2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems

An Analytical Model of Hardware Transactional Memory

Daniel Castro[1], Paolo Romano[1], Diego Didona[2], and Willy Zwaenepoel[2]

[1]INESC-ID and Instituto Superior Técnico, Universidade de Lisboa
[2]École Polytechnique Fédérale de Lausanne (EPFL)

***Abstract*—This paper investigates the problem of deriving a white box performance model of Hardware Transactional Memory (HTM) systems. The proposed model targets TSX, a popular implementation of HTM integrated in Intel processors starting with the Haswell family in 2013.**

An inherent difficulty with building white-box models of commercially available HTM systems is that their internals are either vaguely documented or undisclosed by their manufacturers. We tackle this challenge by designing a set of experiments that allow us to shed lights on the internal mechanisms used in TSX to manage conflicts among transactions and to track their readsets and writesets.

We exploit the information inferred from this experimental study to build an analytical model of TSX focused on capturing the impact on performance of two key mechanisms: the concurrency control scheme and the management of transactional meta-data in the processor's caches. We validate the proposed model by means of an extensive experimental study encompassing a broad range of workloads executed on a real system.

I. Introduction

One of the main sources of complexity in parallel programming stems from the need to properly synchronize accesses to shared memory regions. The traditional, lock-based approach is well-known to be error-prone, even for experienced programmers [23]. Transactional Memory (TM) [22] has emerged as a simpler, and hence more attractive, alternative to lock-based synchronization.

Over the last two decades, the research on TM has led to many different designs and implementations, either in software [18], [17], [7], hardware [24], [30], or combinations of both [6]. Software-based TM (STM) systems rely on software instrumentation to trace memory accesses and detect the concurrent execution of conflicting transactions. STM supports a broad range of concurrency control algorithms, but the overheads resulting from software-based tracking of transactions' data accesses can severely hinder application performance [4]. These overheads can be avoided by delegating the implementation of the TM abstraction to hardware mechanisms, an approach that goes under the name of hardware transactional memory (HTM). While a number of alternative HTM designs have been proposed in the literature, the HTM implementations that are currently commercially available [24], [30] are built as relatively non-intrusive extensions of the cache coherency algorithm and, as such, suffer from several restrictions [16], [20]. Overall, make the performance of HTM is much dependent on a number of workload parameters and architectural design choices [16], [20], [28], [15], [10] — which makes the problem of predicting the performance achievable by HTM-based applications a very challenging task.

This paper takes a step towards clarifying our understanding of HTM's performance by developing what is, to the best of our knowledge, the first analytical model of an HTM system ever published in the literature. The presented model targets a popular implementation of HTM, which has been integrated in mainstream Intel processors since 2013 and goes under the name of Transactional Synchronization Extensions (TSX).

The first challenge we had to face in order to enable the construction of an analytical model of TSX was to obtain information on some key internal mechanisms, which are undocumented by Intel and undisclosed by previous literature. We addressed this issue by designing a set of experiments that allowed us to gain insights on how TSX resolves conflicts between transactions and tracks memory accesses across the cache hierarchy.

Based on our experimental findings, we develop an analytical model focused on capturing the dynamics of two mechanisms that have a crucial impact on the performance of HTM systems: the schemes employed to manage conflicts among concurrent transactions and to track the memory regions accessed by transactions. The model allows us to gain a deeper understanding of the effect that design choices, parameterization and workload characteristics have on performance. Moreover, the model may serve as a building block to implement performance prediction and optimization schemes for applications that are built on top of TSX.

We validate the proposed model using a real system and a set of synthetic micro-benchmarks. The experimental results show that the model can predict the application's throughput and abort rate with $< 10\%$ error for a broad range of workload settings.

2375-0227/17 $31.00 © 2017 IEEE
DOI 10.1109/MASCOTS.2017.29

II. Background

HTM applications use compiler directives to demarcate the begin and end (commit) transactions. The code enclosed between these two directives is executed atomically and in isolation, as if it was protected by a mutex lock. The TM runtime executes the code speculatively and leverages the underlying cache coherence protocol to detect and resolve conflicting accesses to memory.

