Performance Modelling of Hardware Transactional Memory

Daniel Filipe Salvador de Castro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: **Paolo Romano**

Examination Committee

Chairperson:	Prof. José Carlos Martins Delgado
Supervisor:	Prof. Paolo Romano
Member of the Committee:	Prof. Aleksandar Ilic

October 2016

Acknowledgements

I would like to thank my supervisor Paolo Romano and my research colleague Diego Didona for all the support and in the development of the analytical model. Also, thanks for all the suggestions when the hardware, simulator and model did not match well. Without them, writing this dissertation would not be possible.

Abstract

Transactional Memory (TM) is a recent alternative to traditional lock based synchronization mechanisms for parallel programming. Our analysis of existing literature in these areas highlights the existence of a relevant gap, which we aim to fill with this dissertation: the lack of performance models for hardware-based implementations of TM, also known as Hardware Transactional Memory (HTM). In order to monetize all the available transistors in a modern processor, HTM is usually build on top of the existing cache coherency protocols, whose dynamics we capture in the presented simulative and analytical models. Both models capture the empirically observed conflict and capacity detection dynamics observed in Intel's implementation of HTM. Moreover, the simulation model predicts, with little discrepancies, the probability of abort and throughput. Subsequently, the analytical model is validated against this simulation, with an average error of 1.69% and 4.07% regarding probability of abort and throughput, respectively.

Keywords: transactional memory, hardware, performance modeling, concurrency control

Resumo

A Memória Transacional é uma alternativa recente à utilização da tradicional sincronização com exclusão mútua em programação paralela. A nossa análise sobre a literatura existente averiguou que existe uma falha relevante nesta área, a qual nós pretendemos preencher com a presente dissertação: a falta de modelos de desempenho para implementações em hardware da Memória Transacional, também conhecido como Memória Transacional em Hardware. De forma rentabilizar todos os transístores existentes nos processadores modernos, normalmente a Memória Transacional em Hardware é desenvolvida tendo por base os já existentes protocolos de coerência das caches. Nós capturamos esse dinamismo nos nossos modelos analítico e simulativo. Mais em detalhe, a simulação prevê, com pouca descrepância relativamente ao sistema real, a probabilidade de abortar e quantidade de transações em hardware completas por unidade de tempo. Consequentemente, tendo em conta o modelo simulativo, validamos o modelo analítico com erros médios na ordem dos 1.69% e 4.07%, respectivamente, para a probabilidade de abortar e taxa de transações completas.

Palavras-chave: memória transacional, hardware, modelação de desempenho, controlo de concurrência

Contents

1 Introduction **5**

2 Related Work **6**
2.1 Transactional Memory 7
2.2 Software Transactional Memory 8
2.3 Hardware Transactional Memory 10
2.3.1 Unbounded Hardware Transactional Memory. 10
2.3.2 Best-effort HTM. 11
2.4 Hybrid Transactional Memory 12
2.5 Benchmarks for Transactional Memory 13
2.6 Comparison between STM, HTM and HyTM 14
2.7 Basic methodologies for performance modeling 14
2.7.1 White-box models. 15
2.7.2 Black-box models. 16
2.7.3 Gray-box models. 16
2.8 Performance modeling of transactional systems 17
2.8.1 Database Management System models. 17
2.8.2 TM models. 18
2.9 Simulation models for transactional systems 20
2.9.1 Simulators for DBMS. 20
2.9.2 TM simulators. 20
2.10 Tuning of TM 21

3 Study on Intel TSX **21**
3.1 Investigation on the design of Intel TSX 21
3.2 Concurrency control in TSX 22
3.2.1 Explanation for conflict detection with MESI protocol . . 23
3.3 Capacities exceptions 25
3.3.1 Capacity of a set 26
3.3.2 Theoretical capacity for a transaction 27
3.3.3 Random access patterns 30
3.3.4 Study on the preoccupied cache granules 30

4 TSX with fall-back to global lock **32**

5 Analytical model **34**
5.1 System model and notation 35
5.2 Modeling the system as a Markov chain 36
5.2.1 Brief Introduction of Markov chains 37
5.2.2 Solving the Markov chain 37
5.2.3 The case of HTM 38
5.3 Modeling the conflict dynamics 40

5.4 Modeling the probability of abort . . . 42
5.4.1 Adjusted probability of abort . . . 43
5.5 Modeling the response time . . . 44
5.5.1 Adjusted response time . . . 45
5.6 Modeling the Markov chain transition rates . . . 46
5.7 Model of cache capacity . . . 47

6 Simulation model **49**
6.1 Modeled behavior assumptions . . . 49
6.1.1 Concurrency control . . . 50
6.1.2 Capacity exceptions . . . 50
6.1.3 Additional assumptions . . . 50
6.2 ROOT simulator . . . 51
6.3 Simulation internals . . . 52
6.3.1 Enabling capacity exceptions . . . 55

7 Validation study **55**
7.1 Analytical model validation . . . 56
7.1.1 Conflict exceptions . . . 56
7.1.2 Capacity exceptions . . . 57
7.2 Simulive model validation . . . 60
7.2.1 Conflict exceptions . . . 60

8 Conclusion **62**
8.1 Future work . . . 63

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].

Transactional Memory (TM) has emerged as a simpler, and hence more attractive, alternative to lock-based synchronization: unlike mutual exclusion, 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 [2, 3].

On the one hand, 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. On the other hand, Dragojević et al. [3] argues that TM is on its early days and its performance should increase in time, also presented scenarios where STM performs better than other concurrency control approaches.

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, which makes it even harder to predict how an application will perform when using HTM as its synchronization mechanism.

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

1. First a discrete event based simulator is developed. The simulator is focused on capturing the dynamics of the concurrency control mechanism employed by commercial HTM implementations, such as those included in

Intel's and IBM's CPUs [4, 5]. As a preliminary step to achieve this goal, we designed a set of experiments aimed to clarify the internal dynamics of these HTM implementations and to provide sufficient information to develop a white-box simulation model. The results of this "reverse engineering" study can actually be useful in a broader context that this dissertation, as the quantitative (e.g., effective HTM capacity in presence of read vs write intensive workloads) and qualitative (e.g., model of the concurrency control algorithm implemented in hardware) information that we present can be valuable to application developers or to engineers of other HTM performance models.

2. Next an analytical model is developed. The analytical model uses queuing theory and probabilistic techniques, extending prior literature in the area of performance modeling of concurrency control algorithms in STMs and Database Management Systems (DBMSs) literature [6, 7, 8, 9, 10]. The analytical model is tested across a broad number of workloads showing errors in throughput estimation of 4.07% (in average).

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. Section 3 discusses our assumptions on Intel's HTM architecture and presents studies on Haswell machines. With the gathered data from Intel, we should be finally comfortable on our assumptions of the system. Section 6 presents the simulation model. The analytical model is presented in Section 5. In Section 7, the simulation is validated against the real system and the analytical model against the simulation, by means of scatter plots and the computation of the obtained errors. Finally, Section 8 summarizes this dissertation with some final conclusions. Future work, and possible upgrades to the model, is presented in Section 8.1.

2 Related Work

During the past decade there has been 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 HTM 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 TM libraries. In Section 2.6, HTM, STM and HyTM are compared with each other. The focus of the subsequent sections is on the modeling of performance of transactional systems. In Section 2.7 black-box and white-box techniques are surveyed. Approaches based on analytical modeling 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 [11].

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 [12]. 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 [13]); 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, variable a is never be less than 0. The call to `__transaction_atomic` to identify the critical section is the only change regarding the non-synchronized version. The advantage when compared to a global lock alternative is that transactional code blocks can run truly in parallel and the locked critical section must be executed sequentially.

In order 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 compiler, within `__transaction_atomic`, translates the memory accesses with respect to the provided library, which can be STM, HTM or a hybrid solution.

Program 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) [14, 9, 2]. Nevertheless, in Dalessandro et al. [15], 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 [12], 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) [16], TinySTM [17, 18] and SwissTM [19, 3] 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, 3], and there are contradictory results. While in Cascaval et al. [2] TM is harshly criticised; in Dragojevic et al. [3], 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 [20]. 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 [21, 22, 23], 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 [24]. 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).

Due to the best-effort nature of HTM, debugging/profiling transactions that use this concurrency control mechanism become a hard task, since breakpoints within transactions necessarily abort it due to context switches and time interrupts. Despite of this fact, there are some studies on profiling HTM, more precisely Intel Transactional Synchronization Extensions (TSX) implementation [57].

2.3.1 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 [21, 22, 23], but its efficient implementation introduces further complexity in hardware.

A first approach of Unbounded Transactional Memory (UTM), was the Large Transactional Memory (LTM) [21], 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) [23] 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 [22], 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.

2.3.2 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 [24, 25], 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. [25], 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 [26]) 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 ($400lines \times 64B = 25600B$). 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. [27] discuss how the Modified, Exclusive, Shared, Invalid, Forward (MESIF) protocol was used to support TSX. This protocol work for machines with more than one processors, i.e., multi-socket machines, due to the extra Forward state. In MESIF, each cache line is in one of the following states:

Modified: the cache line of the current cache is the only existing copy, and its value may not match main memory (was recently updated);

Exclusive: the cache line of the current cache is the only copy in all caches, and its value matches the main memory;

Shared: other private caches have copies of 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) [28] 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. [28] have shown 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 [11], 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 [29];

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 [30];

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 [31];

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 [32].

2.6 Comparison between STM, HTM and HyTM

As discussed in the previous sections, STM, HTM and HyTM have limitations, which we 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 [16]);

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 modeling

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.

2.7.1 White-box models.

Like analytical and simulative models, these 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.

While black-box need to be fed with large learning sets, white-box models, due to its nature, once provided the input set, it compute the output set without further preparation steps. And, by looking into the provided equations, it is possible to understand why the predictions are being made, i.e., easier to debug.

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. [33, 10]

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 [6, 7, 8, 33, 34, 10]. 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.

2.7.2 Black-box models.

Black-box modeling 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 [35] and support vector machines [36].

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., [37, 38].

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.

2.7.3 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 modeling, and apply some ML technique to those remaining; finally devise a formula to combine the outputs of each modeling of each subsystem [39, 40];
- 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 [41, 42];
- combine AM and ML using other Machine Learning techniques to learn when one model outperforms the other [43].

2.8 Performance modeling of transactional systems

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

2.8.1 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 [6, 7, 8]. While Tay et al. [6] and Yu et al. [8] propose analytical models, in Agrawal et al. [7], 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 [12].

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 [8].

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 [8]:

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

And for optimistic concurrency control mechanisms:

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

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 [8], accurately predict the real system performance, and the obtained deviation scaled linearly with the input domain.

2.8.2 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 modeling of TM systems.

In particular, the only works that we are aware of that target performance modeling 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 modeling 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 modeling of STM, Zilles et al. [9] 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:

$$\begin{aligned} \text{conflict} &\propto \frac{\text{footprint}^2}{\text{table\_size}} \\ &\propto \text{concurrency}^2 \end{aligned}$$

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 [33, 34, 10]. 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 [33]. 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 [44] 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 [41] 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 2.3.1). Hence, modeling the performance of these systems depends on how accurate the simulator is.

2.9.1 Simulators for DBMS.

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

1. validate some AM of specific components of a DBMS. This is the case for instance for the work by Tay et al. [6] and Yu et al. [8], which used simulation to validate AM of the performance of various concurrency control schemes.

2. predict the database performance trends as a whole [45], or shedding light on the performance dynamics of a DBMS specific modules (e.g., real-time transactional scheduling [46] or buffer management [47]).

2.9.2 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 2.3.1).

AMD has an experimental HTM instruction set, called Advanced Synchronization Facility (ASF) [48], 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 [32] 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 [49] 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. [50], lightweight reinforcement learning techniques (i.e., hill climbing [51], an on-line black-box modeling 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 Study on Intel TSX

In this chapter, we present the available information on Haswell processors. Section 3.1 presents the specifications of our test processor and data from previous studies on TSX. In order to better understand its internals, further experiments with TSX are presented in Sections 3.2 and 3.3. Section 3.2 is dedicated to the study of conflict detection, and in Section 3.3 our main focus is on capacity.

3.1 Investigation on the design of Intel TSX

All tests were executed in an Intel Xeon E3-1275 v3 @ 3.50GHz. This CPU has 4 cores and have Hyper-Threading enabled, thus it should be able of running up to 8 threads concurrently, by sharing resources in the physical cores. The size of L3 cache level is 8192 KB, cache line have size of 64 bytes. Moreover, the address space size of 39 bits physical and 48 bits virtual. As presented in table Table 1, L1 has 512 cache lines of capacity, and L2 has 4096 cache lines.

Intel TSX capacity limits rely on the cache size [52, 24, 25]. Table 1 presents the specifications for a Haswell processor. Similar specifications are found in Broadwell and Skylake processor families.

Intel TSX comes with two APIs. The first one is Hardware Lock Elision (HLE). It allows the programmer to upgrade the already existing lock-based applications to use optimistic concurrency control mechanism. This is done by

Level	Capacity	Line size	Fastest Latency	Peak Bandwidth (Bytes/Cycle)	Update policy
L1	64 sets (8-way)	64B	4 cycles	64 (Load), 32 (Store)	Writeback
L2	512 sets (8-way)	64B	11 cycles	64	Writeback
L3	varies	64B	34 cycles	varies	Writeback

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

means of an assembly prefix in the lock operation (usually a compare-and-swap operation). Then, other assembly prefix is introduced for the release of the lock. The critical section avoids to acquire the lock, if it conflicts, then, it restarts and effectively acquire the lock.

The second API is Restricted Transactional Memory (RTM). This is the API that we are evaluating in this dissertation. Differently from HLE, begin and commit operations must be explicitly called.

3.2 Concurrency control in TSX

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. The conflict results are shown in Table 2. Each of the test cases 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.

We setup the experiment with two concurrent threads τ and τ', running concurrently two transactions, T and T', respectively. Granules are acquired, with the given schedule presented in Figure 2.

This experiment to work, transactions must run without any other source of contention, but the target conflicting granule. Spinning times enforce the desired schedules. Assuming that both threads start concurrently at the same time, if thread τ acquires the granule and then spin a carefully calculated amount of time, it is possible, within some margin of error, to synchronize both τ and τ' when accessing the granule.

Spinning times are simple empty `for` loops, where a variable is incremented up to the given value. However, nowadays compilers drop this type of code that do not compute anything. A way of optimize it is to set the incrementing variable immediately to the last value. Fortunately, GCC allows the programmer to specify a flag (-O0) that deactivates this kind of optimization.

Other types of abort that are not conflicts include the context switch of our

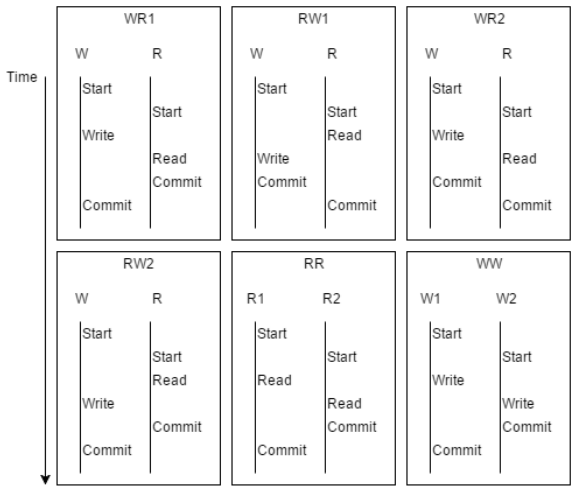


Figure 1: Tested schedules in Intel TSX.

