever the chosen implementation, overheads will be introduced regarding the sequential execution of the same application.

Transactional Locking II (TL2) [13], TinySTM [14, 15] and SwissTM [16, 17] use similar approaches to the ones described above. They have a metadata structure where the current timestamp of each memory region is stored, the timestamp is usually fetched from a global counter. At commit-time, the transaction is validated by checking whether the timestamps of the resources read is never greater than the transaction timestamp (i.e., other transaction concurrently changed the resource during the first transaction execution) and the actual changes to memory are made at commit-time. A locking strategy is then followed to ensure that the changes to memory are atomically made.

As seen before, accessing memory in transactional blocks has a greater overhead than accessing memory in non-transactional blocks. Quantifying how large the introduced overhead was a matter of study [2, 17], and there are contradictory results. While in Cascaval et al. [2] TM is harshly criticised; in Dragojevic et al. [17], it is shown that STM can scale very well in the presence of increasingly more physical processing units, suggesting that, with further improvements, TM will be able to outperform other, more error prone, synchronization techniques.

Presently, high level programming languages are introducing new ways of simplifying parallel programming based on transactional memory. For instance, in Java, by identifying transactional blocks with an @Atomic annotation, inexperienced programmers can write simple, yet correct, parallel applications [18].

Other high level programming languages let the programmer use built-in keywords to identify transactional blocks, e.g., Clojure has the `dosync` keyword.

2.3 Hardware Transactional Memory

Implementing TM at the hardware, i.e., at the cache coherence protocol, level reduces the overhead of synchronizing memory accesses, regarding software implementations, thus achieving better performance.

Although proposes for unbounded HTM already exist [19–21], changing the processor to support HTM is very complex, so Intel and others have opted for making minimalistic changes to the cache coherence protocols of CPUs that result in limited best-effort HTMs.

Besides Intel, also IBM has commercially available multiprocessors that support best-effort HTM. Both Intel's and IBM's implementation of HTM are based on relatively simple and non-intrusive extensions of the cache coherence protocols [22]. Therefore, they have limitations due to cache capacity, non transactional code also using the cache and how memory addresses are mapped into cache lines (i.e., if some addresses map to the same cache line it will overflow even if the transaction does not use much memory).

Unbounded Hardware Transactional Memory. As best-effort HTM maintains data accessed within transactions (and the corresponding metadata) in cache, it cannot handle transactions with large footprints (i.e., accessing many more memory lines than the lines that can be cached in the private cache), there are some works that address this issue [19–21], but its efficient implementation introduces further complexity in hardware.

A first approach of Unbounded Transactional Memory (UTM), was the Large Transactional Memory (LTM) [19], that supports nearly unbounded transaction, which is enough for real life application. Its implementations is similar to best-effort HTM, but it modifies the cache, so that cache lines that hold transactional values and overflow go into uncached DRAM in virtual address space. Nevertheless, when a context switch occurs, all uncommitted transactions must abort, because those cache lines that hold transactional values could be used by other threads and using uncached DRAM as a cache is extremely slow.

Virtual Transactional Memory (VTM) [21] follows a similar approach to LTM, as it uses the virtual address space to store the overflowed cache lines into memory. Each processor has a VTM system, and, whenever the processor requests an address within a transaction, it is responsible of updating a special structure (held in virtual memory) with the needed metadata to handle conflicts. Therefore, every memory address that the transaction uses does not need to be in cache at the same time. Also, as this structure is in memory, transactions can survive context switches and can be debugged.

Another UTM solution is logTM [20], which adopts a log based approach to track the accessed memory regions, similar to what is used in DBMS. To implement it, some new registers must be added to the processor to handle

where the transaction started and where the log currently is. As in VTM, a special structure must be held in memory with the transaction log. It was shown to have better performance than VTM, but at the time of publication it still lacked support for paging and context switching.

Best-effort HTM. Differently from UTM, best-effort HTM can only execute transaction with a footprint smaller than the cache size, and do not survive context switches (i.e., transactions that take more time than a time slice are aborted). Apart of the time limitation, the number of addresses than a transaction can use depends on the workload. Assuming that the addresses are mapped into cache using the least significant bits, if a transaction uses contiguous memory, theoretically, it can use more memory than a transaction that accesses distant addresses. This is due to cache associativity, due to which each cache line can hold n words (e.g., a 4-way associative cache line can hold 4 words).

As the full specification of HTM implementations is not completely known, several works aimed at reverse-engineering commercial HTM systems to shed lights on their internal mechanisms and on their actual limitations. This line of research is important to allow the development of models that predict its performance (e.g., maximum degree of parallelism, data contention, abort rate and throughput), which is also the main focus of this report.

The following results and conclusions were drawn from recent studies that compared Intel and IBM HTM solutions [22, 23], although this report focuses mostly on Intel's implementation. The multiprocessor tested (Intel Core i7-4770) has 32KB, 8-way L1 data cache (512 cache lines), 256KB, 8-way L2 data cache (4 096 cache lines) and 8MB, 8-way L3 data cache (131 072 cache lines).