Current HTM systems provide a *best-effort* implementation of the TM abstraction, in the sense that transactions are not guaranteed to commit even if they run in absence of concurrency. This is due to the fact that existing HTM systems use the processor's cache hierarchy to track transactional accesses, and rely on the cache coherence protocol to detect conflicts. As a consequence, transactions whose footprint exceeds the processor's cache capacity are subject to *capacity* aborts. A transaction can also experience other types of spurious aborts (i.e., aborts not imputable to conflicting accesses), because of external events like page faults, context switches and system calls.

HTM-based applications must, thus, rely on a fall-back mechanism to guarantee that a transaction eventually commits. The default approach is to allow transactions to execute in a software fall-back execution path, guarded by a Single *Global* Lock (SGL). When a hardware transaction aborts, it can acquire the SGL instead of retrying its execution in hardware. The SGL is also read by each transaction upon its start. Therefore, when a transaction activates the fall-back path by acquiring the SGL, any concurrent hardware transaction is aborted, and only restarted when the lock becomes free again. This approach ensures that a transaction that acquires the SGL executes in isolation from other transactions. The downside is that it serializes the execution of transactions, which can severely decrease performance.

The policy governing the retry logic of a transaction (upon an abort event) can be implemented either in hardware or in software. The latter approach provides more flexibility, allowing for tuning not only the maximum number of hardware attempts, which we call the *budget*, but also how such budget should be consumed in presence of different abort types [16].

III. Dissecting Intel's HTM implementation

We investigate two key aspects of TSX's HTM implementation: i) how it manages conflict among concurrent transactions, and ii) how a transaction's metadata are maintained in the cache hierarchy and what impact this has on its capacity limitations.

Intel has disclosed limited information on the internal mechanisms employed by its HTM implementation. The information reported in the rest of this section is either based on previous external studies [28], [29], or inferred via experiments designed explicitly to shed light on the internals of Intel's HTM implementation. The results reported in this section and in the remainder of this paper are based on a i7-5960X eight-core processor running at 3.0GhZ, equipped with 32 GB RAM and Ubuntu 15.04.

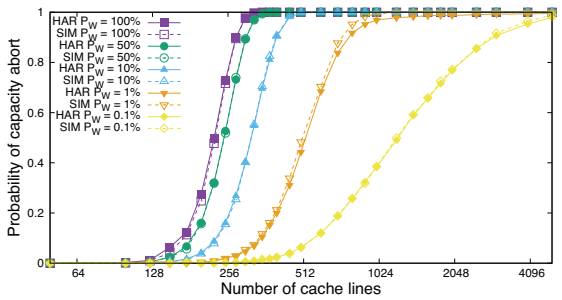


Figure 1: Probability of capacity abort when accessing a different number of cache lines. Comparing a real system (HAR) vs a simulator modelling only L1 (SIM).

A. Conflict detection and resolution.

Existing literature [28], [29] has already pointed out that Intel's HTM implementation relies on an eager conflict detection scheme, i.e., when a conflict arises between two transactions, one of the two transactions is immediately aborted. Another relevant aspect of the conflict detection schemes in existing HTM implementations is that, since they are built on top of a pre-existing cache coherency protocol, the conflict detection granularity is equal to the cache line size, which is, for our target Intel CPU, 64 bytes. The conflict resolution policy used by TSX, i.e., which transaction is aborted in the presence of a conflict, is an aspect that, to the best of our knowledge, has not been documented by Intel and has not been systematically investigated by previous studies. We tackled this issued by designing an experiment that forces two transactions to issue conflicting memory accesses (load or store of one memory word) in different orders, by injecting properly tuned delays during transaction's execution. This experiment revealed that TSX uses a a "last requester wins" policy, i.e., if two concurrent transactions conflict (i.e., they access the same memory word and at least one is a write), the first transaction to have performed the access is aborted.