	R	**W**
WR1	C	A
RW1	A	C
WR2	C	A
RW2	A	C

	R1	**R2**
RR	C	C

	W1	**W2**
WW	A	C

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

current process in the machine (i.e., our application is not the only one running in the machine). Also, if τ and τ' are not well aligned, thus not executing the desired schedule, the outcome of the experiment is not the correct one.

Despite of the previous sources of errors, carefully tuning the spinning times, and taking a large sample of over 50 000 commit/abort measurements for each schedule, we are confident in the results presented in Table 2. The commit/abort situations occur over 95% in the obtained samples for all presented schedules.

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

This experimental knowledge is crucial to devise the simulative and analytical models, which are presented in Sections 6 and 5, respectively. Also, TSX conflict detection can be explained by the behavior of the Modified, Exclusive, Shared, Invalid (MESI) protocol (see Section 3.2.1). Apparently, the last transaction to access the granule wins. The transaction that acquires the granule g in last place invalidates the other copies of g.

3.2.1 Explanation for conflict detection with MESI protocol

As previously presented, Intel TSX eagerly detects conflicts, and we assumed this behavior is caused due to the MESI invalidation process. Such a experiment led to the results in Table 2, which are intuitive regarding the nature of cache coherence protocols. This section provides extra details on how the conflict detecting might be designed in TSX.

Each physical core has a private cache. An access to a generic granule g necessarily makes the private cache request the value to higher levels of cache, with more capacity, but slower. If the value is in none of the cache levels, then, the value must be read from main memory.

MESI assumes multiple caches that have copies of existing granules in main memory. Those copies are stored in cache lines. Each line is in one of the four following states: i) (M)odified; ii) (E)xclusive; iii) (S)hared; or iv) (I)nvalid.

A cache line l_i in state M indicates that a given cache c_i is the only one that has a copy of the modified line. No other cache c_j has a valid copy of that line. On setting a line to state M, all other copies, in the remaining caches, must be invalidated.

A cache line l_i in state E also indicates that only one cache c_i has a copy of that line. The granule have been requested to read, thus, it has the same value as in main memory. No other cache c_j has a valid copy of that line.

In state S, the line l is present in multiple caches. The most recent cache to request the granule copied the value from other private cache. If the copied l was in state M or E, then, it is copied to the requester cache, and the state of both copies is changed to S.

State I indicates that a line l_i is not valid. The invalidation usually occurs when other cache c_j modifies some copy that c_i contains.

To illustrate the MESI behavior, assume an example with 4 caches, i.e., c_1, c_2, c_3 and c_4. Assume the state of a granule in the caches s_g: $[I, I, I, I]$. In this state, a copy of the value of granule g is not present in any of the caches. At a given instant, c_1 reads g. Now the state is $s_g = [E, I, I, I]$. Then, c_3 reads g. The state becomes $s_g = [S, I, S, I]$.

Assume, for instance, that c_2 writes g, the copies of g in the other caches must be invalidated, i.e., $s_g = [I, M, I, I]$. On a future read produced by c_4, the state changes to $s_g = [I, S, I, S]$, as c_2 shares its most recently modified copy.

When a generic transaction T starts, all accesses in the cache mark the accessed line as "transactional". If a transactional line is invalidated, then T aborts due to a conflict exception. If a set is full and a transactional line is evicted, then, T aborts due to capacity exception.

Program 2: Experiment of the conflict detection schedule Read after Write.

```
1  type_t a = 2;
2
3  void T1() {
4    status = _xbegin();
5
6    check_transaction(status);
7    for(i = 0; i < 1000; ++i);
8    a = 10; // changes 2 to 10
9    for(i = 0; i < 2000; ++i);
10   _xend();
```

```
11 }
12
13 void T2() {
14   type_t b;
15
16   check_transaction(status);
17   for(i = 0; i < 2000; ++i);
18   b = a; // tries to read T1 written value
19   // T1 aborts and T2 reads the value 2
20   for(i = 0; i < 1000; ++i);
21   _xend();
22 }
23
24 main() {
25   start_T1();
26   start_T2();
27
28   join_T1();
29   join_T2();
30 }
```

According to MESI, we are forced to assume that Non-Transactional Code Blocks (NTCBs) interact with Transactional Code Blocks (TCBs). If a NTCB invalidates some granule in the read/write set of a TCB, it aborts. Algorithm 3 in Section 4 relies on this behavior to abort all the active transactions when the lock is taken (Line 14). The lock is added to the read set of all transactions in Line 14 and in Line 27 the lock is written by a non-transaction, thus, aborting all active transactions.

Assume now the schedule for the write(W)/read(R) conflict as is presented in Program 2. T_1 acquires a for the write set (writes $a = 2$), then, T_2 acquired a for its read set. Note that T_1 and T_2 run concurrently. This access pattern issues an invalidation request to the private cache of T_1.

According to MESI, in the state of a starts as $s_a = [I, I]$. Then, T_1 writes setting the state to $s_a = [M, I]$. In a NTCB, T_1 would share its copy of a with T_2. But, since it is a TCB, this copy is invalidated instead, i.e., $s_a = [I, E]$. This process effectively rolls-back undesired modifications.

The invalidation process occurs in the following situations: T_1 reads a then T_2 writes a, T_1 writes a then T_2 reads a (the situation in Program 2), T_1 writes a then T_2 writes a, i.e., R/W, W/R, W/W, respectively.

3.3 Capacities exceptions

TSX has limited capacity, and, therefore, if a transaction T access more memory than the cache capacity, T experiences a capacity exception.

This section addresses the impact of application memory access patterns on the probability of having a capacity exception after the i-th operation, i.e., the effective capacity of the cache when used in the context of a hardware transaction. The geometry of our study Haswell processor is already presented in

Section 3.1, and we will use this information when devising theoretical capacity for this processor.

We start by validating the set associativity of the L1 cache in Section 3.3.1, where we also discover the minimum capacity a transaction may have in presence of very specific access patterns. Then, in Section 3.3.2 we stress the maximum capacity by experimenting contiguous vs strided access patterns. Finally, in Section 3.3.3, we experiment random access patterns, with mixed read and writes, and present the average capacity for such workloads.

3.3.1 Capacity of a set

Assume the address for a granule as composed by a number of bits for the tag, set and cache line. Testing some access patterns and knowing that a cache lines is 64 bytes long, i.e., needs 6 bits to be addressed, the set bits are in range 6-11. Each set has a given number of slots (i.e., the n-way set associativity).

Knowing the geometry of L1 cache for our experiment processor, we designed an experiment to stress the capacity of its cache.

The experiment in Figure 2 shows 9 test cases, i.e., for 2 granules up to 10, each of it is repeated 100 000 times to get a sample. As the address location may have some impact in the location of the metadata/"polluted" granules, the program reruns 5 times in order to obtain different ranges of addresses. For $n \leq 8$ the commit probability is greater than 90%, but for $n > 8$ the probability of abort is nearly 100%. The workload is write only.

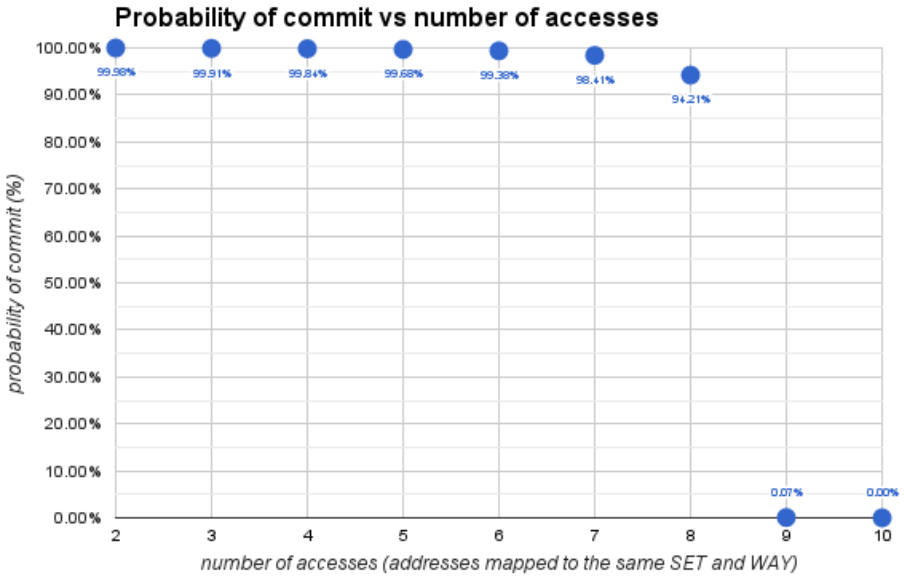


Figure 2: Probability of commit given that we access granules that map the same set and way set associative slot in the L1 cache, i.e., stride 512.

First, a struct of the size of the cache line is created (i.e., 64 bytes). GCC allows the declaration of some alignment directives in order to effectively load the struct into the whole cache line. Then, a large array of this struct is allocated using `malloc`.

Since L1 has 64 sets, a struct at address a is mapped to the same line as a struct in address $a + 64 \times 64B$. In the array with contiguous granules, the granule at index 0 maps to same cache line as the granule in index 64, 128, 192, and so on. Basically, assume indices Y and W, both map to the same set if $Y = W \pmod{64}$.

Our study processor has 8-way set associative caches. Thus, it is guaranteed that the 9th access that maps to the same set produces a capacity exception.

Figure 2 presents the results for this experiment. It tests the full n-way set associativity of Intel cache and also stresses the maximum capacity of each set. Intel provides fully associative caches, i.e., when a granule is mapped into a set, the hardware finds a empty slot for it, or evicts the slot with the oldest granule. This means that accessing memory with stride 64, 128, 192, 256, etc, i.e., the workload capacity is always 8. Assuming bits in range 12-14 map the slot within a set, a stride 512 workload should map the granules to the same set and slot.

Although the caches are 8-way set associative, it is not guaranteed that a capacity exception does not occur before exhausting the cache resources. TSX probably stores some more data that is not directly related with the provided TCB, or that the programmer is not aware, as, for example, written granules due to local variables and function calls. Thus, the private cache is not completely "empty" when the transaction starts.

However, this capacity is only valid for write accesses. We did not discover how Intel is storing the read set in cache. A transaction is, probably, capable of storing its read set in L2 and/or L3 caches, which allows the transaction to have a much larger read set than the write set.

This section concludes with the following conclusions: i) as reported also in other studies [25], our experiment confirm that writes are stored in L1 in Intel's CPUs; ii) as expected cache geometry has an impact: abort as soon as a set is full, and one cache line has to be evicted; iii) effective cache capacity is lower than expected. Hypothesis is that part of the cache is used up by other information, which has been stored transparently to user level code either by the hardware (e.g., space reserved to store transaction metadata, like address of abort handler or status bits on the transaction) or by the compiler (e.g., extra read/writes to memory, future studies on the transaction code disassemble are proposed in Section 8.1).

3.3.2 Theoretical capacity for a transaction

In the previous section, we experimented for a stride multiple of 64 to match the same set. Now we will experiment for a different experiment where the maximum number of granules that Intel TSX is capable of access is stressed.

Stride	Theoretical maximum	Maximum accesses (writes)	Maximum accesses (reads)
1	512	453	≈520
2	256	227	≈3000
3	512	466	≈480
4	128	127	≈15000
8	64	64	≈7000
10	256	236	≈20000
12	128	127	≈17000
16	32	32	≈4000

Table 3: Intel Xeon E3-1275v3 maximum capacity accessing contiguous memory, with the offset given in the first column. The theoretical maximum assumes the geometry of 64 sets 8-way set associative, and all cache lines are available for the transaction.

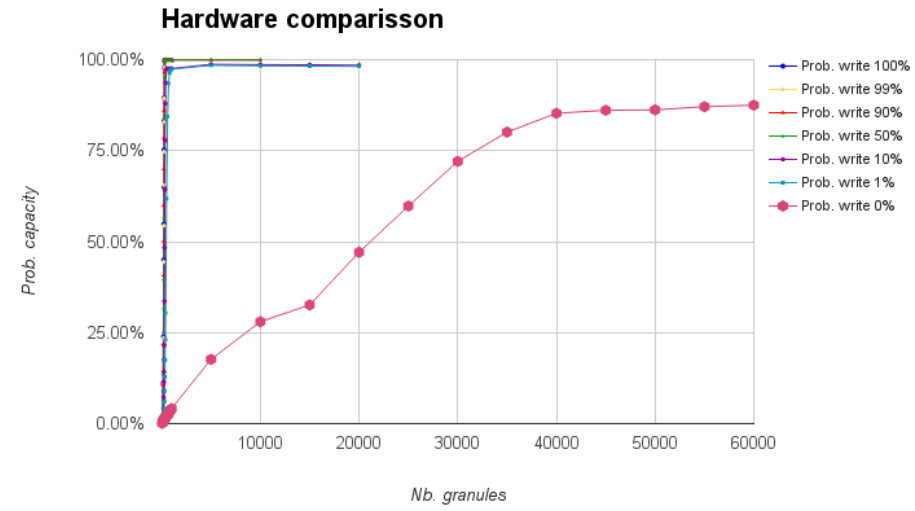
With stride 1 the maximum capacity should be reached. For stride 2 the maximum capacity is halved. For stride 4 the capacity is halved again, until stride 64 is reached, where the capacity is always 8. Basically, if the stride is 1, then memory is accessed contiguously, if the stride is 2, then memory is read in intervals of 2, e.g., 0, 2, 4 and 6.

The granules are accessed from a granule pool, which is allocated using `malloc`. As in the stride 64 experiment, granules are of the size of a cache line and are correctly aligned in memory.

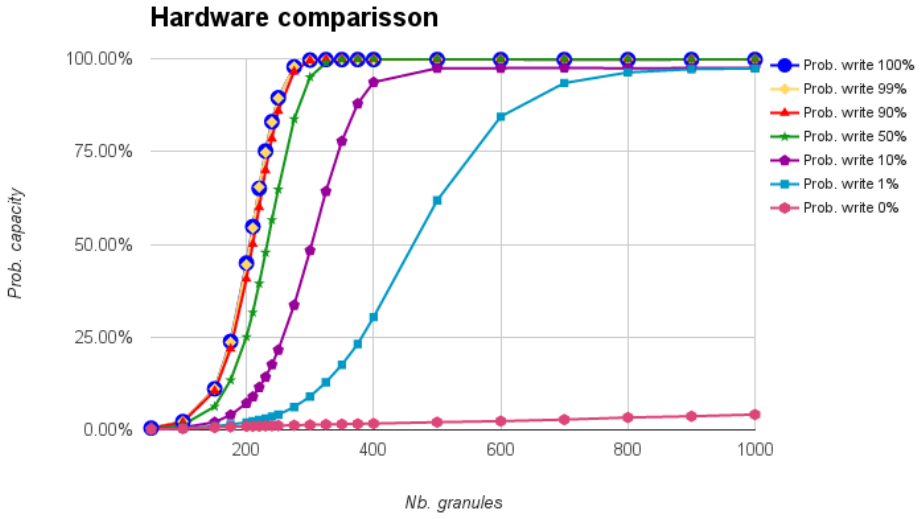
Table 3 shows the results of this experiment. The granules are accessed with the given offset in the first column, i.e., the stride. An obvious conclusion is that, with stride powers of 2, doubling the stride halves the maximum acquisition capacity, which matches the theoretical capacity expectations.