According to the experiments reported by Hasenplaugh et al. [23], it was possible to conclude that load (reads) and store (writes) maximum capacities are 4MB and 22KB, using read-only and write-only transactions respectively. Note that the reads capacity vastly surpass L1 and L2 cache capacities, which suggests that it uses L3 shared cache (or some space efficient encoding, e.g., based on bloom-filter [24]) to keep track of the read values. More in detail:

- A transaction can read around 75 000 contiguous lines with 58% success rate (after 32 500 lines the success rate starts dropping from 98%);
- A transaction can write around 400 contiguous lines with 100% success rate (after that it drops to 0%);

According to these results, apparently all transactional writes must fit entirely inside L1 cache ($400 lines \times 64B = 25600B$). Although memory accesses are made contiguously, the total L1 cache size of 32KB is not reached. There is also a good correlation between the maximum cache lines and the inverse of the logarithm of the accessed contiguous lines, which suggests that memory addresses are hashed using the least significant bits.

Molka et al. [25] discuss how the Modified, Exclusive, Shared, Invalid, Forward (MESIF) protocol was used to support Transactional Synchronization Extensions (TSX). In MESIF, each cache line may be in one of the following states:

Modified: the cache line is being used only in the current cache, and the value does not match the main memory (was updated);

Exclusive: the cache line is being used only in the current cache, and the value matches the main memory;

Shared: other private caches may be using the cache line, and the value matches the main memory;

Invalid: other processor updated the cache line locally and an invalidation signal was received, or the cache line is not being used (free);

Forward: when a memory region is accessed, the current private cache requests the value to other private caches, setting the state to forward if some other cache has the value in shared mode (does not ensure that the closest resource is the one that is gathered).

When using HTM approaches, one has always to specify a fall-back synchronization mechanism that is activated when a transaction cannot be successfully committed using hardware-aided transactions. This fallback-path is typically a global lock. Firstly, because it is an easy solution and aborts should not appear very often. Therefore, most of the time transactions should be using HTM and not the global lock (although this clearly depends on the actual workload characteristics). Secondly, falling back to STM is not simple because neither the software layer has access to best-effort HTM internals, nor HTM is aware of the STM. Therefore, if one transaction is using HTM and another uses STM undetectable conflicts would occur (as if no synchronization was used at all).

2.4 Hybrid Transactional Memory

As both best-effort HTM and STM have drawbacks, one may try to combine both in order to get the best of the two. As a matter of fact, currently, most multi-core processors available in the market (also for the foreseeable future) support only best-effort HTMs, which, as already discussed, can suffer of severe limitations when faced with workloads including long transactions. Yet, any programmer relying on TM expects a complete solution that works all the time with all kinds of transactions.

Hybrid Transactional Memory (HyTM) [26] was proposed to fill this gap in current HTM libraries. Preference is given to the best-effort HTM, but the implementation is not completely HTM dependent, overcoming some of its limitations by using a STM library as fall-back path. Therefore, if the HTM library is not capable of handling the transaction, there is a STM routine that will handle it in the future.

Nevertheless, falling back to STM is not trivial. If a STM transaction is running, HTM transactions will not be aware of which addresses are being accessed by STM transactions.

The experiments in Dameron et al. [26] showed that this approach achieves better results than using only a STM library. Also, using many cores, the solution scaled as well as logTM (tested in a simulator), which is an unbounded HTM

solution, therefore, being an accessible solution that makes use of the currently available best-effort HTM.

Other more recent studies [8], though, revealed that existing HyTM solutions, when employed in commercial HTM-enabled processors, such as Intel's Haswell, can suffer of strong overheads and deliver, in many realistic workloads, lower performances than approaches based on pure STM or HTM.

2.5 Benchmarks for Transactional Memory

Since various TM implementations exist, the programmer must be able to compare them in order to pick the one that best suits the programmer needs.

Currently, there are lots of parallel applications that make use of locks for concurrency control. With their adaptation to use transactional memory, it is possible to compare different locking systems and how well different versions of TM behaves.

The benchmarks, therefore, are either real world or synthetic applications that use TM to synchronize concurrent access. In most cases, one must implement a wrapper to his/her library, then the benchmark, using explicit instrumentation, will use the given TM library.

Some of the most popular benchmarks for TM systems are the following:

Lee-TM: A circuit routing algorithm. Due to the large number of routings that a typical circuit needs, large degrees of parallelism must be achieved. Hence, it is an application that needs alternatives to traditional lock-based synchronization techniques [27];

STMBench7: An implementation of a cooperative Computer Aided Design (CAD) editing environment. This benchmark represents the porting to TM environments of the OO7 benchmark, originally proposed for object-oriented DBMSs [28];

Memcached: An object caching system used in many web services. Mainly, after database or API calls, this application stores its results to avoid new calls, hence improving the system's overall responsiveness. The usage of TM should improve the storage capabilities, by allowing the results of asynchronous calls to be stored in memory concurrently [29];

STAMP: A suit of eight different applications, each provided with different inputs and configurations, therefore, generating different workloads to stress various aspects within the given TM library, namely, transaction length, contention and scalability [30].

2.6 Comparison between STM, HTM and HyTM

As discussed in the previous sections, STM, HTM and HyTM have limitations, which I summarize in the following:

STM: Large overhead due to the use of software structures that handle resource ownership and versioning are needed. It is suitable for large transactions (i.e.,

unbounded), e.g., in C programming language often memory allocation routines (i.e., `malloc`, `calloc`, `realloc` and `free`) have to be replaced. Also, the programmer may have to explicitly call read/write routines to access memory regions inside transactional blocks (though some exceptions exist [13]);

best-effort HTM: Small overhead thanks to the hardware support and to their best-effort nature. However, it is not suitable for large transactions and aborts frequently due to non transactional events (e.g., context switching). Also, a fall-back path must be defined. e.g., One advantage is that there is no need to explicitly call read/write routines to access memory;

HyTM: Tries to handle transactions, firstly, using HTM and then, if hardware cannot handle it, STM is used. While appealing in theory, the coupling of HTM and STM can introduce non-negligible overheads, which can strongly hamper performance in practice.