B. Capacity limitations.

Intel has not disclosed how transactional metadata are maintained by its HTM implementation, but previous studies [28], [29] have already partially answered this question, reaching the following conclusions:

- Writes issued in a transaction are stored in the L1 data cache. However, the maximum number of writes that can be executed by a transaction is smaller than what

could be accommodated by the entire L1 data cache, around 450 cache lines vs a total of 512. Nguyen [29] hypothesized that this reduction of the effective capacity of the L1 cache could be explained by considering that a transaction must also have sufficient space to store other program metadata, like the head of the program stack.

- Read-only transactions can perform a much larger number of reads than the L1 and L2 caches can possibly store, and the maximum read capacity is around half of the total L3 cache size. In the light of these observations, Nguyen [29] therefore hypothesised that the transactional reads are maintained in L3. Unlike L1, L3 is shared among all the cores of the same processor, as well as by programs code and data — which may explain why a transaction's read capacity is smaller than the size of the entire L3 cache.

In this study, we address two questions that are still unanswered by previous studies: i) how many cache lines in L1 are occupied by additional metadata maintained by transactions (i.e., metadata not used to track the transaction readset and writeset)? and ii) what is the effective capacity of transactions that execute a mix of read and write operations?

To answer these questions we built a simulator of an L1 cache that uses the same geometry of our reference processor (8-way associative, 64 sets, 64-bytes cache lines, 32KB capacity) and implements a Least Recently Used (LRU) eviction policy. To validate our assumptions on the internal mechanisms employed in the considered HTM implementation, we compare the output produced by the execution of synthetic programs running on the real system with the output generated by simulating the execution of the same programs.

Size of additional transactional metadata in L1. To determine the size of the additional metadata stored by transactions, we designed the following experiment. We occupy P cache lines, chosen uniformly at random in the simulated cache, to emulate the insertion of additional transactional metadata upon the start of a transaction. Then we simulate random writes to memory using the granularity of a cache line. We report a capacity event in the simulation when we evict one of the cache lines storing one of the addresses written by the transaction or one of the additional transactional metadata. We varied the value of P$\in$[0,512] and compared the average number of writes that a transaction could successfully execute in 50000 simulated and real runs. The value of P that produces the best match is 3, a value that appears reasonable especially if one considers that the transactional metadata may not be cache line aligned and hence may span multiple cache lines.

Capacity with mixes of read/write operations. Previous works on the capacity of HTM implementations, e.g., [29], [28], have considered workloads composed solely of read-only or write-only transactions. We report in Figure 1 the probability for a transaction to incur a capacity exception when attempting to access i distinct cache-aligned addresses selected uniformly at random, where each access has probability P_W of being a write.

Figure 1 reveals that halving the number of writes issued by a transaction (P_W=0.5) does not lead to a doubling of the effective capacity of transactions, but yields only a modest increase of the transaction's capacity — whose median moves from around 220 to 250 accesses. We argue that this phenomenon is not imputable to evictions of (read) cache lines in the L3 cache, which has a 8MB capacity and can accommodate thousands of random reads with high probability. We hypothesize, conversely, that, given the large relative difference in size between L1 and L3 (32KB vs 8MB), the transaction capacity is, for non-negligible values of P_W, largely dependent on dynamics taking place at the L1 cache. In fact, whenever a transaction issues a read access, the corresponding cache line has to be loaded in the L1 cache. This may cause the eviction from L1 of cache lines that had been previously accessed by the same transaction. If the evicted cache line had been written by the transaction, a capacity exception is triggered. If the evicted cache line had been read, the transaction does not have to abort, since the corresponding metadata are still stored in L3.

We tested our hypothesis using the same L1 simulator mentioned above, and, as can be observed in Figure 1, we obtain a very close match for P_W values as small as 1%. Below this value, as expected, the likelihood of incurring evictions of cache lines in the transaction's readset (stored in L3) becomes non-negligible.