Due to some possible in-memory "polluted" granules, e.g., the transaction abort handler, and other accesses that might be done transparently from the programmer, i.e., accessing local variables, pushing arguments into the stack when calling a function, etc. The theoretical capacities are unlikely to be reached.

As for the read only pattern, it is hard to speculate how Intel is storing the read set. The maximum capacities change considerably from run to run. One explanation for that behavior, is the location of the memory pool is relevant for the mapping algorithm in L3 cache, since this is the only thing that changes in different runs. Also, there is no correlation in the strides values and the theoretical values for L1 cache. In the end of the next section there is some discussion on how to model the read set.



(a) Tested workloads are 100%, 99%, 90%, 50%, 10%, 1% and 0% writes.



(b) 3a amplified from 0 to 1000 granules for better visibility.

Figure 3: Random access workloads.

3.3.3 Random access patterns

The previous experience assumed contiguous lines, now we present other experiment where granules are acquired randomly from the previously presented granule pool.

In this experiment, to ensure that the transaction only accesses the given number of granules, prior to start the transaction, the memory pool is populated, not with random values, but with a path.

For example, the value 53 is randomly generated. In granule with index 53, the index of the next granule is written, e.g., 27. When the transaction starts, it reads the starting position from some general purpose non-volatile register R_1 (in order to reduce unnecessary memory accesses to a minimum). Then, it reads the value at the given index. The new read index is stored in R_1. Now R_1 contains the index of the next granule, and we repeat the iteration. To store the number of access we use other register variable, when this register reaches 0 the workload ends. To simulate a write the read value is written back in other part of the same granule (i.e. the granule is 64 byte long, but we are only explicitly using two integers of size 8).

10000 different random paths in the memory pool were tested, each path is executed in the same run 5000 (i.e., 5000 transactions per run).

With write probability of $P_W = 10\%$, the transaction cannot acquire more than 300 cache lines, but with $P_W = 0\%$, it is possible to access up to 40 000 cache lines, see Figure 3. Also, in the read only workload, the transaction eventually gets so large that, after the 40 000 cache lines mark, the probability of commit is 0%. The transaction aborts due to unspecified cause 10% of the samples, and due to capacity almost 90% of the samples.

Reads were reported to be kept in L3 and influence the maximum number of writes [25]. Our experiments with random access patterns, mixed read and writes, also suggest the same conclusions.

For example, in Figure 3, for the 50% writes workload, accessing 250 granules has around 50% chance of capacity exception, but 125 writes (half of the 250 accesses), in a write only workload, have less than 10% chance of capacity. This is expected as when reading cache lines need to be brought in L1, and may cause the eviction of written values from L1.

One can try to model this interference by assuming that each read consumes some space in L1, reducing the effective capacity for writes. Results plotted in Figure 4 show that, for a given workload, the capacity reduction is more or less constant, thus, it may be modeled as a function of the percentage of writes in the workload.

3.3.4 Study on the preoccupied cache granules

As pointed in previous sections, the theoretic capacity for write only workloads is never reached. We conclude that there might be some "polluted" granules in

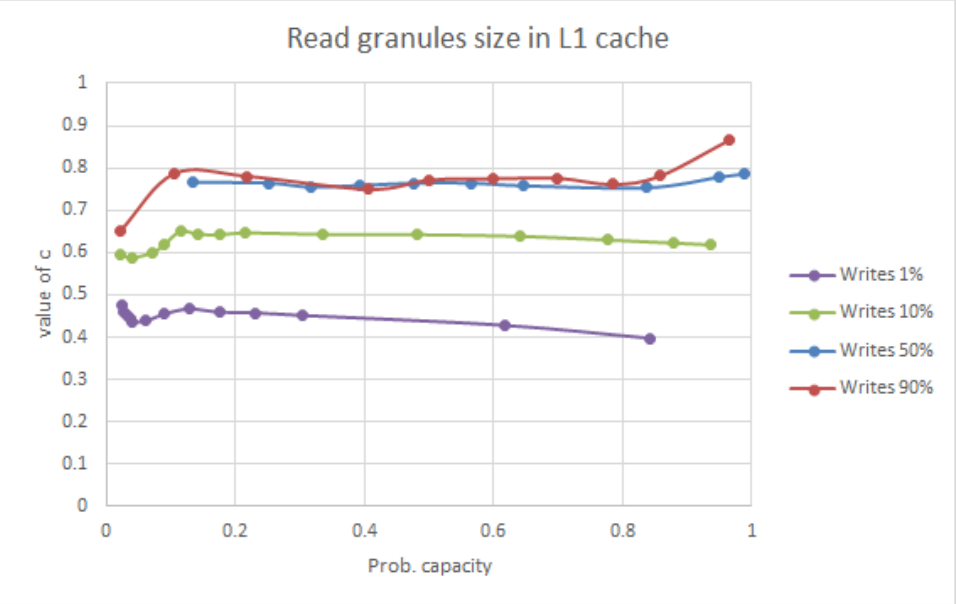


Figure 4: Read granules eviction might be modeled as their interference in L1 capacity is smaller than the interference of write granules. Such that $g_W = 1$ and $g_R = c \times g_W$, $0 < c < 1$.

L1 cache that, transparently, from the programmer cannot be used.

With the help of a simulation, we mounted a experiment where the L1 cache geometry is tested. Assume Least Recently Used (LRU) eviction policy and the mapping process as in the previous experiments (i.e., bits 6-11 to map the set).

Then, granules are randomly acquired from the granule pool, as we did to stress capacity in Section 3.3.3. The same approach is done in the simulation, granules are pick at random and put in the respective set, when a full set receive a granule, the transaction aborts.

To simulate the "polluted" granules. We provide the simulator a number of granules that are put in the cache contiguously before the transaction starts. This effectively reduces the maximum capacity, since some slots are occupied.

The cache geometry is 64 sets 8-way set associative, and the workloads are write only, i.e., $P_W = 1$, because we know that reads are managed differently. The number of tested "polluted" granules is 20, 30, 40 and 50.

The experiment in hardware is the same as presented in the previous section for the write only workload.

We obtained the plot in Figure 5. Which indicates that around 30 cache lines are in use when the transaction starts.

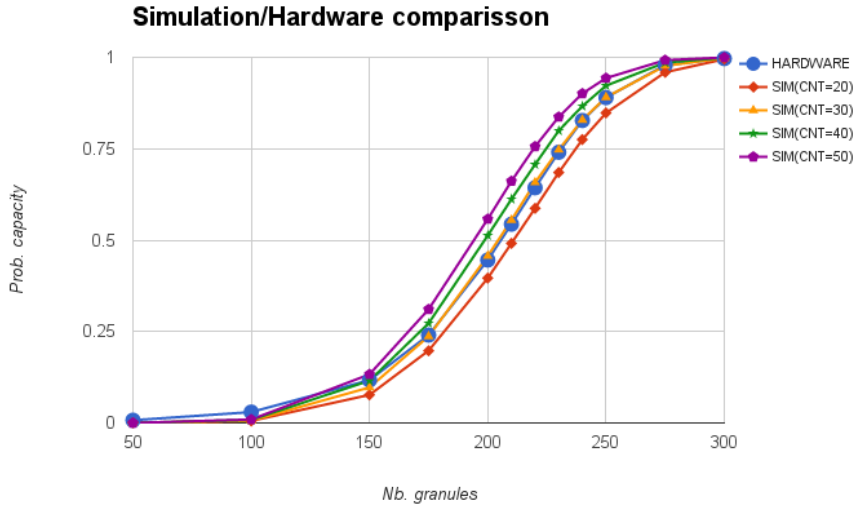


Figure 5: "polluted" granules stored contiguously.

4 TSX with fall-back to global lock

This section describes an algorithm (Algorithm 3) used in best-effort HTM. When transactions are not capable of completing in the normal speculative path, they fall-back to a software path. The software path studied in this dissertation is a global lock solution, where each fall-back transaction runs serially. This algorithm is the base to build the analytical model presented in chapter 5, and event-based simulation presented in chapter 6.

This approach assumes that each transaction starts with a given number of retries, i.e., its budget, and when the budget exhaust, then, the global lock (i.e., `sgl`) must be acquired.

Algorithm 3 uses the RTM API available in Intel TSX, but it should be easy enough to extrapolate it to other HTM solutions as, for example, the HTM implementation provided by IBM P8 processors.

In order to start a transaction T the routine `BeginXact` (Line 4) must be called, it deals transparently from the programmer the TSX error handling and the budget update process.

RTM requires a call to the commit routine in order to finish the transaction. This is done in `EndXact` (Line 36) transparently from the programmer as well. It also handles the serialized transactions by releasing the global lock.

The `sgl` is tested twice, once immediately before starting the transaction (Line 6), and again within the transaction (Line 14). This double checking procedure avoids the lemming effect, which makes transactions exhaust their budget unnecessarily.

Note that there is a data race condition in the `sgl`. For example, assume three transactions, T_1, T_2 and T_3. T_1 finishes executing a serialized TCB. T_2 was waiting the lock to be free, but still has budget left. T_3 was waiting the lock to execute a serialized TCB (i.e., its budget is 0). If T_2 advances before T_3 and starts a transactions, it aborts unnecessarily and wastes 1 budget, i.e., all transactions in the fall-back may not be scheduled sequentially.

Algorithm 3: Preparation for the execution of a transaction in TSX.

```
1  int budget = INIT_BUDGET
2  int roaPolicy = {LINEAR, HALF, ZERO}
3
4  void BeginXact() {
5    while(1) {
6      while(sgl == 0) {
7        // Avoid lemming effect
8        asm("pause;");
9      }
10     int status = _xbegin(); // Start hardware transaction
11     // xact re-starts from here
12     if(status == XBEGIN_STARTED) {
13       // Check the sgl and put it in the read-set
14       if(IS_LOCKED(sgl)) {
15         // Some xact has acquired the sgl: abort
16         _xabort(30);
17       }
18       break;
19       if(status == XABORT_CAPACITY) {
20         budget = SPEND_BUDGET(budget);
21       } else {
22         // Abort due to conflict or acquisition of the sgl
23         tries--;
24       }
25       if(tries <= 0) {
26         // Budget expired: fall-back to global lock
27         while(CAS(sgl, 0, 1) == 1) {
28           asm("pause;"); // Wait for the sgl
29         }
30         break;
31       }
32     }
33   }
34 }
35
36 void EndXact() {
37   if{budget > 0) {
38     // Xact has been executed in hardware
39     _xend(); // Commit the TSX transaction
40   } else {
41     // Xact has been executed in fall-back mode
42     sgl= 0;  // Release the lock
43   }
```

```
44 }
45
46 int SPEND_BUDGET(int b) {
47   if(roaPolicy == LINEAR) {
48     return b - 1;
49   } else if(roaPolicy == HALF) {
50     return b/2;
51   } else {
52     return 0;
53   }
54 }
```

Checking the `sgl` immediately after T starts (Line 14) ensures that `sgl` is in the read set of T. As we presented in Section 3, any posterior invalidation of `sgl` causes T to abort. This behavior guarantees that writing the lock (Line 27) effectively aborts all active transactions.

A compare-and-swap (CAS) operation (Line 27) checks if the lock is not taken, as in a traditional lock-based concurrency control, and produces the write that aborts all active transactions.

When T aborts, the respective thread returns to Line 11, but it is not running transactionally anymore. The abort rolls-back the modifications transactionally made, and an error status is written in the variable `status` (Line 10). This error status must be handled non-transactionally.

Every time T successfully starts the transaction, `status` is set to code `XBEGIN_STARTED`. Thus T runs transactionally (Line 12).

Every time T aborts, it restarts non-transactionally. `status` is set to one of the following codes: i) `XABORT_CONFLICT`; ii) `XABORT_CAPACITY`; or iii) 0, for a non-specified cause.

If the cause is a conflict, i.e., case i), either the lock is taken, or T has acquired a granule g that other transaction T' has also acquired. The acquisition of g by T' produces an invalidation of g that causes T to abort.

On abort due to conflict, T decreases its remaining budget by 1 (Line 23). If the cause of the abort is a capacity exception, i.e., case ii) (Line 20), T might be too large to fit into the cache. In this situation, three different policies (Line 2) have been studied in literature [50]: i) `LINEAR`, decreases the budget by 1 (same behavior as conflicts); ii) `HALF` decreases the budget by half; and iii) `ZERO`, immediately sets its budget to 0.

T repeats the code block until either it commits or exhausts its budget and is forced to start a serially. On commit, routine `EndXact` call the TSX commit instruction (Line 36). On finish serially, the lock is released (Line 42).

5 Analytical model

In this chapter we present our analytical model.

As in any white box model, knowledge of the system internals is needed. Unfortunately, the low-level details about Intel TSX implementation are not

disclosed. Therefore, the analytical model that we present assumes the internal behavior described in Section 3.

The ultimate goal for this analytical model is to predict the system probability of abort, i.e., p_a, and its throughput, i.e., X.

The description of the model follows a top down approach. First, we describe how we model the evolution of the system depending on the transactional events that take place (e.g., aborts and commits). The probabilities of, and the rate at, which these events occur are then analytically derived.

More in detail, this chapter is organized as follows: first, Section 5.1 presents the target system model and introduces the notation used to describe the model. In Section 5.2, we show how the system can be modeled as a Markov chain and how the rates for each of its state, the probability of abort and the throughput are computed. Further considerations on the implementing the transition matrix is presented in Section 5.2.2. In Section 5.3 it is present our considerations on how conflicts in HTM works, i.e., model contention having in account how granules are stored in the processor's cache. The sections that follows present the implementation of the given abstractions. Section 5.4 presents the equations for the probability of abort, and how we devise them. The equations for the response time are presented in Section 5.5. Finally, Section 5.6 presents the formulas for the transition rates.

5.1 System model and notation

We assume the system runs θ threads, which execute in closed loop transactional and non-transactional code blocks (noted, respectively, TCB and NTCB). The number of threads that is running TCB is θ_t, θ_n is the number of threads running NTCBs and θ_f is the number of threads running transactions in the fall-back path, whether they are executing or they are blocked, waiting the global lock to be freed.

Upon successfully completing the execution of the current block, a thread starts a TCB with probability p_t, or a NTCB with probability $1 - p_t$.

We note B the budget of a starting HTM transaction, i.e., the number of attempts the transaction is granted to execute as a hardware transaction. Upon restarting after an abort, a transaction decrements its available budget by one. When its available budget reaches 0, a transaction acquires the lock to execute in the software fall-back path.

As described in Section 4, the acquisition of the fall-back lock causes all other transactions to abort and to decrement their respective available budgets. This might cause other transactions to deplete their budget. Since only one thread can hold the fall-back lock at any point in time, the execution of all the transactions with a budget of 0 is serialized.

Transactions with more than one retry available are stalled until all the transactions with budget equal to 0 have completed their execution in the fall-back path, releasing the fall-back lock.