To finalize the TM related work, Intel's HTM will be the primary focus of this thesis due to its novelty and availability in commodity end-user machines (unlike IBM's processors). Also, its underline mechanics have not being completely undisclosed yet, which makes it an interesting object study.

2.7 Basic methodologies for performance modelling

As seen before, TM performance is highly dependent on the application workload. Therefore, models are needed to predict whether a TM solution is suitable to some new application or not.

Performance models mainly divided in the following three techniques: white-box, in which the model exploit detailed knowledge about the internals of the target application; black box, in which the performance model is inferred by means of collecting a training dataset and using a learning algorithm; gray box, which combines the previous two approaches.

As a goal for this thesis is to develop an Analytical Model (AM) to predict best-effort HTM performance, this section surveys similar work on the development of models for various systems.

White-box models. White-box models, like analytical and simulative models, assume and exploit the knowledge of internals of the target system to predict its performance dynamics. In the case of AM, the system is represented by means of equations, which may be hard to devise and generalize.

However, white-box models have the advantage over the black-box approaches of not having to be fed with large learning datasets. Also, by looking into the equation, it is possible to understand why the predictions are being made.

Often queueing theory and discrete-time Markov chains are useful tools in describing the entire system, or subsystems (then aggregated in a more global model). From there, the fundamental equations are derived. [31, 7]

Usually, the more input parameters the model has the more accurate and versatile the model is. For instance, using as input parameters the time it takes

to read, write, begin the transaction, commit it, the mean number of transactions and the mean number of reads and writes per transaction, the model could predict the expected execution time for a specific workload.

Also, AMs can model specific implementations of systems by taking as input, for example, the characteristics of some in-memory structures to predict, for example, conflicts.

Simulation can be combined with AM in order to infer more information from the system (e.g. abort rate distribution), which can be incorporated in the AM.

A wide body of literature has been published on the development of AMs aimed to capture the performance of alternative concurrency control mechanisms [3–5, 31, 32, 7]. Given the reliance on the common abstraction of atomic transactions, the literature on AM of concurrency control for DBMS will be reviewed in Section 2.8.

Actually, AMs applied in the context of databases is important in order to choose the right concurrency control mechanism. Due to some similarities with TM systems it will also be included in this report in further sections.

Black-box models. Black-box modelling approaches are based on the idea of learning statistical relations between a set of input variables (called features) and one or more output variables. As the name suggests, the target system is treated as a black-box, in the sense that no information on its internals is assumed. The dataset provided as input to build a black-box model is typically called training set. Some of the most important requirements to build good training sets are:

- good coverage of the input domain;
- not over-training certain areas of the input domain;
- low incidence of outliers in the training dataset.

Black-box models are mainly Machine-Learning (ML) approaches, like, e.g., neural networks [33] and support vector machines [34].

However, predictions from ML approaches are not easily explained, as the output model is not, in general, interpretable by humans. On the other hand, one can find out why some prediction was given by an AM by looking into the equations.

Nevertheless, if the system internals change, ML approaches will be more resilient than AM approaches due to be possible to re-train them with a new dataset. If provided with sufficient training data reflecting the system's change, in fact, ML approaches can automatically derive an updated model. When using AM techniques, however, a new set of equations will have to be manually derived (and validated).

However, if a system characterization changes, and the AM captures it, by updating this new value, the predictions shall continue accurate, not needing additional effort as in ML.

Performance predictors using ML techniques exist associated with STM systems, e.g., [35, 36].

With these approaches, first, some ML technique is used to learn an initial training set. Next, a profiling module extracts runtime information from the workload to feed the ML-based model and possibly optimize the system according to the model's predictions.

Gray-box models. Both analytical and machine-learning models have drawbacks, i.e., equations for the AM may be difficult to derive and ML, though more generic, requires gathering a large training set. Thus, one may try to combine both to achieve better predictions.

In gray-box techniques, both white-box and black-box models are combined in order to reduce the drawbacks of each other, improve performance and/or accuracy. To do the combination of the models one of the following approaches is usually taken:

- subdivide a complex system into simpler subsystems; then, identify the subsystems that are tractable for analytical modelling, and apply some ML technique to those remaining; finally devise a formula to combine the outputs of each modelling of each subsystem [37, 38];
- bootstrap the ML model with the AM, in order to alleviate the need for gathering a large initial training set. Initially, the ML should be as good as the AM; then, the ML is improved with workload information gathered from the running application, thus, improving its predictions [39, 40];
- combine AM and ML using other Machine Learning techniques to learn when one model outperforms the other [41].

2.8 Performance modelling of transactional systems

This section is devoted to overviewing existing literature in the area of analytical modelling of performance of TM and Database Management System (DBMS).

Database Management System models. Relational databases are a robust and well studied technology, and a large body of literature has been devoted to model the performance of the various concurrency control mechanisms proposed for relational DBMSs [3–5]. While Tay et al. [3] and Yu et al. [5] propose analytical models, in Agrawal et al. [4], already existing analytical models are compared by means of simulation models.