Overall, this study confirms that, for a broad range of P_W values ([1.0 - 0.01]), it is possible to predict the probability of capacity aborts quite accurately via models that capture exclusively the behavior of L1 and neglect the dynamics affecting L3 — which are inherently more complex given the typically shared nature of L3.

IV. Analytical model

This section presents an analytical model of Intel's TSX. Section IV-A presents the model's key parameters and assumptions. Section IV-B illustrates the methodology adopted to derive the model. Sections IV-C and IV-D present a first version of the model that, for the sake of presentation, does not consider capacity exceptions. Then, in Section IV-E, we discuss how to extend the model to encompass also these sources of transaction aborts. Finally, Section IV-F details how to solve the model and obtain the predicted KPIs.

A. Key parameters and assumptions

We consider an HTM system with θ threads that, in a closed loop, execute either a transactional code block

(TCB) or a non-transactional code block (NTCB). Intel CPUs support simultaneous multi-threading (SMT) of up to two threads in the same physical core. When multiple threads run on the same physical core, they contend for the core's hardware resources, including its cache. For the purpose of this work, we do not model the effects of SMT. When a thread completes a code block, it starts a new TCB with probability p_t and a new NTCB with probability $1 - p_t$. A TCB has an average service time, i.e., CPU demand, of C time units, and a NTCB has an average service time C_n. The time to complete a TCB is assumed to be the same, independently of whether it is executed using HTM or using the software fall-back path. However, the HTM path has additional costs for starting (T_B) and committing (T_C) a transaction. Likewise, the software fall-back path incurs costs for acquiring (T_B^l) and releasing (T_C^l) the SGL.

We model the following retry policy to deal with transaction aborts: transactions are initially assigned a budget of B retries; upon an aborts, the transaction's budget is decremented by one; if the budget is exhausted, the transaction is executed using the software fall-back.

A transaction accesses on average L distinct cache lines, or *granules*. The timing of such accesses is spread uniformly at random during a transaction's lifetime. In other words, a transaction performs a memory access, on average, every C/L time units. The granule accessed at each iteration is chosen uniformly at random over a set of cardinality D. The probability that a memory access is a write is denoted P_W.

The model assumes that memory accesses issued by threads running NTCBs do not interfere with the execution of transactional threads — an assumption that is met, for instance, by the C++11 data-race-free model [3]. The model also does not consider aborts caused by interrupts or page faults. This last assumption simplifies the development of the model without significantly impacting its accuracy, as these sources of aborts are typically negligible in real-life workloads [16].

In the model a restarted transaction is indistinguishable from a transaction that starts for the first time. In addition, the execution times of code blocks are assumed to be exponentially distributed i.i.d. variables.

Finally, the model assumes a stable and ergodic system [26], so that quantities like abort probabilities and the mean execution times exist and are finite, and defined to be either long-run averages or steady-state quantities.

B. Modeling methodology and target KPIs

Our model is based on average value analysis [34]: it takes as input system parameters, e.g., θ and B, the average values corresponding to the workload characterization, e.g., C and C_n, and it returns average values of three Key Performance Indicators (KPIs). Specifically, the model computes the probability that a transaction aborts, P_A, the average throughput of the system, X, and the average response time of a transaction, R, i.e., the average time spent by the transaction including multiple re-executions possibly in the fall-back path.

We model the evolution of the system by means of a Continuous Time Markov Chain (CTMC) [26]. The CTMC's vertices represent the states in which the system can be and the edges represent the rates at which the system transitions from one state to another. A CTMC's state is uniquely identified by a tuple $\langle \theta_B, \theta_{B-1}, \ldots, \theta_0, \theta_n \rangle$, where $\theta_i,\ i = B, \ldots, 1$ indicates the number of threads that are running a TCB and still have i hardware retries remaining from their initial budget. θ_0 is the number of threads that have exhausted their budget and have to execute using the sequential fall-back path. θ_n is the number of threads executing a NTCB. Since we are modeling a closed system where threads constantly execute a code block, it follows that $\sum_{i=0}^{B} \theta_i + \theta_n = \theta$. Overall, the number of states in the CTMC is equal to the number of ways in which θ indistinguishable balls can be put into $B+2$ distinguishable bins, which is given by $\binom{\theta+B+1}{B+1}$ [19].