Symbol	Meaning
B	Initial budget of every transaction, i.e., maximum number of retries until the serialized path is used.
θ	Number of threads.
C	The average completion duration of a code block.
L	Number of accessed granules in a code block.
W	The average time window between every two accesses, i.e., $W = C/L$;
P_W	The probability of accesses being writes.
p_t	The probability of a thread starting a Transactional Code Block (TCB).
TCB	Transactional Code Block, i.e., portion of code in the application that must run atomically.
NTCB	Non-Transactional Code Block, i.e., code where HTM concurrency control is not considered.
X	Throughput.
p_a	Probability of abort.

Table 4: Summary of the input parameters of our model, among other symbols.

We refer to a transaction with only one hardware restart remaining as *dangerous*. By contrast, we call *non-dangerous* the transactions with an available budget greater than 1.

The model does not capture the effect of Hyper-Threading. Therefore, it assumes that θ is always less or equal to the number of available physical cores in the processor.

Each code block, i.e, TCB, serialized TCB and NTCB has an average completion time of C time units.

Each TCB access L distinct *granules*, i.e., memory words, that are chosen uniformly at random from a pool of size D. We assume that the granules are accessed uniformly during the duration of the transaction, namely every $W = C/L$ time units.

5.2 Modeling the system as a Markov chain

In this section we start by explaining the purpose of a Markov chain and how it is applied in our analytical model.

5.2.1 Brief Introduction of Markov chains

Markov chains are named after Andrey Markov, a Russian mathematician with vast works in stochastic processes.

Markov chains are stochastic processes that satisfy the "memorylessness" property, i.e., in each state there is no information on how the process reached that state, or which path is taken, i.e., the probability of moving to other state is independent of its history.

A special case of a Markov chain is the Continuous-time Markov chain (CTMC). A CTMC has a finite space of states and the time spent in each state is a real value that follows a exponential distribution (due to its "memorylessness" properties).

A CTMC can be described by a finite space of states S and a transition rate matrix $Q_{|S|\times|S|}$. Assuming that the system is ergodic, i.e., the graph is recurrent and aperiodic (i.e., there is always more than one path between any two nodes in the graph and all nodes are accessible), the resulting state probability matrix P, after some amount of time, converges to a "steady state" given by matrix P^*. All rows in P^* should be equal to a vector $\vec{\pi}$, where each position $\vec{\pi}_s$ is the probability of being in state s.

5.2.2 Solving the Markov chain

Assume a transition rate matrix Q. Each position q_{ij} correspond to the rate of moving from state i to state j.

We construct Q by filling in the positions q_{ij}, such that $i \neq j$. Then, the diagonal is completed as $q_{ii} = -\sum_{j=0}^{N} q_{ij}$, where N is the number of states. Thus, each row in the matrix sums 0.

Every state s has a given probability of being reached, i.e., $\vec{\pi}_s$, such that the sum of all the elements in $\vec{\pi}$ is 1 (Equation 1).

$$\sum_{s \in S} \vec{\pi}_s = 1 \tag{1}$$

Also, the rate matrix multiplied by $\vec{\pi}$ is equal to the $\vec{\pi}$, as shown in Equation 2. Assume $Q' = Q - I$. In Equation 3, we extend matrix Q' with a row of 1's, this is the constraint given by the previous Equation 1, note that the extended Q' is no longer a square matrix.

$$Q\vec{\pi} = \vec{\pi} \quad \Leftrightarrow \quad Q\vec{\pi} - I\vec{\pi} = \vec{0} \quad \Leftrightarrow \quad (Q - I)\vec{\pi} = \vec{0} \tag{2}$$

$$\left[\begin{array}{ccc} Q'_{11} & \cdots & Q'_{1N} \\ \vdots & \ddots & \vdots \\ Q'_{N1} & \cdots & Q'_{NN} \\ \hline 1 & \cdots & 1 \end{array}\right] \left[\begin{array}{c} \vec{\pi}_1 \\ \vdots \\ \vec{\pi}_N \end{array}\right] = \left[\begin{array}{c} 0 \\ \vdots \\ 0 \\ \hline 1 \end{array}\right] \tag{3}$$

Regarding the $A\vec{x} = \vec{b}$ equation, most solvers require A to be a square matrix. To do so, we can replace one of the rows in the Q' matrix with the row of 1's. We can do such a operation given that the equations are linear independent, thus, adding a new linear independent equation, with respect to the sum of $\vec{\pi}$, does not add more information to solve the matrix. An alternative solution that we use is multiply each part of the equation by Q'^T, note that in the second side we get the last column of Q'^T which is $\vec{1}$, as shown in Equation 4.

$$\left[\begin{array}{ccc|c} Q'_{11} & \cdots & Q'_{N1} & 1 \\ \vdots & \ddots & \vdots & \vdots \\ Q'_{1N} & \cdots & Q'_{NN} & 1 \end{array}\right] \left[\begin{array}{ccc} Q'_{11} & \cdots & Q'_{1N} \\ \vdots & \ddots & \vdots \\ Q'_{N1} & \cdots & Q'_{NN} \\ \hline 1 & \cdots & 1 \end{array}\right] \begin{bmatrix} \vec{\pi}_1 \\ \vdots \\ \vec{\pi}_N \end{bmatrix} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \tag{4}$$

As we implemented the analytical model in C++, the choosen solver is provided by the Eigen library [56], other framework/library could be used instead, with gains or loses in performance, there was not much effort to discover the best library. This library is capable encoding efficiently sparse matrix in memory and of solving the $A\vec{x} = \vec{b}$ equation for a large sparse matrix.

5.2.3 The case of HTM

We model our system as a CTMC. Ideally, to track the evolution of the system, a state in the Markov chain should represent the inner state of each active thread.

Namely, each state should count the available budget of every transactional thread and the number of granules each transaction has already accessed. This approach, however, would quickly run into scalability problems as the number of possible states and transitions would grow exponentially in B, L and θ.

Therefore, we choose to model the state of the system via a more compact representation. A state in our Markov chain is a tuple $[\theta_B, .., \theta_1, \theta_f, \theta_n]$, where θ_B and θ_1 are the number of transactions with, respectively, B and 1 retries left. θ_f are the number of threads running in the serialized path, i.e., with remaining budget 0, and θ_n are the number of NTCBs. Using this representation, the number of states only grows exponentially on the number of threads and budget. For instance, with $p_t = 1$ (i.e., discarding all states with NTCBs), 4 budget and 8 threads, there are 495 possible states. With 4 budget and 4 threads, there are 70.

For example, at some moment in the system, assume there are 4 transactions. Also, the initial budget is 2. Two transactions have the full budget, and two have 1 retry left. Then the resulting state s is: $[2200]$.

When building the transition matrix, one must be careful to not include states with non-transactions if p_t is equal to 1, i.e., only states with TCB are reachable, thus, states with NTCB must be discarded. The same applies when

p_a is very low. This can led to numerical issues when computing the solution vector $\vec{\pi}$, due to the fact that states, where TCBs have budget less than B, are very unlikely to be reached.

The possible events that may cause a transition are serialized TCB and NTCB finishing, or TCB either committing or aborting. Assuming a generic thread τ, the possible transitions are the following: i) τ finishes a NTCB and starts other NTCB; ii) τ finishes a NTCB and starts a TCB; iii) τ commits a TCB and starts other TCB; iv) τ commits a TCB and starts a NTCB; v) τ aborts a TCB and is non-dangerous, then, restarts and decreases its budget by 1; vi) τ aborts a TCB and is dangerous, then, restarts in the serialized path, all the remaining transactions abort and decrease its budget by 1; vii) τ commits a serialized TCB and starts other TCB; finally viii) τ commits a serialized TCB and starts a NTCB. These transitions are summarized in Table 5.

	Source state	Destination state	Rate
i)	$[\theta_B, .., \theta_f, \theta_n]$	$[\theta_B, .., \theta_f, \theta_n]$	$\mu_{\text{i)}}$
ii)	$[\theta_B, .., \theta_f, \theta_n]$	$[\theta_B + 1, .., \theta_f, \theta_n - 1]$	$\mu_{\text{ii)}}$
iii)	$[\theta_B, .., \theta_i, .., \theta_1, 0, \theta_n]$	$[\theta_B + 1, .., \theta_i - 1, .., \theta_1, 0, \theta_n]$	$\mu_{\text{iii)}}$
iv)	$[\theta_B, .., \theta_i, .., \theta_1, 0, \theta_n]$	$[\theta_B, .., \theta_i - 1, .., \theta_1, 0, \theta_n + 1]$	$\mu_{\text{iv)}}$
v)	$[\theta_B, .., \theta_i, t_{i-1}, .., \theta_1, 0, \theta_n]$	$[\theta_B, .., \theta_i - 1, t_{i-1} + 1, .., \theta_1, 0, \theta_n]$	$\mu_{\text{v)}}$
vi)	$[\theta_B, .., \theta_1, 0, \theta_n]$	$[0, \theta_B, .., \theta_1, \theta_n]$	$\mu_{\text{vi)}}$
vii)	$[\theta_B, .., \theta_f, \theta_n]$	$[\theta_B + 1, .., \theta_f - 1, \theta_n]$	$\mu_{\text{vii)}}$
viii)	$[\theta_B, .., \theta_f, \theta_n]$	$[\theta_B, .., \theta_f - 1, \theta_n + 1]$	$\mu_{\text{viii)}}$

Table 5: The state transitions for our Markov chain.

Recapitulating the previous example where $s = [2200]$, assume that one of the two transaction with a retry left (a dangerous transaction) aborts. It grabs the global lock and sets its budget to 0. Every active transactions also aborts. The transition is the following: $[2200] \Rightarrow [0220]$. In state $[0220]$, one fall-back transaction at the time executes. In this case there is one other thread blocked waiting to enter the fall-back path, and two other threads waiting to start a transaction. The following transitions are supposed to always occur: $[0220] \Rightarrow [1210] \Rightarrow [2200]$. Basically, the fall-back transactions exit, one by one, and assuming that $p_t = 1$, after they exit they attempt to start a new transaction.

Now we introduce the target Key Performance Indicator (KPI) for our model. Once calculated the given transition rates previously described, the Markov chain can be solved to compute two KPI of interest: the average probability of abort (i.e., p_a) and the throughput (i.e., X).

We note $s(\theta_i, \theta_f, \theta_n)$ the state with θ_i active transactions, θ_f fall-back transactions and θ_n active non-transactions. We note $\vec{\pi}_s$ the probability for the system to be in state s, as obtained by Section 5.2.2. Then, noting $p_{a,s}$ the abort

probability in state s, we compute the average probability of abort, i.e., p_a, as in Equation 5. Take a closer look at the fact that, in the computation of p_a, the states where $\theta_f > 0$ are discarded, because the serialized transactions do not abort and the non-serialized ones are blocked.

$$p_a = \sum_{s(\theta_i,\theta_f=0,\theta_n)\in S} \vec{\pi}_s p_{a,s} \tag{5}$$

The throughput of the system, that we note X, can be similarly computed as a weighted average of the throughputs in each state. Noting $\mu_{n,s}$ the rate at which NTCBs are completed, $\mu_{c,s}$ the rate at which TCB are completed and $\mu_{f,s}$ the rate at which serialized transactions are completed. We compute X as in Equation 6. Note, again, that for the states with $\theta_f \geq 1$, the throughput given by hardware transactions is zero, because they are stalled.

$$X = \sum_{s(\theta_i,\theta_f=0,\theta_n)\in S} \vec{\pi}_s(k\mu_{n,s} + i\mu_{c,s}) + \sum_{s(\theta_i,\theta_f\geq 1,\theta_n)\in S} \vec{\pi}_s(k\mu_{n,s} + \mu_{f,s}) \tag{6}$$

To obtain the response time of a transaction, we use little's law, a fundamental result in queuing theory. Little's law states that the average number of customers in a queuing system, l, is equal to the rate at which customers arrive and enter the system, λ, multiplied by the average time of a customer spends in the system, w, i.e., $l = \lambda \times w$.

Once we have X, we compute the response time of transactions R_t as follows in Equation 7. Given than R^* is the average time of serving a code block (either a transaction or NTCB), then, $X = \theta/R^*$. R^* is the weighted average of the average time serving a transaction, R_t, and serving a NTCB, R_n, i.e., $R^* = (1-p_t)R_n + p_t R_t$. NTCB are assumed to last C time units so $\mu_n = 1/C$ and $R_n = C$.

$$R_t = \frac{\frac{\theta}{X} - (1-p_t)R_n}{p_t} \tag{7}$$

As stated, computing p_a, X and R^* requires to solve the Markov chain. This, in turns, requires to know transition rates. These rates (and the per-state abort probabilities) are not supplied as input to the model, and need to be analytically computed.

5.3 Modeling the conflict dynamics

When a transaction T accesses a granule g at time t, it opens a so-called "vulnerability window". Namely, if T has read g, when another transaction writes g before T commits, then, T aborts. Similarly, if T has written g, if another transaction T' reads or writes g before T commits, then T aborts.

Since we assume that a transaction accesses on average a granules every $W = C/L$ time units, the i-th granule is accessed, on average, iW time units after the start of the transaction, and the corresponding vulnerability window lasts $w = C - iW$.

For the same reason, the rate at which the other $\theta - 1$ transactions access granules is $(\theta - 1)/W$ (Equation 8).

$$\lambda = \frac{\theta_t - 1}{W} \tag{8}$$

The probability of any of those accesses resulting in the abort of T during the vulnerability window w depends on the probability that at least one of such accesses is directed to g and that that access is incompatible with T's access on g. The probability that any access is issued against the specific granule g is $1/D$ (Equation 9).

$$P(g) = \frac{1}{D} \tag{9}$$

The probability that another access to g is incompatible with T's access to g corresponds to one minus the probability that the two accesses are compatible, i.e., one minus the probability that both the accesses are read accesses (Equation 10).

$$P_I = 1 - (1 - P_W)^2 \tag{10}$$

Therefore, the rate at which incompatible accesses to g are produced by the other $\theta - 1$ active threads is given by Equation 11.

$$H(i) = P_I P(g \leq i)\lambda \tag{11}$$

Assuming that such rate is exponentially distributed, we can express the likelihood that a conflicting access on g is issued at time t as shown in Equation 12.

$$P(\text{conflict } t \in [iW, (i+1)W[\,) = H(i)e^{-H(i)t} \tag{12}$$

The likelihood of T aborting because of concurrent incompatible accesses grows with the number of granules accessed by T. If T accesses i granules, the probability that an access issued by another transaction is on one of such i items is i/D.

For example, imagine three granules: g_1, g_2 and g_3. They are randomly acquired respectively by the given order from a pool of D granules. At a given moment in time, T only has g_1. The probability of some transaction conflicts with granule g_1 is $P(g_1) = 1/D$. Then, T acquires g_2. The probability of some transaction conflicts with granule g_2 is $P(g_2|\neg g_1) = P(g_2)$. For the third acquired granule the probability is $P(g_3|\neg g_1 \wedge \neg g_2) = P(g_3)$.

Due to the equiprobability and independence on the acquisition of the granules, the probability of conflicting with a generic granule g_i is $P(g_i) = 1/D$, i.e., it is not relevant if the previous granules were or were not conflicts.