DBMS are closely related with TM, given that they share the abstraction of atomic transaction. Nevertheless, there are also some differences, DBMSs are complex systems, they may support distributed databases, accesses from remote users, crash recovery, durability, etc, which TM is not supposed to support. TM will run in the same machine, each transaction will use volatile memory. Hence, several mechanisms that are required by DBMSs, are not relevant in a TM environment. Also, TM transactions execute in a non-sandboxed environment, unlike DBMSs. This motivates the adoption of more conservative consistency criterion in TM, e.g., opacity vs serializability [9].

Also, in DBMS, sockets are often used to connect to the database, with commands being written in Structured Query Language (SQL), thus introducing communication and parsing overheads, which does not exist TM systems.

In the DBMS literature, a large number of alternative concurrency control schemes have been proposed. Existing approaches to concurrency control can be coarsely clustered in two main classes: optimistic and pessimistic concurrency control [5].

An optimistic concurrency control approach will deal better with low data contention scenarios. A pessimistic concurrency control approach, on the other hand, behaves better in the presence of high data contention (as it does not have to rollback to previous states).

Most AMs of DBMS focus on predicting the probability of a transaction aborting (P_A). For instance, inputs for this model could be the mean number of accessed resources (N_L), the transaction arrival rate (λ), the total number of available resources (L), in average how much time a resource is held (T_H) and the mean value of the commit time (T_{Commit}).

For Strict Two Phase Locking, one of the most popular pessimistic concurrency control mechanisms, the following analytical formula has been derived to predict the transaction abort probability [5]:

$$P_A \approx 1 - \exp(-\frac{\lambda N_L^2 T_H}{L})$$

And for optimistic concurrency control mechanisms:

$$P_A \approx 1 - \exp(\frac{-\lambda N_L^2 T_H}{L})(1 - \frac{\lambda N_L T_{\text{Commit}}}{L})^{N_L}$$

In this model, the optimistic concurrency control mechanism is expected to abort more often than the pessimistic approach. On the other hand, the optimistic version is more responsive, thus having smaller T_H. This model assumes that each resource have equal probability of being accessed, which is not the case in most applications workloads.

Also, the predictions are mostly approximations due to the complexity the system would introduce in the model. But, the obtained results for the previous model [5], accurately predict the real system performance, and the obtained deviation scaled linearly with the input domain.

TM models. Given that the TM area is much more recent and unexplored than conventional DBMSs, as expectable, there is a smaller number of works devoted to the performance modelling of TM systems.

In particular, the only works that we are aware of that target performance modelling of TM have considered exclusively Software-based TM implementations. To the best of our knowledge, in fact, there are no works in the literature that tackled the problem of modelling the performance of Hardware, or Hybrid, TM systems. One of the goals of this dissertation consists precisely in moving a step towards filling this gap by deriving analytical models capturing the performance of best-effort HTM systems.

Among the work that targeted the performance modelling of STM, Zilles et al. [6] have shown that there is a close resemblance between the probability of conflicts between transaction and the birthday paradox.

According to this model, given the amount of accessed memory regions (*footprint*) and the size of the ownership table (*table_size*, most STMs have a in memory structure to handle which memory regions are owned by each transaction), it is possible to predict the conflict likelihood (*conflict*) as follows:

$$\text{conflict} \propto \frac{\text{footprint}^2}{\text{table\_size}} \qquad \text{conflict} \propto \text{concurrency}^2$$

This means that, the larger this table is, the smaller is the probability that two memory regions maps to the same entry, which, intuitively, makes sense.

Also, response time, commit probability, abort probability, throughput, etc, was taken in account in various works [31, 32, 7]. Most approaches use discrete-time Markov chains and queueing theory to represent the transaction or thread behaviour.

Some works try to relax assumptions in order to devise simpler models, e.g., assuming that the transaction restarts right after it aborted [31]. These assumptions, generally, are made in a way that do not compromise the results much, thus, achieving predictions close to what is observed in the real system.

As TM systems are complex, other lines of work resort to using ML techniques. These approaches typically construct an initial model using training data gathered using a mix of predefined applications. Then, via a profile, performance traces acquired from the target application's workload are acquired in order to fine-tune the model and enhance its accuracy.

There are also works that tackle the cold start problem of ML approaches by combining an AM, resulting in better performance and better initial predictions. For instance, In Rughetti and Di Sanzo [43] use an analytical estimator (f_A), alongside with some initial profiled data, estimates the initial virtual training set for the analytical ML estimator (f_{AML}) which creates a virtual-real mixed trained set. The last training set is updated with profiled data from the running application and feeds a neural network estimator (f_{ML}), which predicts the application performance.

Finally. Didona and Romano [39] introduce a technique called bootstrapping. This technique relies on an AM to generate an initial training set that is used to build an initial black-box model.

This approach reduces the initial profiling stage and generates an accurate non overfitting training set.

The machine learning model will initially behave exactly like the analytic model, and then with more samples from the actual application the machine learning model will achieve increasingly better results.

2.9 Simulation models for transactional systems

In contrast with AM, simulations produce statistical results gathered from the an actual representation of the system. Within a simulation environment, non-deterministic events can be controlled as well as the complexity of the system. Simulation is usually used during the development of an AM, in order to verify to what extend the introduction of assumptions aimed to simplify the model can impact its ultimate accuracy.