When T reaches the i-th granule, it has exactly i granules. The probability of conflicting with each of them is $1/D$. Since there are i granules, the probability of conflicting with any of them is $i \times (1/D)$, i.e., Equation 13.

$$P(g \leq i) = \frac{i}{D} \tag{13}$$

Thanks to the probability density function presented in Equation 12, we can compute the probability that T successfully acquires the i-th granule. We note such probability $P_R(i)$

$P_R(i)$ is recursively expressed as the probability of reaching granule $i-1$ and not aborting in the time window of length $W = C/L$ between the access to granules $i-1$ and i (Equation 14). Naturally, the probability of reaching the first granule is 1 because, at the beginning, none granule is available to generate conflicts.

$$P_R(i) = \begin{cases} 1 & \text{, if } i = 0 \\ P_R(i-1)e^{-H(i-1)W} & \text{, if } i > 0 \end{cases} \tag{14}$$

The probability of not conflicting in $[i-1, i[$ is computed as the probability that no incompatible access is issued against any of the $i-1$ granules held by T during a time window of time C/L, i.e. Equation 15.

$$\begin{aligned} P(\neg\text{conflict in } [i, i+1[\,) &= 1 - P(\text{conflict in } [i, i+1[\,) \\ &= 1 - \int_0^W P(\text{conflict } t \in [iW, (i+1)W[\,)dt \\ &= 1 - \int_0^W H(i)e^{-H(i)t}dt \\ &= 1 - (1 - e^{-H(i)W}) \\ &= e^{-H(i)W} \end{aligned} \tag{15}$$

5.4 Modeling the probability of abort

In the previous section we have computed the probability of conflict. In a given state s, the probability of abort due to a conflict is the probability of not reaching the L-th granule, i.e., $p_{a,s} = 1 - P_R(L)$, as shown in Equation 16.

$$p_{a,s} = 1 - P_R(L) \tag{16}$$

However, a transaction can be aborted because of a conflict or because a dangerous transaction aborts, causing the cascade aborts of all other transactions.

Therefore, the probability of T aborting at time t is given by the probability that some other transaction issues an incompatible access at time t or that another dangerous transaction aborts.

This recursive definition makes it cumbersome to compute the abort probability.

To tackle this issue we adopt a two-step approach.

First we assume that no fall-back aborts occur and compute the rates at which transactions aborts, i.e., $\theta_t \mu_t p_{a,s}$. μ_t is computed regarding the inverse of the response time R of a TCB that either aborts or commits, as is presented in the next section.

Secondly, we adjust $P_R(i)$, i.e., compute an new version noted $P'_R(L)$, incorporating the fact that dangerous transactions can abort according to the rate $p'_{a,s}\mu'_t$. Similarly to $P'_R(L)$, we also adjust μ'_t.

5.4.1 Adjusted probability of abort

Assume that we have already computed the vector $\vec{\pi}$ using the $P_R(L)$ without fall-back aborts. Now we adjust $P_R(L)$ to encompass the fall-back aborts. As for now, the conflict rate $H(i)$ (Equation 11) does not include the rate of conflicts generated by the acquisition of the global lock, i.e., when a dangerous transaction aborts.

A generic transaction T may be either dangerous or non-dangerous. In the one hand, if T is dangerous, then, T may be aborted due to any of the other $d-1$ transactions, thus, the arrival rate of fall-back aborts is $(d-1)\mu_t(1-P_R(L))$. On the other hand, if T is non-dangerous, then, d dangerous transactions may abort T, hence, the arrival rate is $d\mu_t(1-P_R(L))$, as shown in Equations 17 and 18, respectively.

$$H_d(i) = H(i) + (d-1)\mu_t(1-P_R(L)) \tag{17}$$

$$H_n(i) = H(i) + d\mu_t(1-P_R(L)) \tag{18}$$

The values of d and n, respectively, the number of dangerous and non-dangerous transactions, are obtained from the current state information. In Section 5.2.3 s is defined as $s(\theta_i, \theta_f, \theta_n)$. It encodes the current number of TCBs, serialized TCBs and NTCBs. In order to compute $H_d(i)$ and $H_n(i)$ we need to know how many of the θ_i TCBs have budget 1, i.e., are dangerous, and how many have budget greater than 1, i.e., are non-dangerous. For the sake of understandability, we redefine s as $s(d,n)$ in this section and in Section 5.5.1.

The new $P_R(L)$ is shown in Equation 19. The computation of $P'_R(L)$ is the weighted average between the cases in which T is dangerous and in which is non-dangerous.

$$P'_{R,s(d,n)}(L) = \frac{d}{\theta_t} P_{R,d}(L) + \frac{n}{\theta_t} P_{R,n}(L) \tag{19}$$

The computation of $P_{R,d}(i)$ and $P_{R,n}(i)$, respectively, Equations 20 and 21, is done exactly as in Equation 14 for $P_R(i)$ with the slight difference of using, respectively, $H_d(i)$ and $H_n(i)$, instead of $H(i)$. The former is the probability of reaching granule i assuming T is dangerous. The latter assumes T is non-dangerous.

$$P_{R,d}(i) = \begin{cases} 1 & \text{if } i = 0 \\ P_{R,d}(i-1)e^{-H_d(i-1)W} & \text{if } i > 0 \end{cases} \tag{20}$$

$$P_{R,n}(i) = \begin{cases} 1 & \text{if } i = 0 \\ P_{R,n}(i-1)e^{-H_n(i-1)W} & \text{if } i > 0 \end{cases} \tag{21}$$

In order to compute the global probability of abort we use the probability of abort in a given state s as is presented in Equation 22.

$$p_{a,s} = 1 - P'_{R,s}(L) \tag{22}$$

Take closer look at the fact that $P'_R(L)$ is only used after $\vec{\pi}$ is obtained to adjust the fall-back aborts. The Markov chain is computed using $P_R(L)$.

5.5 Modeling the response time

In this section we present the calculation of the average response time of a TCB that either commit or abort, R, as the weighted average of a TCB that abort at granule 1 up to $i-1$ and a TCB that commits, i.e., reach the last granule.

Equation 23 proposes how to compute R. Assume $t_{\text{conflict } t\in[iW,(i+1)W[}$ to be the average time to abort in between the access of granules i and $i+1$.

$$R = P_R(L)C + \sum_{i=1}^{L-1} t_{\text{conflict } t\in[iW,(i+1)W[} \tag{23}$$

$P(\text{conflict in } [i, i+1[\,)$ is the probability of reach granule i and conflict in the instant between $[i, i+1[$. The probability of conflict in $[i, i+1[$ is already presented in Section 5.3 Equation 12.

$$\begin{aligned} t_{\text{conflict in } [i,i+1[} &= \int_0^W (iW+t)P_R(i)P(\text{conflict } t \in [iW,(i+1)W[\,)dt \\ &= \int_0^W (iW+t)P_R(i)H(i)e^{-H(i)t}dt \\ &= P_R(i)\left(iW(1-e^{-H(i)W}) + \frac{1}{H(i)} - e^{-H(i)W}\left(W + \frac{1}{H(i)}\right)\right) \end{aligned} \tag{24}$$

Then, by replacing Equation 24 in Equation 23 we obtain the needed formula for R in Equation 25.

$$R = P_R(L)C + \sum_{i=1}^{L-1} P_R(i) \left(iW(1 - e^{-H(i)W}) + \frac{1}{H(i)} - e^{-H(i)W} \left(W + \frac{1}{H(i)} \right) \right) \tag{25}$$

The definition of μ_t and μ_c is related with the inverse of the response time R, as is shown in Equations 26 and 27. μ_t is the rate that a single TCB ends (either commit or abort), and μ_c is the rate a single transaction commits. To obtain the rate that all transactions commit we simply multiply by θ_t, i.e., $\theta_t \mu_c$.

$$\mu_t = \frac{1}{R} \tag{26}$$

$$\mu_c = P_R(L)\mu_t \tag{27}$$

Also, assume the duration for both NTCB and serialized TCB to be C, thus, their completion rate is equal to the inverse of C, as is shown in Equations 28 and 29, respectively. The only difference is that NTCBs run concurrently, thus, the total rate from all threads running NTCBs is given by $\theta_n \mu_n$, and serialized TCBs run one at the time, independently of its number, hence, their completion rate is always μ_f.

$$\mu_f = \frac{1}{C} \tag{28}$$

$$\mu_n = \frac{1}{C} \tag{29}$$

5.5.1 Adjusted response time

As in Section 5.4.1, assume we have already calculated $\vec{\pi}$ using the $P_R(L)$ without fall-back aborts. Then, we compute $H_d(L)$, $H_n(L)$, $P_{R,d}(L)$ and $P_{R,n}(L)$ as in Equations 17, 18, 20 and 21, respectively.

As in the computation of $P'_R(L)$, R' (Equation 32) is computed as the weighted averages of the cases in which T is dangerous and in which is non-dangerous, respectively, R_d (Equation 30) and R_n (Equation 31).

$$R_d = P_{R,d}(L)C + \sum_{i=1}^{L-1} P_{R,d}(i) \left(iW(1 - e^{-H_d(i)W}) + \frac{1}{H_d(i)} - e^{-H_d(i)W} \left(W + \frac{1}{H_d(i)} \right) \right) \tag{30}$$

$$R_n = P_{R,n}(L)C + \sum_{i=1}^{L-1} P_{R,n}(i) \left(iW(1 - e^{-H_n(i)W}) + \frac{1}{H_n(i)} - e^{-H_n(i)W} \left(W + \frac{1}{H_n(i)} \right) \right) \tag{31}$$

$$R'_{s(d,n)} = \frac{d}{\theta_t} R_d + \frac{n}{\theta_t} R_n \tag{32}$$

Once we have R', μ'_t is the inverse of R', i.e., $\mu'_t = 1/R'$. Furthermore, the rate of committing transactions μ'_c is given by $P'_R(L)\mu'_t$ (Equation 33).

$$\mu'_{c,s} = (1 - p_{a,s})\mu'_{t,s} = P'_{R,s}(L)\mu'_{t,s} \tag{33}$$

As in Section 5.4.1, note that R' is only used to adjust the response time after the computation of $\vec{\pi}$. The Markov chain is computed using the R version of the response time.

5.6 Modeling the Markov chain transition rates

The rate for the Markov chain transitions are presented in this section. The transition table is in Section 5.2 Table 5.

Assume a generic thread τ that may be running a TCB, serialized TCB or NTCB. The transitions from Section 5.2 are summarized as follows:

i) τ finishes a NTCB and starts other NTCB:
$[\theta_B, .., \theta_f, \theta_n] \Rightarrow [\theta_B, .., \theta_f, \theta_n]$

ii) τ finishes a NTCB and starts a TCB:
$[\theta_B, .., \theta_f, \theta_n] \Rightarrow [\theta_B + 1, .., \theta_f, \theta_n - 1]$

iii) τ commits a TCB and starts other TCB:
$[\theta_B, .., \theta_i, .., \theta_1, 0, \theta_n] \Rightarrow [\theta_B + 1, .., \theta_i - 1, .., \theta_1, 0, \theta_n]$

iv) τ commits a TCB starts a NTCB:
$[\theta_B, .., \theta_i, .., \theta_1, 0, \theta_n] \Rightarrow [\theta_B, .., \theta_i - 1, .., \theta_1, 0, \theta_n + 1]$

v) τ aborts a non-dangerous TCB:
$[\theta_B, .., \theta_i, .., \theta_1, 0, \theta_n] \Rightarrow [\theta_B, .., \theta_i - 1, \theta_{i-1} + 1, .., \theta_1, 0, \theta_n]$

vi) τ aborts a dangerous TCB:
$[\theta_B, .., \theta_i, .., \theta_1, 0, \theta_n] \Rightarrow [0, \theta_B, .., \theta_{i+1}, .., \theta_2, \theta_1, \theta_n]$

vii) τ finishes a serialized TCB and starts a TCB:
$[\theta_B, .., \theta_1, \theta_f, \theta_n] \Rightarrow [\theta_B + 1, .., \theta_1, \theta_f - 1, \theta_n]$

viii) τ finishes a serialized TCB and starts a NTCB:
$[\theta_B, .., \theta_1, \theta_f, \theta_n] \Rightarrow [\theta_B, .., \theta_1, \theta_f - 1, \theta_n + 1]$

For transitions i) and ii), respectively, with probability $1 - p_t$ or p_t a thread starts a NTCB or a TCB. The arrival rate of this kind of threads is $\theta_n \mu_n$, where $\mu_n = 1/R_n$. R_n is the completion time of a non-transaction, which is set to C.

Transitions iii) and iv) are the cases where a thread commit a TCB. Then, respectively, with probability p_t or $1 - p_t$ starts a TCB or a NTCB. We define

the probability of commit as the complement of the probability of abort, i.e., $1 - p_a$. The total arrival rate of threads in scenarios iii) and iv) is $\theta_i \mu_t$.

Transitions v) and vi) are the cases where a thread aborts a TCB. Both dangerous and non-dangerous transactions have the same probability of abort, thus, the rate in both transitions is the same.

Transitions vii) and viii) are the cases where a thread ends a serialized TCB. Note that serialized TCBs run one at the time. When they complete, they may either start a TCB or a NTCB, as in the previous cases.

The transition rules are shown in Table 6.

	Transition	Rate
i)	$[\theta_B, .., \theta_f, \theta_n] \Rightarrow [\theta_B, .., \theta_f, \theta_n]$	$\theta_n \mu_n (1 - p_t)$
ii)	$[\theta_B, .., \theta_f, \theta_n] \Rightarrow [\theta_B + 1, .., \theta_f, \theta_n - 1]$	$\theta_n \mu_n p_t$
iii)	$[\theta_B, .., \theta_i, .., \theta_1, 0, \theta_n] \Rightarrow [\theta_B + 1, .., \theta_i - 1, .., \theta_1, 0, \theta_n]$	$\theta_i \mu_t (1 - p_a) p_t$
iv)	$[\theta_B, .., \theta_i, .., \theta_1, 0, \theta_n] \Rightarrow [\theta_B, .., \theta_i - 1, .., \theta_1, 0, \theta_n + 1]$	$\theta_i \mu_t (1 - p_a)(1 - p_t)$
v)	$[\theta_B, .., \theta_i, .., \theta_1, 0, \theta_n] \Rightarrow [\theta_B, .., \theta_i - 1, \theta_{i-1} + 1, .., \theta_1, 0, \theta_n]$	$\theta_i \mu_t p_a$
vi)	$[\theta_B, .., \theta_i, .., \theta_1, 0, \theta_n] \Rightarrow [0, \theta_B, .., \theta_{i+1}, .., \theta_2, \theta_1, \theta_n]$	$\theta_1 \mu_t p_a$
vii)	$[\theta_B, .., \theta_1, \theta_f, \theta_n] \Rightarrow [\theta_B + 1, .., \theta_1, \theta_f - 1, \theta_n]$	$\mu_f p_t$
viii)	$[\theta_B, .., \theta_1, \theta_f, \theta_n] \Rightarrow [\theta_B, .., \theta_1, \theta_f - 1, \theta_n + 1]$	$\mu_f (1 - p_t)$

Table 6: The state transitions rates for our analytical model.

5.7 Model of cache capacity

In this section we present our model for capacity exceptions.

Assuming that the granules are stored in a known private cache, with a well defined number of sets and ways-associativity, it is possible reformulate this problem into the following: "how many combinations one can fill B bins, each with capacity C, with I distinguishable balls", i.e., a combinatorics problem.