Some experimental transactional systems may do not have any support in physical systems (as in UTM, see Section 2.3). Hence, modelling the performance of these systems depends on how accurate the simulator is.

Simulators for DBMS. Simulators have been long used in the DBMS literature and developed with two main purposes:

1. validating some AM of specific components of a DBMS. This is the case for instance for the work by Tay et al. [3] and Yu et al. [5], which used simulation to validate AM of the performance of various concurrency control schemes.
2. predict the performance trends of a database as a whole [44], or shedding light on the performance dynamics of specific modules of a DBMS (e.g., real-time transactional scheduling [45] or buffer management [46]).

TM simulators. As seen before, simulators are important when the piece of hardware that is needed still does not exist or is unavailable (as in Section 2.3).

AMD has an experimental HTM instruction set, called Advanced Synchronization Facility (ASF) [47], which should compete with Intel TSX, but was not released (yet). As it was a prototype for AMD multiprocessors, this instruction was implemented in a cycle-accurate simulator. Then, by extending a C/C++ compiler to generate ASF, it was run the STAMP benchmark.

Cycle-accurate simulators are able to simulate real systems, and run complex software (e.g., operating systems) on top of them. One important feature is that they are able to capture the time each event would take in the real system by simulating low level machine execution. For instance, PTLsim [48] is a x86-64 simulator that explicitly simulates the program counter and other resources. The number of cycles that each operation takes is then made available for the subsequent statistical analysis.

2.10 Tuning of TM

As shown in Section 2.3, best-effort HTM are not capable of handling certain types of transactions. Therefore, a fall-back path must use other synchronization methodology to execute the transaction. As best-effort HTM do abort for a lot of reasons, falling back right away is not the smartest approach.

In Diegues et al. [49], lightweight reinforcement learning techniques (i.e., hill climbing [50], an on-line black-box modelling technique) was exploited to identify the optimal number of retries that should be attempted when using Intel TSX. The approach works fully on line and does not require any a priori workload characterization.

After a transaction starts, depending of the current application behaviour, the tuner may want to optimize the current configuration, and if so, it starts profiling cycles, fetches the previous configuration and then starts the transaction. After the transactions ends, according to the profiled information, the optimal configuration is predicted for the next transaction. Using this approach,

it was shown a significant performance increase over other approaches. It was also shown that the proposed solution behaves only 5% worse than using optimal static configurations, with the advantage of being dynamic and capable of adapting to any workload.

3 Solution Architecture

This section discusses how I intend to pursue the goals of my dissertation, which I recall, consist in developing an analytical and a simulation model of best-effort HTM systems.

More in detail, Section 2.3 presents a set of information regarding the internal architecture of Intel processors that is most relevant for the HTM subsystem. This information has been gathered by studying three main sources in the public literature [51, 22, 23].

Next, Section 2.3 reports the results of a preliminary experimental study that has been aimed to reverse engineer the concurrency control algorithm implemented by Intel's TSX. This information, which is unfortunately undisclosed by Intel, represents a fundamental preliminary step to purse the goal of building both analytical and simulation models of Intel's TSX.

3.1 Preliminary investigation on the inner design of Intel TSX

Intel TSX capacity limits rely on the cache size [51, 22, 23]. In Table 1, the specifications for a Haswell processor. Similar specifications are found in Xeon and Broadwell processors. In most recent processors from Skylake family, this specifications also apply.

Level	**Capacity (KB)**	**Associativity**	**Line size (Bytes)**	**Fastest Latency**	**Peak Bandwidth (Bytes/Cycle)**	**Update policy**
L1	32	8-way	64	4 cycles	64 (Load), 32 (Store)	Writeback
L2	256	8-way	64	11 cycles	64	Writeback
L3	varies	varies	64	34 cycles	varies	Writeback

Table 1: Intel's Haswell specification regarding its cache system.

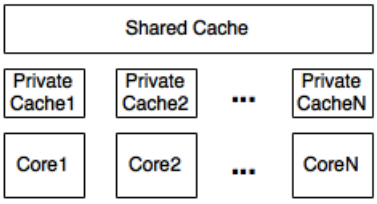
In a multi-core environment, there are several cache levels. For the sake of simplicity, let us abstract the system using the following levels, as in Figure 1a:

- private cache, different for each core;
- shared cache, accessible to all cores.

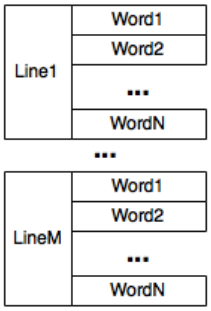
In Intel's case, most processors have three levels of cache (L1 and L2 are private, L3 is shared). For now, the system is abstracted as only having the previous two types of cache: private (one per core) and shared. Also, taking in account the conclusions from Hasenplaugh et al. [23] in Section 2.3, we will

assume that TSX handles writes at private cache level and reads at shared cache level.

When a processor updates a value, the cache manager is responsible for updating higher level caches and ensuring the consistency of the private caches of other processors, invalidating any changed private value.



(a) Cache hierarchy abstraction.



(b) N associative cache abstraction.

Fig. 1: Cache components.