The maximum number of configurations of the B bins with the I distinguishable balls, without the capacity constraint, is given by B^I. To justificate this formula. Assume the first attempt. The ball can be put in one of the B bins, thus, B possible states arise. In the second attempt, for all those states, a ball can again be placed in any bin, hence, $B \times B$ combinations. And so on. Therefore, at the I-th attempt, the number of combinations is B^I.

With the capacity constraint, the calculation can be done recursively as shown in Equation 34.

$$N(B, C, I) = \begin{cases} 1 & \text{, if } B = 0 \vee I = 0 \\ \sum_{x=\min_c}^{\max_c} \binom{B}{x} \prod_{y=0}^{x-1} \binom{I - yC}{C} \times N(B - x, C - 1, I - xC) & \text{, otherwise} \end{cases} \quad (34)$$

In order to devise Equation 34, firstly we define the maximum number of bins that one can fill with C balls, out of a total of I balls, i.e., $\max_c$. Secondly, we define the minimum number of bins, each also filled with C balls, from a total of I balls, i.e., $\min_c$. Respectively, Equations 36 and 35.

$$\max_c(B, C, I) = \left\lfloor \frac{I}{C} \right\rfloor \tag{35}$$

To compute the maximum, always try to throw the balls at the same bin until it fills up. Then, when the bin is full, we proceed the same way for some other bin. How many groups of C balls is possible to obtain with I balls? It is computed as the integer division of I per C, the remainder is not enough to fill a bin, thus is discarded.

$$\min_c(B, C, I) = \max(0, I - B(C - 1)) \tag{36}$$

To compute the minimum, always try to fill the bins without reaching its full capacity. When we are no longer able to do so, i.e., the bins are all at capacity $C - 1$, the subsequent balls have to, necessarily, fill a bin. How many groups of C balls is possible to obtain with I balls, given that we want to avoid full bins? Before the bins are all filled up, it is possible to accommodate $B \times (C - 1)$ balls. To obtain the minimum, subtract $B \times (C-1)$ balls from the total I balls, then, if the number is negative or zero, it means it is possible to place the balls without completely fill any bin.

Given the B bins with capacity C and the I balls, the number of full bins x is in between $\min_c$ and $\max_c$, inclusive. How many of the x bins can one choose from the total B? Basically the binomial B choose x, i.e., $\binom{(}{B}, x)$.

Then, in how may ways can one arrange xC balls in the x bins? Basically, for the first bin, there are I balls choose C. For the second, there are $I - C$ balls choose C. This is given by the productory in Equation 34.

Once placed the xC balls in the x full bins, $I - xC$ balls are still remaining. These must be split in the remaining $B - x$ bins. To do so, we recursive call $N(B - x, C - 1, I - xC)$, which gives the combinations for the remaining bins and balls, but with capacity $C - 1$. Capacity is decreased in each iterations because x captures all the full bins with capacity C, so, the remaining bins cannot be full.

$N(B, C, I)$ is the total number of configurations of I distinguishable balls in B bins of capacity C. $N(B, C, I)/B^I$ is the frequency of "valid" configurations. Where "valid" means that bins capacity is not exceeded. $B^I - N(B, C, I)$ configurations are not "valid", i.e., the capacity of at least one bin is exceed by any amount of balls.

Thus, the probability of capacity at the i-th granule is the one given in Equation 37. Assume that B and C are globally defined for the system, respectively, as the number of sets the n-way associativity.

$$P(\text{capacity} \leq i) = 1 - \frac{N(B, C, i)}{B^i} \tag{37}$$

At this point, $P_R(i)$ must be extended to support the capacity aborts. In Section 5.3, $P_R(i)$ is defined as the probability of reaching the i-th granule. This is the same as the probability of reaching granule $i-1$ and not conflict in between granules $i-1$ and i. Now an extra constraint is introduced: "and not experiencing a capacity exception at the i-th granule".

$$\begin{aligned} P''_R(i) &= P_R(i) \times P(\neg\text{capacity} \leq i) \\ &= P_R(i) \times \frac{N(B, C, i)}{B^i} \end{aligned} \tag{38}$$

As shown in Equation 38, the $P_R(i)$ now includes the capacity constraint. Assume as in Section 5.3 $P_R(0)$ to be equal to 1. Also note that the analytical model is computed in two steps: first $\vec{\pi}$ is obtained using $P_R(L)$, and second, the approximate $P'_R(L)$ is computed to encompass fall-back aborts. The presented $P''_R(i)$ must replace $P_R(L)$ in the first step and in the second step. For the second step, instead of using $P_R(i)$ in Equation 38 we use the approximated $P'_R(i)$.

Take closer look at the budget update rule for capacity exceptions, that, in this case, is the same as in the case of conflicts and fall-back aborts. Therefore, a transactions in the presence of any type of abort always reduce its budget to 1 (linear policy). If other policy is used instead, then the Markov chain must the encompass new transitions.

6 Simulation model

In this chapter our simulation model is presented. The simulation is built on the studied behavior for HTM, regarding conflict and capacity detection. Also, as it is built to validate the analytical model, some further parametrization allows the simulation to behave closely to the real system, but it can be tuned to behave regarding the analytical model assumptions.

In Section 6.1 we present some assumptions regarding the differences between the analytical model and the HTM system. Then, Section 6.2 briefly describes the usage of ROme OpTimistic Simulator (ROOT-SIM) simulator in our developed event-driven simulation. Finally, in Section 6.3 extra details of the simulations are presented.

6.1 Modeled behavior assumptions

In this section we present the assumptions of the simulative model regarding the empirical studies presented in Section 3 and the abstractions made by the

analytical model. The assumptions regarding conflict and capacity detection are presented in Sections 6.1.1 and 6.1.2, respectively. Additional assumptions are made in Section 6.1.3.

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. We are assuming caches to be inclusive, i.e., the higher levels of cache include the lower ones. We assume also the eviction policy to be LRU, i.e., the line that is not used for the longest time is evicted [53].

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.

6.1.1 Concurrency control

We assume the conflict detection studied in Section 3.2. Given two transactions, T_1 and T_2 disputing a granule g, the last to access wins, and the first aborts immediately.

Regarding the fall-back lock, we assume serialized TCB to be executed one at the time, before any other TCB that is not in the fall-back path executes. This is not completely true for the HTM code. As stated in Section 4 the data race on the fall-back lock have some impact on the waste of budget by waiting transactions. The simulation do not capture this behavior.

6.1.2 Capacity exceptions

The simulation (as the analytical model) assumes data is kept in a single cache with S sets and N-way associativity. As in the balls and bins problem of Section 5.7, the cache is abstracted in a group of B bins of capacity C. When one set is full, and receives a granule, the simulation aborts the transaction due to a capacity exception. As stated in Section 3.3, this seems to be a valid consideration regarding write only workloads.

Caches also come with an eviction policy. Since we do not model the cache system, we do not simulate eviction. When a transaction starts the cache is empty. When a set is full and receives a granule the transaction aborts.

6.1.3 Additional assumptions

Though Intel processors have three levels of cache (L1 and L2 are private, L3 is shared), we do not model the cache subsystem, neither its interaction with main memory. The focus of this dissertation is to investigate the efficiency of HTM's concurrency control and not the impact of multi-level caches. Granules are acquired and, what happens next is defined by the gathered information on the TSX behavior presented in Section 3.

Differently from the analytical model, in HTM applications, the commit instruction is not instantaneous. Thus, the simulation must be versatile enough

to better match the real system without degradation on the analytical model assumption. Extra latency is parametrized in the simulation as: the time of execute the commit instruction, i.e., T_{com}; and, the non-transactional time between two TCB, i.e., T_{NTC}

We are only assuming granules of the size of a cache line. Different granularity may cause a misalignment, and, thus, if g_1 and g_2 are kept in the same line, conflicts with g_1 may also affect g_2. Other problems may arise if a granule occupies more than a cache line. These dynamics are not captured in the model presented in this dissertation.

6.2 ROOT simulator

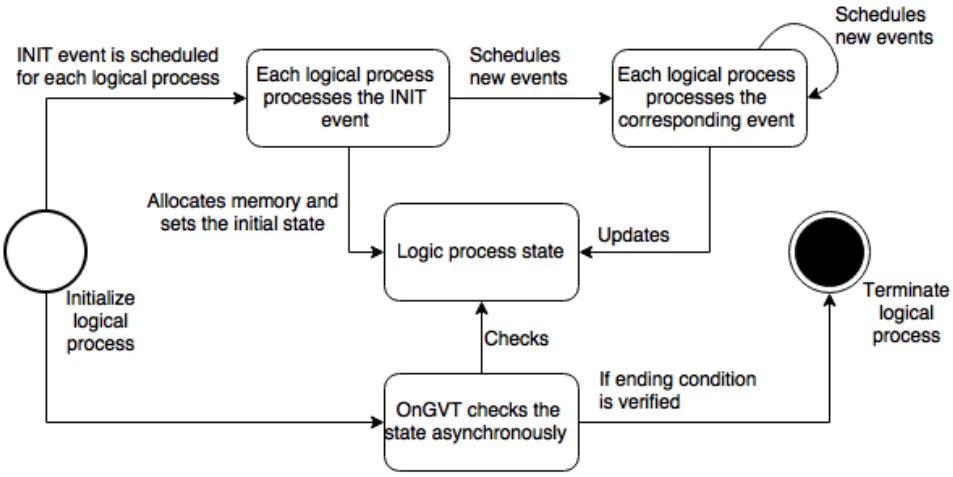


Figure 6: ROOT-SIM behaviour. In each Logical Process (LP) initialization the next event(s) is(are) scheduled and the LP's state is set. Then, the events are executed having in account the given timestamp. Events may schedule other events in an arbitrary LP, thus, the simulator is looping in the current events. A correct simulation is assumed to always generate events. The simulation ends when the ending condition is reached in all LPs.

To simulate HTM we will use a event-driven simulator, namelly, the ROOT-SIM [55] simulator. 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 the scheduled events, set the state of each Logical Process (LP) and a last API routine to check ending conditions. In Figure 6, a summarization on how ROOT-SIM works is presented.

ROOT-SIM needs the programmer to provide how many LPs the simulation will use and if it is a sequential or a multi-threaded simulation. Once the simulation starts, each LP receives an `INIT` event. In that event, LPs must initialize their internal state and schedule the first events.

ROOT-SIM also requests the programmer to implement a routine to check if the LP ended. When all LPs ends the simulations finishes.

An event is characterized by its event code and a buffer with information/-parameters of the event. LPs do not have necessarily to schedule events in themselves, they may schedule events in other LPs.

In a non-sequential execution, the usage of global information is not advisable, since ROOT-SIM do not roll-back data that is not in the state of some LPs. Thus, all state information should be stored within the state of each LPs.

6.3 Simulation internals

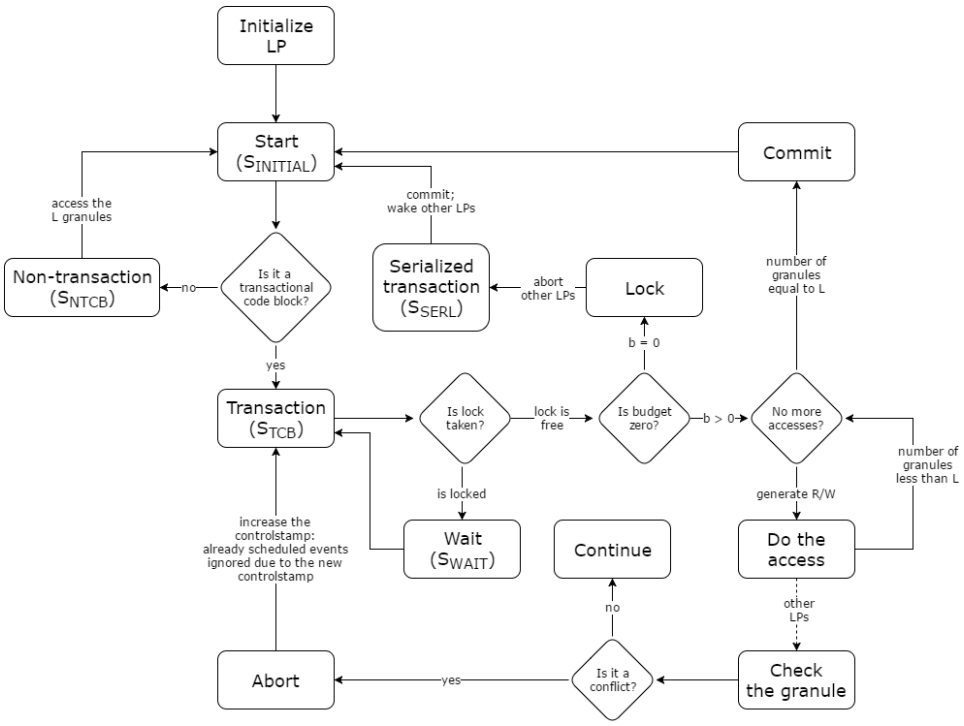


Figure 7: Simulation diagram.

In this section we will present the details of the simulative model. To simplify the presentation, we will first neglect the issue of modeling capacity exceptions, postponing the discussion on how this aspect is simulated to Section 6.3.1.

We extensively use a discrete event-driven simulator, namely the ROOT-SIM simulator presented in Section 6.2, to simulate HTM. Each LP simulates a thread. Each thread may be running a TCB or a NTCB.

Each thread executes a loop. After a TCB or a Non-Transactional Code Block (NTCB) finishes, the LP picks a new code block, and starts over. For now, we will consider the case where we have only transactions running we our system.

We make sure to filter out results obtained during the initial phase of the simulation, which may be affected by transient phenomena (e.g., initial synchronization of actions by the various simulated threads), by resetting the collected simulation statistics (e.g., abort rate, throughput) after a fixed number of event (which we typically set to 5000).

A LP may be in one of the following states: i) initial state, S_{INITIAL}, i.e., the LP still has to decide whether to start a TCB or a NTCB, also it resets its state; ii) TCB state, S_{TCB}, i.e., the LP is generating granules and it broadcasts its choices; iii) serialized state, S_{SERL}, i.e., the LP ran out of budget and needed to enter the fall-back path, since no conflicts arise were, we just schedule the LP to release the lock after C time units; iv) the NTCB state, S_{NTCB}, it does the same that the TCB state, but it never experiences aborts; and finally v) the wait state, S_{WAIT}, i.e., the LP is waiting the lock to be released either to go to state ii) or state iii).

The previously described states are summarized in Figure 7.

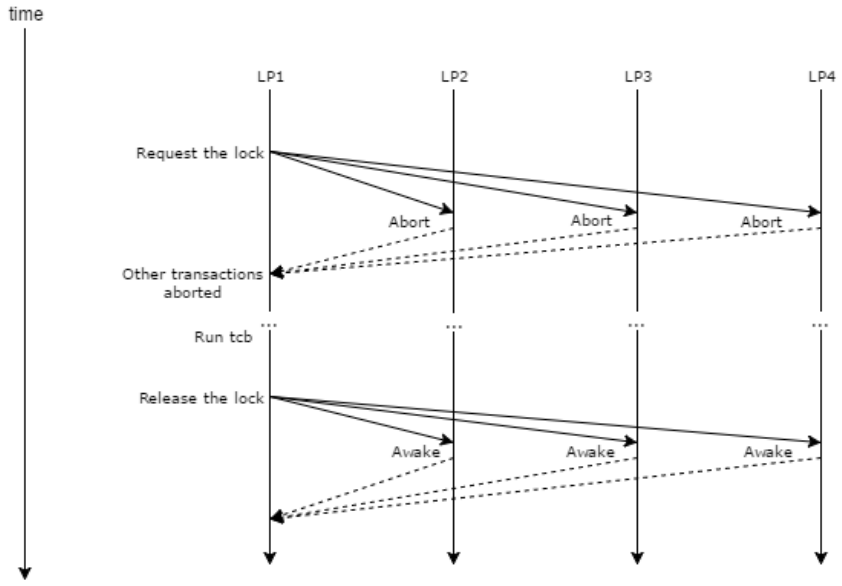


Figure 8: Lock aquisition timeline.