In current processors each cache has a fixed number of lines, and each line has a fixed number of words. Each word stores a value from the main memory. Intel uses the MESIF protocol, so each word can be in one of its five states (modified, exclusive, shared, invalid or forward, see Section 2.3). Whenever a transaction accesses a memory word, the cached memory word is established as *owned* by that transaction: if some *owned* word gets an invalidation signal, then a transaction that is running in that processor core is immediately aborted; furthermore, if the cache is full and a word that is *owned* by a transaction has to be evicted, the corresponding transaction is aborted.

Reverse engineering TSX's concurrency control. Intel did not disclose any detail regarding the concurrency control employed by TSX to ensure consistency. Yet this information is of paramount in order to build detailed white-box models of TSX of either simulative or analytical nature. This has led us to design a set of 6 test cases based on the usage of two concurrent transactions, which execute different patterns of read/write accesses to the same shared data item. The experiment has the ultimate objective of inferring how conflicts are managed in Intel's implementation of HTM.

Figure 2 reports the 6 tested transactional schedules. In order to enforce the desired access order to memory by the concurrent threads, empty loop cycles were added to the transaction code (i.e., spinning times). To avoid GCC to eliminate them through its optimizer, no optimization level was used (i.e., -O0).

Assuming that most of the time both threads are running in parallel, the schedules in Figure 2 were achieved by adjusting the spinning times. Nevertheless, as the current process may switch (i.e., our application was not the only one running in the machine), some noise was observed, to reduce its impact, the

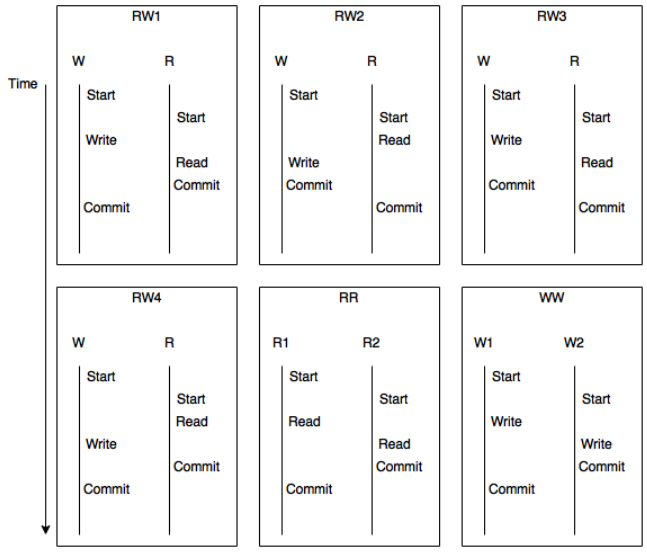


Fig. 2: Tested schedules in Intel TSX.

	R	**W**
RW1	C	A
RW2	A	C
RW3	C	A
RW4	A	C

	R1	**R2**
RR	C	C

	W1	**W2**
WW	A	C

Table 2: Conflict results, A means aborted and C means committed.

experiment was repeated over 50 000 times for each schedule and the results that were observed more often are the ones recorded in the Table 2.

According to Table 2, if a transaction T accesses a memory word, and later a second transaction T' conflicts on the same word with T, then T is aborted eagerly, i.e., as soon as T' causes the conflict.

A more interesting, and definitely necessary, set of experiments will be aimed to profile the costs, in terms of processor cycles, of operations like begin, commit (in successful or unsuccessful cases), and abort.

3.2 Best-effort HTM simulation

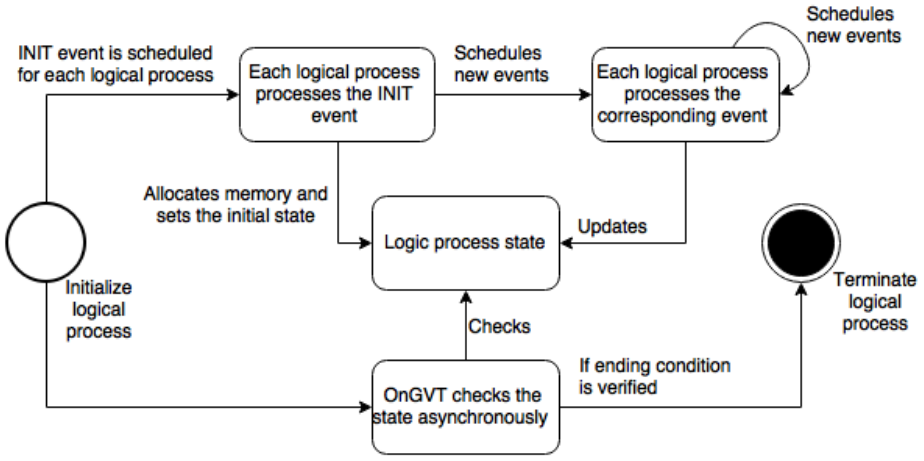


Fig. 3: ROOT-SIM behaviour.

To simulate the best-effort HTM we will use a event-driven simulator. We choose ROme OpTimistic Simulator (ROOT-SIM) [52] because it supports multi-thread simulations which might be faster than sequential ones, and the API is also very concise, relying only in four routines to schedule new events, process scheduled events, set the global state of the simulation and check ending conditions. In Figure 3, a simplification on how ROOT-SIM works is presented.

Early experimentation were made to check whether, from within this simulator, a complex best-effort HTM simulation can be easily simulated or not. As we do not want to drop it after several modules are already made.

Taking in account the previous information, we will create a simulator capable of extracting workloads from well known benchmarks (e.g., Lee-TM, STMBench7 and Memcached) and run the simulation.

The simulation model to be developed in my dissertation will focus in particular on modelling the dynamics of the processor's caches, given that this component is expected to have the strongest impact on the performance of HTM-based applications.

In particular, the simulation model will implement the MESIF cache coherence protocol, and extend it to implement the concurrency control scheme implemented in TSX. Regarding the fall-back path, the simulator will aim first to capture the dynamics of a scheme using a single global lock. Whether possible, the simulator will be then be extended to include also other fall-back solutions (e.g., NoREC [12]).

4 Evaluation Methodology

As for evaluation methodology, I will compare how accurately my AM will behave respectively with the results taken from the simulator. The simulator must be able to take workload information from well known benchmarks, e.g., Lee-TM, STMBench7 and Memcached, and then the simulation should output performance measures, e.g., throughput, abort rate and response time. Then, the measured results will be compared with what our model predicts for that same measures.

The simulation model will validate the analytical model. As consequence, an Intel's TSX processor will validate the simulation model, hence, corroborating the correctness of both models.

4.1 Error measurement

In order to compare the model results with the simulation results, various metrics shall be used, including Mean Absolute Error (MAE) and Relative Absolute Error (RAE).

Given a sample of n pairs of predicted/obtained values, the error is defined as $e_i = f_i - y_i$, where f_i is the predicted value (model) and y_i is obtained value (simulation). MAE is defined as $\frac{1}{n}\sum_{i=1}^{n}|e_i|$ and RAE is defined as $\frac{1}{n}\sum_{i=1}^{n}e_i^2$.

Also, the obtained distribution shall be analysed, through the usage of statistical tests, such as Student's t-test and other correlation tests.

5 Calendarization

In this Section follows the calendarization of activities for my dissertation. As presented in Section 3, an analytical and simulation model have to be developed.

I propose the following deadlines for each activity:

25/03/2016: completion of the simulation for the associative caches using the MESIF protocol and the simulation of fall-back synchronization techniques (starting with the global lock); the simulator must be easy to configure, hence, simulating changes in the hardware configuration (e.g., cache size);
15/04/2016: completion of several wrappers to extract statistical information from the benchmarks described in Section 2.5;
29/04/2016: completion of several wrappers to the benchmarks described in Section 2.5 using Intel's TSX and the different fall-back paths developed to the simulator;
13/05/2016: validation of the simulation model with Intel's TSX obtained behaviour and compare the results using the metrics presented in Section 4;
17/06/2016: completion of an initial analytical model;
22/07/2016: validation of the analytical model though the simulation model and compare the results using the metrics presented in Section 4;
02/09/2016: completion of an article describing this dissertation results;
09/09/2016: completion of the dissertation.

In parallel with these activities, further tests with Intel's TSX shall be preformed; in the same line to what is proposed in Section 3.1.

6 Conclusion

This report surveyed the state of the art in Transactional Memory, by explaining its abstraction and implementations in software, hardware, and combinations of both. Given the recent release of best-effort hardware support in Intel processors, some emphasis was given to this new technology, whose details are undisclosed and of the utmost relevance to understand and tune its performance.

As such, the proposal presented here is to create a simulation and an analytical model aimed to capture the performance dynamics of Intel's Restricted Transactional Memory. For this, the main approaches for modelling concurrent systems were presented, ranging from black-box to white-box models.

Though white-box approaches boosted by means of black-box approaches were surveyed, as this work aims to develop a first analytical model for best-effort HTM, hence, filling a gap in current literature. Black-box models will not be an initial concern of this dissertation.

Furthermore, a simulation model will be developed as well, to capture the behaviour of Intel's processors. The reason for developing both models is that the simulator has a twofold purpose: it can allow to assess the impact of future changes in the processor architecture (e.g., larger caches), and it can serve to validate the analytical model.

As a result of this proposal, the expectation is to obtain statistical results similar to those of a real multi-core processor with HTM.

References

1. Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. mar 2008.
2. Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: why is it only a research toy? *Queue*, 6(5):46, 2008.
3. Y. C. Tay, Nathan Goodman, and R. Suri. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4):415–462, 1985.
4. Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987.
5. Philip S. Yu, Daniel M. Dias, and Stephen S. Lavenberg. On the analytical modeling of database concurrency control. *Journal of the ACM*, 40(4):831–872, 1993.
6. C Zilles and R Rajwar. Transactional memory and the birthday paradox. *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 303–304, 2007.
7. Pierangelo Di Sanzo, Bruno Ciciani, Roberto Palmieri, Francesco Quaglia, and Paolo Romano. On the analytical modeling of concurrency control algorithms for Software Transactional Memories: The case of Commit-Time-Locking. *Performance Evaluation*, 69(5):187–205, 2012.
8. Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. *Pact*, pages 3–14, 2014.
9. Rachid Guerraoui. Opacity : A Correctness Condition for Transactional Memory. *Communication*, 2007.
10. Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.
11. Maurice Herlihy, Maurice Herlihy, J Eliot B Moss, and J Eliot B Moss. Transactional memory. *ACM SIGARCH Computer Architecture News*, 21(2):289–300, 1993.
12. Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: streamlining STM by abolishing ownership records. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78, 2010.
13. Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. *Distributed Computing*, 4167:194–208, 2006.
14. Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, 2008.
15. Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-Based Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, dec 2010.
16. Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *ACM SIGPLAN Notices*, 44:155, 2009.
17. Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4):70, 2011.
18. João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