This is the typical retry logic implemented in software (see Algorithm 3 in Section 4), which we also model in the simulation.

Once the LP decided to execute a TCB in state S_{INITIAL}, it checks the lock by broadcasting a event to all other LPs. The lock is considered not occupied if there is a consensus among the other LPs that: no one is using the lock; and, no one intends to use the lock. We do this to avoid the declaration of global variables, as ROOT-SIM do not handle roll-back them in multi-threaded runs.

If the lock is taken, the transaction T goes to a state S_{WAIT}. If it is not taken, the transaction proceeds to state S_{TCB}, where it processes the granule accesses.

When all the granules have been accessed, then the transaction commits and can no longer be aborted.

Assume now that T is aborted consecutively and enters the fall-back path, state S_{SERL}. The same consensus approach is used. However, if two want to enter state S_{SERL} a tiebreaker criterion must be applied.

In order to provide fairness accessing the fall-back path, a special timestamp is stored in each LP state, i.e., $TS_{\text{last SERL}}$, with the finish timestamp of the last serialized execution.

The LP with the oldest $TS_{\text{last SERL}}$ enters state S_{SERL} first. The remaining enter one at the time according to the same criterion.

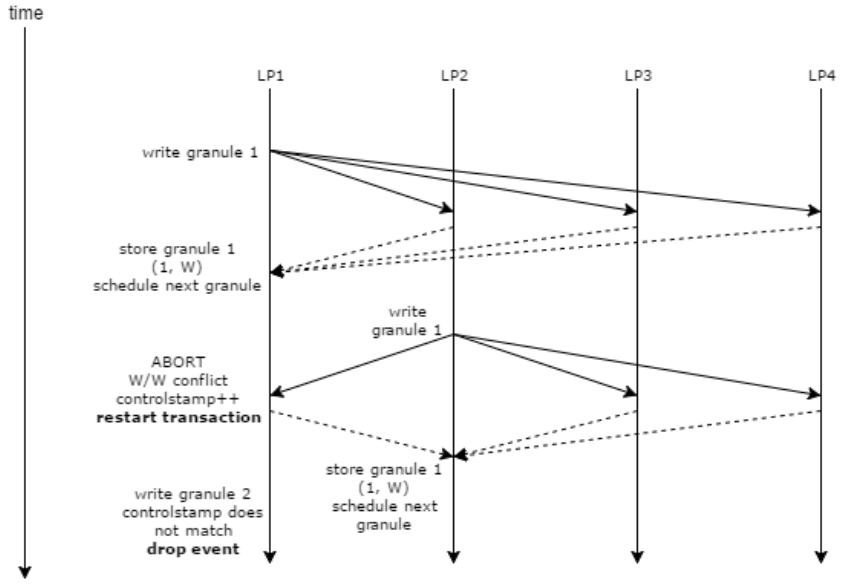


Figure 9: Conflict due to granule aquisition.

When a LP receives a lock request it immediately aborts and enters state S_{WAIT}. When no more LPs are in state S_{SERL}, LPs continue the execution of TCBs. Figure 9 shows a timeline for the lock acquisition.

While in S_{TCB}, LPs generate random granules g uniformly distributed in the granule space, such that $0 \leq g < D$. After each access, g is broadcast to the remaining LPs. If some LP has g in an incompatible access, it aborts the running transaction T.

Aborts occur asynchronously of the acquisition of the granules. New acquisitions may be scheduled in the future when an abort event triggers. ROOT-SIM does not provide a unschedule primitive, thus, we needed to develop a mechanism to validate each event.

In the state of each LP an extra sequence number is added, i.e., TS_{control}, which we call controlstamp in Figure 9. This sequence number is incremented each time a transaction within the LP aborts or commits, and is sent in the

buffer of each event. When the event is executed, the event TS_{control} is checked against the sequence number in the LP state, if they do not match it means the event needs to be discarded. This dynamic is presented in Figure 9.

6.3.1 Enabling capacity exceptions

In the previous section, we only talked about conflicts, and how we simulate them. Now we will approach how we simulated capacities. As discussed in Section 6.1.2, we do not model the entire multi-level cache subsystem.

As previously stated, we are confident write accesses can be modeled as they are being held in a single cache with a given number of S sets, each set with capacity N. This can be seen as a matrix of size $S \times N$. Each position in that matrix is a boolean that tells if it is occupied or not.

When the transaction starts the matrix is empty.

Granules are already being kept in the LP's state. Each LP's state is provided with the presented matrix. When a granule is acquired its row in the matrix is calculated using the modulus function, i.e., $\mod S$, then it is accommodated in one of the available slots. If we want to simulate other mapping algorithm, it should also be straightforward to implement.

7 Validation study

This chapter is devoted to presenting the results of a twofold validation study aimed at quantifying the accuracy of both the analytical and the simulation models presented so far.

We start, in Section 7.2, by validating the analytical model presented in Chapter 5, using the simulation model described in Chapter 6.

Next, in Section 7.2, we report the results of a study aimed at validating the event-based simulation model describe in Chapter 6, by comparing its performance predictions with a real system (Intel Xeon E3-1275 v3 @ 3.50GHz) equipped with Intel TSX and running a set of synthetic workloads.

Here we present our experimental results. In Section 7.2, the gathered data from the hardware and from the simulation is compared. Then, in Section 7.1 the analytical model is validated against the simulative one.

The accuracy of the proposed models will be evaluated using 3 main metrics:

- Mean Absolute Error (MAE), defined as $\frac{1}{n}\sum_{i=1}^{n}|x_i - y_i|$
- Mean Absolute Percentage Error (MAPE), defined as $\frac{1}{n}\sum_{i=1}^{n}\frac{|x_i - y_i|}{y_i}$
- The Pearson correlation factor r, defined as $\frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i-\bar{y})^2}}$.

Where x_i and y_i represent the values of a performance metric (e.g., abort rate or throughput) output by the model being validated and by the system

being used to validate (i.e., the simulation in Section 7.1 and the real system in Section 7.2).

7.1 Analytical model validation

In this section we present the validation of our analytical model based on the gathered data from the developed simulative model. We start by validating the analytical model regarding conflicts only workloads in Section 7.1.1. Then, in Section 7.1.2, we do the opposite, focusing on workloads with negligible probability of conflict, but with a non-negligible likelihood of incurring capacity exceptions.

7.1.1 Conflict exceptions

We start by presenting the experiment setup, then, we present the results, and after that, we discuss possible conclusions from the obtained results.

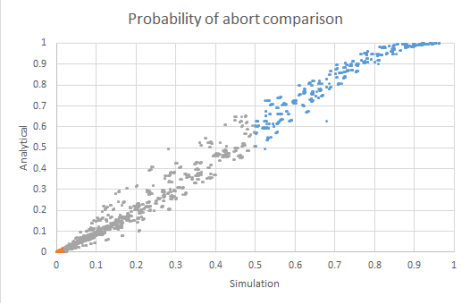
We setup this this experience as follows:

- θ, number of threads, in range 2-4 we restrict B, budget to values in $\{1, 2, 3, 4, 5, 6\}$;
- θ, number of threads, in range 5-8 we restrict B, budget to values in $\{1, 2, 3, 4\}$;
- L, number of acquired granules within a TCB, which vary in the following set of values: $\{2, 5, 10, 20, 50\}$;
- D, size of the granule pool, which vary in the following set of values: $\{2000, 5000, 10000, 50000, 100000\}$;
- P_W, probability of write access, varying in the values: $\{0.1, 0.5, 0.9, 1\}$;
- C, average time to completion, varying in: $\{1, 2, 8\}$.

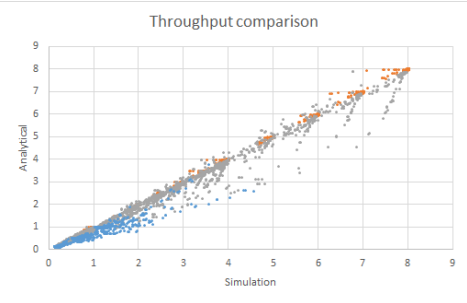
This input should provide a wide variety of different workloads, which should stress various aspects in the analytical model. The total number of workloads is 10200 and the results are shown in Figure 10.

Note that we do not compute for more than 8 threads and 6 budget. Also, for the 5-8 threads workload, only budget from 1 to 4 is used. This is related with the performance for both models. The simulation stops after either have obtained the given large number of sampled TCBs or after a timeout of 5 minutes. The execution time for the analytical model is shown in Figure 11, which grows exponentially.

From the results presented in Figure 10a one can conclude that with $p_a < 50\%$ the analytical and simulative models are very aligned, after the 50% mark, the analytical model generally overestimates the probability of abort. This impact the prediction in the throughput front by generally underestimating it.

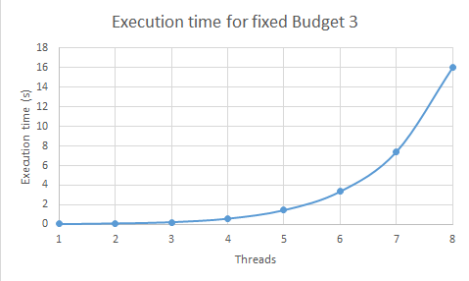


(a) Probability of abort.
MAE $= 1.69\%$, $r = 0.9949$

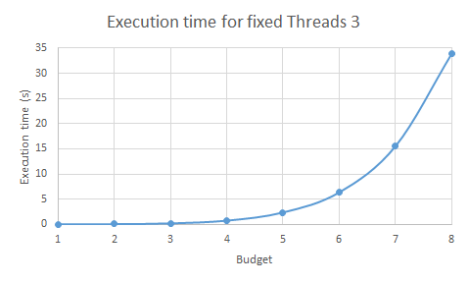


(b) Throughput.
MAPE $= 4.07\%$, $r = 0.9963$

Figure 10: Comparison of the simulation with the analytical model.



(a) Time to execute the analytical model with different number of threads and fixed number of budget 3.



(b) Time to execute the analytical model with different number of budget and fixed number of threads 3.

Figure 11: Performance of the analytical model overview, the vertical axis shows the execution time in seconds (s).

Also note, in Figure 10a, some clusters of points are formed in the axis $Y = X$, i.e., points $(1, 1)$, $(2, 2)$, etc. This is explicable for workloads where the probability of abort p_a is very low. For low p_a, R_t is approximately equal to C, thus, throughput is given by $X \approx \theta/C$.

7.1.2 Capacity exceptions

We could not discover how to Intel is storing the read set, as is discussed in Section 3. Therefore, all tests presented in this section are write only workloads.

Firstly, we validate our capacity formula against the simulation. The Formula 34 in Section 5.7 is very expensive to compute to a large capacity. So, we pick a number of sets S and the n-way set associativity N, such that S and N are close from the real system, but are still computable in a reasonable time with our analytical model.

For this experiment, we pick $S = 32$ and $N = 8$, the results are presented in Figure 12. We did not use $S = 64$ and $N = 8$, as in the real system,

because the combinatorial formula grows into very large numbers that cannot be represented using double-precision floating-point primitives, e.g., with $S = 64$ and $N = 8$ the maximum capacity is 512, but in the 160th granule we get that $N(64, 8, 160) \approx 9 \times 10^{160}$, for larger number of granules the result is infinite. So, to use this formula in the real system configuration we need higher precision.

Also, in order to efficiently compute the $N(B, C, I)$ formula of Section 5.7, we cache the values in a map structure with key equal to the tuple (B, C, I) and value equal to $N(B, C, I)$, i.e., $\langle (B, C, I),\ N \rangle$. Despite of greatly increasing performance with little memory footprint, it still does not solve the precision problem. To have an idea of how much memory the cache consumes, after run $N(32, 8, 200)$, $N(64, 8, 160)$, $N(64, 8, 250)$ and $N(64, 16, 500)$, among other parameter combinations, the number of elements in the cache, without any sort of tuple eviction, is 55073.

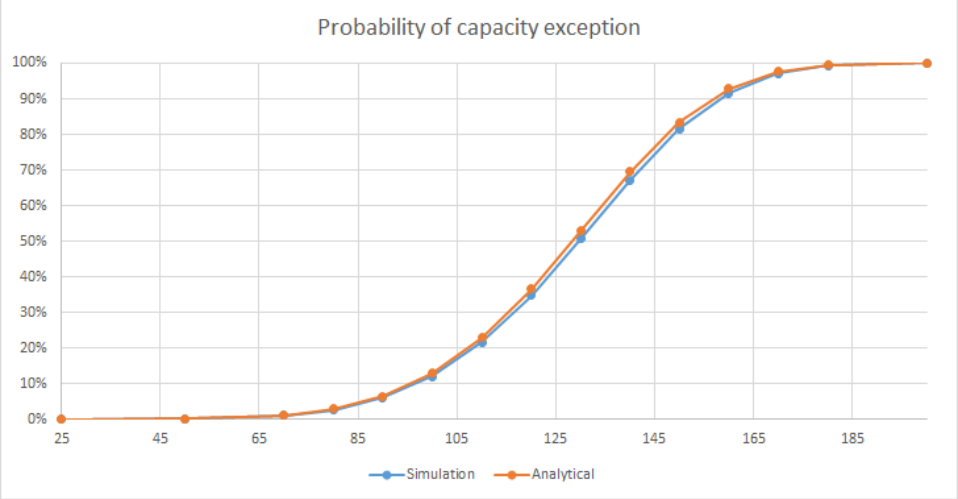


Figure 12: The simulation and analytical probability of capacity exception at the i-th granule. The number of sets is 32 and the n-way set associativity is 8. The Pearson correlation factor is $r = 0.9998$.

As expected, the equation closely match the simulation. The resulting function has a sigmoid shape. Before some number of granules the probability of capacity is $\approx 0\%$ and after a predictable number of accesses the probability is $\approx 100\%$. In the area where the probability is $\approx 50\%$ there are some differences between the simulation and the analytical model. However, when developing HTM applications the analytical predictions are a close match to answer the question: "how many granules can the transaction randomly acquire before starting experiencing capacity exceptions"?

Now, we present an experiment where the previous formula is merged with the model for conflicts using a linear policy for the capacity exceptions, as suggested in Section 5.7. This means that capacity exceptions are treated

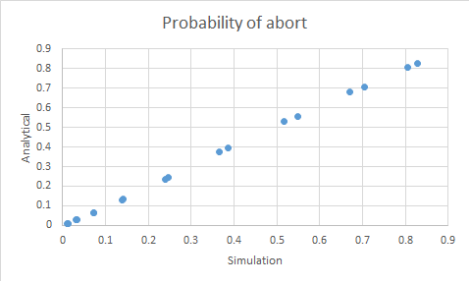
as the conflict aborts, and the budget is decreased by 1 each time a abort is detected. Merging the capacity formulas with the analytical model only modifies the $P_R(i)$ from the conflicts only model to $P''_R(i)$ presented in Section 5.7.