19. C Scott Ananian, Krste Asanovic, Bradley C Kuszmaul, Charles E Leiserson, and Sean Lie. Unbounded Transactional Memory. In *11th Int'l Symp. on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, 2005.
20. Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, and David A Wood. LogTM: Log-based Transactional Memory. pages 1–12, 2006.
21. Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. *Proceedings - International Symposium on Computer Architecture*, 00(C):494–505, 2005.
22. Intel Core, Takuya Nakaike, Matthew Gaudet, and Maged M Michael. Quantitative Comparison of Hardware Transactional Memory for Blue Gene / Q , zEnterprise EC12 ,. *ISCA*, 2015.
23. Andrew Nguyen William Hasenplaugh and Nir Shavit. *Investigation of Hardware Transactional Memory*. PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2015.
24. H Burton Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.
25. Daniel Molka, Daniel Hackenberg, and Wolfgang E Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. 2015.
26. Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.
27. Anu G. Bourgeois and S. Q. Zheng, editors. *Algorithms and Architectures for Parallel Processing*, volume 5022 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
28. Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7. *ACM SIGOPS Operating Systems Review*, 41(3):315, jun 2007.
29. Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: an experience report using GCC and Memcached. *ACM SIGARCH Computer Architecture News*, 42(1):399–399–399–412–412–412, apr 2014.
30. Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun, Cao Minh Chí, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, pages 35–46, 2008.
31. Armin Heindl and Gilles Pokam. An analytic framework for performance modeling of software transactional memory. *Computer Networks*, 53(8):1202–1214, 2009.
32. Diego Didona, Pascal Felber, Derin Harmanci, Paolo Romano, and Jörg Schenker. Identifying the Optimal Level of Parallelism in Transactional Memory Applications. In *Networked Systems*, volume 7853, pages 233–247. 2013.
33. Simon Haykin. Neural Networks: A Comprehensive Foundation. jul 1998.
34. Ingo Steinwart and Andreas Christmann. *Support Vector Machines*. 2008.
35. Márcio Castro, Luis Fabrício Wanderley Góes, Christiane Pousa Ribeiro, Murray Cole, Marcelo Cintra, and Jean François Méhaut. A machine learning-based approach for thread mapping on transactional memory applications. *18th International Conference on High Performance Computing, HiPC 2011*, 2011.
36. Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Machine learning-based self-adjusting concurrency in software transactional memory systems. *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2012*, pages 278–285, 2012.

37. Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional Auto Scaler: Elastic Scaling of In-memory Transactional Data Grids. *ACM Transactions on Autonomous and Adaptive Systems*, 9(2):125–134, 2014.
38. Diego Didona and Paolo Romano. Performance Modelling of Partially Replicated In-Memory Transactional Stores. In *2014 IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 265–274. IEEE, sep 2014.
39. Diego Didona, Francesco Quaglia, Paolo Romano, and Ennio Torre. Enhancing Performance Prediction Robustness by Combining Analytical Modeling and Machine Learning. *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2015.
40. Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Analytical/ML mixed approach for concurrency regulation in software transactional memory. *Proceedings - 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2014*, pages 81–91, 2014.
41. Diego Didona and Paolo Romano. On Bootstrapping Machine Learning Performance Predictors via Analytical Models. In *ICPADS*, 2015.
42. Pascal Felber, Christof Fetzer, Rachid Guerraoui, and Tim Harris. Transactions are back—but are they the same? *ACM SIGACT News*, 39(1):47, 2008.
43. Diego Rughetti, Pierangelo Di Sanzo, Alessandro Pellegrini, Bruno Ciciani, and Francesco Quaglia. Tuning the Level of Concurrency in Software Transactional Memory : An Overview of Recent Analytical , Machine Learning and Mixed Approaches. pages 395–417, 2015.
44. Daniel F Garcia. Performance Modeling and Simulation of Database Servers. *The Online Journal on Electronics and Electrical Engineering (OJEEE)*, 2(1):183–188.
45. O. Ulusoy and G.G. Belford. A simulation model for distributed real-time database systems. In *Proceedings. 25th Annual Simulation Symposium*, pages 232–240. IEEE Comput. Soc. Press.
46. U. Herzog and M. Paterok, editors. *Messung, Modellierung und Bewertung von Rechensystemen*, volume 154 of *Informatik-Fachberichte*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.
47. Dave Christie, Jae-Woong Chung, Stephan Disetelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Trovald Riegel, Pascal Felber, Patrick Marlier, and Etienne Riviere. Evaluation of AMD ' s Advanced Synchronization Facility Within a Complete Transactional Memory Stack. *EuroSys 2010*, pages 27–40, 2010.
48. Matt T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 23–34, 2007.
49. Nuno Diegues and Paolo Romano. Self-Tuning Intel Transactional Synchronization Extensions. *ICAC*, pages 1–11, 2014.
50. Stuart J. Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. feb 2003.
51. Bhavishya Goel, Ruben Titos-Gil, Anurag Negi, Sally A. McKee, and Per Stenstrom. Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell. In IEEE, editor, *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 615–624, Phoenix, AZ, may 2014. IEEE.
52. Alessandro Pellegrini and Francesco Quaglia. The ROme OpTimistic Simulator: A Tutorial. In *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*, pages 501–512. 2014.