In this experiment, the setup is:

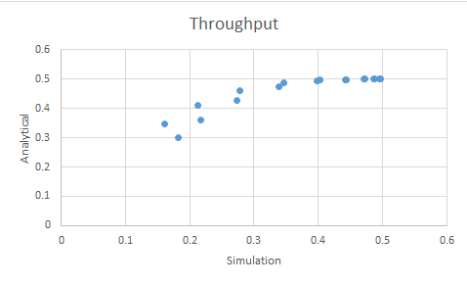
- θ, number of threads set to 2;
- B, number of retries, which we vary in the following set of values: $\{3, 5\}$;
- P_W, probability of write access, equal to 100%;
- D, size of the granule pool, equal to 100000;
- L, number of acquired granules within a TCB, which vary in the following set of values: $\{25, 30, 35, 40, 45, 50, 55, 60\}$;
- C, average completion time, $\{4\}$.

Furthermore, we used a cache geometry of 32 sets 4-way set associative. The values for L are pick to stress capacity in the zone where random accesses produce significative probability of abort, i.e., below the 25 granules mark the capacity exception probability is close to 0%, but over the 60 granules mark the probability is near 100%.

The results for this experiment are shown in Figure 13.



(a) Probability of abort.
MAE $= 0.66\%$, $r = 0.9998$



(b) Throughput.
MAPE $= 35.2\%$, $r = 0.8971$

Figure 13: Comparison of the simulation with the analytical model for with capacity exceptions.

This is a minimal setup that minimizes the number of conflicts, so the capacity exceptions are the most significant source of aborts. As expected, the probability of abort is predicted very accurately, however, the throughput is not.

Note that the throughput near the value 0.5, i.e., $\theta = 2$ threads divided by $C = 4$, is obtained with low probably of abort. For higher probability of abort the analytical model is clearly overestimating throughput. This could be explained with the fact that dangerous transactions are not modeled directly in the Markov chain transitions, or the fact that, with capacity aborts, transactions need to

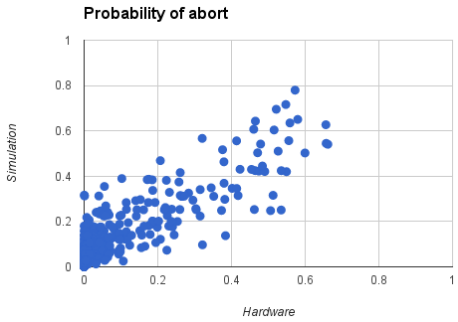
acquire a considerable amount of granules before abort. Thus, transactions that restart due to capacity exceptions waste more time than with the other source of aborts, i.e., conflicts and lock acquisition.

7.2 Simulive model validation

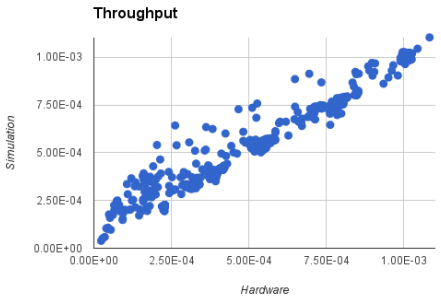
In this section we present the validation of our simulative model based on the gathered data from the TSX system. In Section 7.2.1, a study regarding mostly conflict exceptions is presented. We consider workloads that are very unlikely to be subject to capacity exceptions. Tests regarding the capacity exceptions are presented in Section 3.3.4, where we try to discover the number of "polluted" granules in L1 cache.

7.2.1 Conflict exceptions

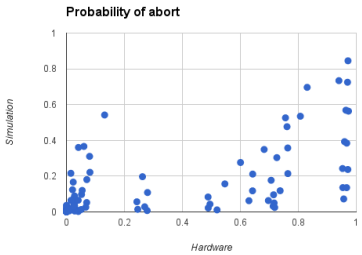
As mentioned this section is focused on validating the accuracy of the simulation model in predicting the performance dynamics of a real TSX system in presence of workloads that do not incur capacity but can be subject to (largely) varying probabilities of transaction conflict.



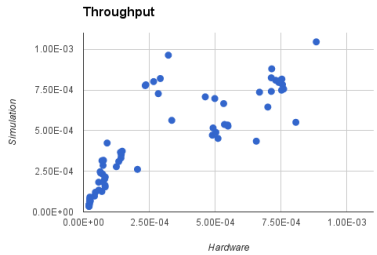
(a) Probability of abort, $B > 1$.
MAE $= 2.75\%$, $r = 0.8781$



(b) Throughput, $B > 1$.
MAPE $= 14.98\%$, $r = 0.9971$



(c) Probability of abort, $B = 1$.
MAPE $= 17.1\%$, $r = 0.6787$



(d) Throughput, $B = 1$.
MAPE $= 71.5\%$, $r = 0.9902$

Figure 14: Comparison of the simulation with Intel TSX. The data is split in the cases where $B = 1$ and $B > 1$.

To this end, we designed a test with 630 different workloads, which were obtained by considering the cross-product of the following values for:

- θ, number of threads in a range of 2-4;
- B, number of retries, which vary in the following values: $\{2, 4, 6, 8, 10\}$;
- P_W, probability of write access of 50% and 100%;
- D, size of the granule pool, which vary in the following set of values: $\{1024, 2048, 4096, 8192, 16384, 32768\}$;
- L, number of acquired granules within a TCB, which vary in the following set of values: $\{1, 2, 5, 10, 15, 20, 30\}$.

We did not experiments of values of θ greater than 4 to avoid the possible effects of Hyper-Threading in the obtained results. Which we do not capture in the simulation.

The system measurements can have large variations to tackle the issue, for each workload setting, we executed 30 runs, filtering out the 1st and 4th quartile, and computing the mean on the remaining samples.

Note that, depending on the workload characteristics (e.g., θ and L), the value of the following parameters of the simulation model can vary:

- The average completion time C;
- Time to execute the commit (`_xend()`) instruction T_{com};
- The non-transactional time between the commit and the start of a new transaction T_{NTC}.

So, we instrumented the program executing on the real system to use the Intel Time Stamp Counter (TSC) to measure the values of the above parameter and feed them to the simulator.

As shown in Figure 14a and 14b, in the case that $B > 2$ the simulation is aligned with the obtained data from the real system. The MAE for the probability of abort is 2.75%, which is relatively good, but the obtained correlation of 0.8781 shows some noise in the data. Though, the obtained correlation for the throughput is much better with a double 9 result of $r = 0.9971$, as for the obtained MAPE of $= 14.98\%$ is acceptable.

On the other hand, the results for $B = 1$ shown in Figure 14c and 14d are not so good, the probability of abort as a low correlation of $r = 0.6787$, and the throughput has a high error, MAPE $= 71.5\%$. This can be explained considering that the data race is not captured by the simulation model, because the simulation model is aligned with the analytical model which does not capture that. Thus, all transactions in the fall-back path execute first, and we assume the transactions that are not in the fall-back path to remain blocked without

trying to start a transaction. However, a number of independent studies [11] have reported that recommended values for B are in the order of at least 5, so, fortunately, this workload is not really representative of real life scenarios.

8 Conclusion

In this dissertation we 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.

Our work aims to build white-box models capable of predicting the performance of commercial HTM, with focus on Intel TSX, which is probably the most widespread one.

As a preliminary step to achieve this goal, we have designed a set of experiments that allowed us to shed lights on internal dynamics of TSX implementation, which are undisclosed by Intel. In particular, our tests aimed at reverse engineering two main aspects of HTM implementation in Intel's processors: i) the concurrency control algorithm they employ; ii) the organization of transactional data in the CPU's cache hierarchy and its impact on the probability of incurring capacity exceptions.

Our study concluded: i), regarding conflicts, the last transaction to acquire a granule aborts all active transaction that have a copy of that granule in a incompatible access type, i.e., R/W, W/R or W/W; and, ii), regarding capacity, in the Intel case, L1 stores the transactional writes. We discuss some hypothesis on modeling the read capacity, as for instance, that reads occupy some space in L1 but its eviction may not cause an abort.

On the basis of the results of this empirical study, we built both an analytical model and a discrete event simulation. The former abstracts the system into a continuous time Markov chain, where in each state threads may be running TCB or NTCB, with a given budget value. The latter mimics the real system empirical observations, but abstracts the cache hierarchy and other transparent dynamics.

Next an extensive validation study was conducted of both the analytical model and the simulation. In particular we first validate the analytical model vs the simulation, and then the simulation vs a real system running on Intel Xeon E3-1275 v3 @ 3.50GHz. The result of the validation considered up to 630 different workload settings, in the case of simulation validation, and 10000 in the case of the analytical model validation, so to span a broad range of working conditions for the system. The results show that the proposed analytical model achieves average accuracy MAE of 4.07% on the probability of abort and MAPE of 4.07% on the throughput, in a conflict detection scenario. In the capacity

exceptions front, we managed to achieve very good average accuracy MAE of 0.66% for the probability of abort.

8.1 Future work

This section points out possible future research avenues that have been opened by this work.

A first direction is extending our models to other architectures, besides Intel, and in particular to IBM P8. Preliminary experiments conducted on P8 suggest that the same abstract model of concurrency control could be adopted also for P8. So both the current analytical and simulation models should be able to capture already P8's dynamics regarding transactional conflicts.

It is instead unclear whether the white-box models developed in this work to predict the likelihood of capacity exceptions would apply straightforwardly to P8. The works in [25] suggest that P8 may store transactional data in "[...] standard core caches, and that these caches likely each have 4 sets and an associativity of 16 [...]". Provided that this hypothesis is correct, then the proposed analytical model would be capable of predict performance as well as in the Intel TSX system.

Yet, a thorough validation study based on real data gathered on a P8 machine is in order before concluding that the current models can be used also in that context.

In Section 3.3 it is discussed that reads interact with the stored writes in L1 cache, but we is still did not completely undisclosed it. The plots in Figure 4 shown that it may possible to model the reads as consuming some fraction of the L1 cache. Further workload data must be collected in future experiments.

In this dissertation we always assumed a workload where transactions do random accesses to memory, uniformly distributed. It would be interesting to relax this assumption by extend the current model to incorporate techniques, such as the one proposed in [39].

Relaxing this assumption would be necessary in order to conduct validations that encompass a broader set of applications, including standard benchmarks, like STAMP [32], which were overviewed in Section 2.5. The workloads of each benchmark might be described once parameterization options are provided.

Regarding modeling of capacity exceptions, this work opened an interesting research question that is still unaddressed. Although reads and write accesses are tracked using different caches in current Intel's processors, i.e., L3 and L1 respectively, read accesses still need to be ultimately served using L1 cache. Hence, read accesses do occupy space in L1 caches. Although read accesses can be evicted from L1 caches without causing the transaction to abort, they can cause the eviction of cache lines that were updated by the transaction causing its abort. Hence, reads do reduce the effective capacity of the L1 cache to serve write accesses. Our study in Section 3 shed some lights on this problem. Yet, it would be interesting to confirm whether the theorized "interference" of

read accesses on the effective capacity of L1 cache can be actually explained by extending the simulation model to encompass explicitly a multi-layer cache and emulate the eviction dynamics caused by load read cache lines in L1. If this hypothesis is confirmed, one could build on this knowledge to extend the current analytical model that predicts the probability of capacity exceptions to capture accurately the real dynamics of Intel processors also for transactions that encompass read access.

Finally, the retry logic considered by the current analytical model assumes that capacity exceptions are treated just like conflicts, i.e., they decrease by 1 the budget of available retries for a transaction. The model could be extended, via a relatively straightforward redefinition of the topology of its Markov chain, to encompass, for instance, policies that acquire immediately the fallback lock as soon as a capacity is detected.

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] 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.

[4] Intel Corporation. Desktop 4th Generation Intel Core Processor Family (Revision 028). Technical report, Intel Corporation, 2015.

[5] IBM. *Power ISA Version 2.07*. www.power.org.documentation, version 2. edition, 2013.

[6] Y. C. Tay, Nathan Goodman, and R. Suri. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4):415–462, 1985.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. *Pact*, pages 3–14, 2014.

[12] Rachid Guerraoui. Opacity : A Correctness Condition for Transactional Memory. *Communication*, 2007.

[13] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[14] 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.

[15] 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.

[16] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. *Distributed Computing*, 4167:194–208, 2006.

[17] 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.

[18] 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.

[19] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *ACM SIGPLAN Notices*, 44:155, 2009.

[20] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[21] 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.

[22] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, and David A Wood. LogTM: Log-based Transactional Memory. pages 1–12, 2006.

[23] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. *Proceedings - International Symposium on Computer Architecture*, 00(C):494–505, 2005.

[24] Takuya Nakaike, Matthew Gaudet, and Maged M Michael. Quantitative Comparison of Hardware Transactional Memory for Blue Gene / Q , zEnterprise EC12 ,. *ISCA*, 2015.

[25] Andrew Nguyen William Hasenplaugh and Nir Shavit. *Investigation of Hardware Transactional Memory*. PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2015.

[26] H Burton Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[27] Daniel Molka, Daniel Hackenberg, and Wolfgang E Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. 2015.

[28] 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.

[29] 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.

[30] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7. *ACM SIGOPS Operating Systems Review*, 41(3):315, jun 2007.

[31] 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.

[32] 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.

[33] Armin Heindl and Gilles Pokam. An analytic framework for performance modeling of software transactional memory. *Computer Networks*, 53(8):1202–1214, 2009.

[34] 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.

[35] Simon Haykin. Neural Networks: A Comprehensive Foundation. jul 1998.

[36] Ingo Steinwart and Andreas Christmann. *Support Vector Machines*. 2008.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] Diego Didona and Paolo Romano. Enhancing Performance Prediction Robustness by Combining Analytical Modeling and Machine Learning. *ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2015.

[42] 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.

[43] Diego Didona and Paolo Romano. On Bootstrapping Machine Learning Performance Predictors via Analytical Models. In *ICPADS*, 2015.

[44] Diego Rughetti and Pierangelo Di Sanzo. Tuning the Level of Concurrency in Software Transactional Memory : An Overview of Recent Analytical , Machine Learning and Mixed Approaches. pages 395–417, 2015.

[45] Daniel F Garcia. Performance Modeling and Simulation of Database Servers. *The Online Journal on Electronics and Electrical Engineering (OJEEE)*, 2(1):183–188, 2009.

[46] 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, 1992.

[47] U. Herzog and M. Paterok, editors. *Messung, Modellierung und Bewertung von Rechensystemen*, volume 154 of *Informatik-Fachberichte*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.

[48] 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.

[49] 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.

[50] Nuno Diegues and Paolo Romano. Self-Tuning Intel Transactional Synchronization Extensions. *ICAC*, pages 1–11, 2014.

[51] Stuart J. Russell and Peter Norvig. Artificial Intelligence: A Modern Approach. feb 2003.

[52] 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.

[53] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–162. IEEE, dec 2010.

[54] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373. IEEE Comput. Soc. Press, 1990.

[55] 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.

[56] Gael Guennebaud and Others and Benoit Jacob. Eigen v3, 2010.

[57] Yujie Liu, Justin Gottschlich, Gilles Pokam, and Michael Spear. TSXProf: Profiling Hardware Transactions. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, 2016-March:75–86, 2016.