The system transitions from one state to another upon the completion of a NTCB, and upon the commit or abort of one or more transactions. When $\theta_0 = 0$, threads executing hardware transactions can execute in parallel. When $\theta_0 > 0$, hardware transactions are stalled until the global lock is free, and the execution of threads with depleted budget is serialized. Threads executing a NTCB are not affected by the acquisition of the global lock.

We denote by $\mu_{t,s}$ the rate at which a thread completes a transactional code block, either successfully or prematurely because of an abort, in the current state s. Note that, whenever the value of a variable is state-dependent, we shall specify the identifier of the current state, s, as a subscript. We denote by $\mu_n = 1/C_n$ the rate at which a NTCB is completed and by $\mu_f = 1/C$ the rate at which a thread completes a TCB in the fall-back path. In general, let the system be in a state s where there are t hardware transactions running concurrently and n threads running a NTCB. Then, a state transition happens if i) any of the t transactions commits; ii) any of the t transactions aborts; or iii) any of the n NTCBs is completed. The first transition is triggered at a rate given by the product of the rate at which a TCB is completed times the (state-dependent) probability that the completion is caused by a commit times the number of concurrent transactions, i.e., $t\mu_{t,s}(1 - p_{a,s})$. Following a similar reasoning, the completion rates for the second and third events are $t\mu_{t,s}p_{a,s}$ and $n\mu_n$, respectively. If a transaction aborts and fall-backs to acquiring the global lock, it induces the abort of the other $t - 1$ transactions and decreases their budget by one. The full set of transition rates is reported in Table I and is based on the above reasoning.

To compute $\mu_{t,s}$ and $p_{a,s}$ we derive analytical expres-

Source state	Destination State	Transition Rate	Corresponding Event
$[\theta_B, .., \theta_0, \theta_n]$	$[\theta_B, .., \theta_0, \theta_n]$	$\theta_n \mu_n (1 - p_t)$	A thread finishes a NTCB and starts another NTCB
$[\theta_B, .., \theta_0, \theta_n]$	$[\theta_B + 1, .., \theta_0, \theta_n - 1]$	$\theta_n \mu_n p_t$	A thread finishes a NTCB and starts a TCB
$[\theta_B, .., \theta_i, ..., \theta_1, 0, \theta_n]$	$[\theta_B + 1, .., \theta_i - 1, ..., \theta_1, 0, \theta_n]$	$\theta_i \mu_{t,s} (1 - p_{a,s}) p_t$	A thread with $i > 0$ retries left commits a TCB and starts another TCB
$[\theta_B, .., \theta_i, ..., \theta_1, 0, \theta_n]$	$[\theta_B, .., \theta_i - 1, ..., \theta_1, 0, \theta_n + 1]$	$\theta_i \mu_{t,s} (1 - p_{a,s})(1 - p_t)$	A thread with $i > 0$ retries left commits a TCB and starts a NTCB
$[\theta_B, .., \theta_i, ..., \theta_1, 0, \theta_n]$	$[\theta_B, .., \theta_i - 1, \theta_{i-1} + 1, ..., \theta_1, 0, \theta_n]$	$\theta_i \mu_{t,s} p_{a,s}$	A thread with $i > 1$ retries left aborts a TCB and restarts
$[\theta_B, .., \theta_i, ..., \theta_1, 0, \theta_n]$	$[0, \theta_B, .., \theta_{i+1}, .., \theta_2, \theta_1, \theta_n]$	$\theta_1 \mu_{t,s} p_{a,s}$	A thread with 1 retry left aborts a TCB and falls-back
$[\theta_B, .., \theta_1, \theta_0, \theta_n]$	$[\theta_B + 1, .., \theta_1, \theta_0 - 1, \theta_n]$	$\mu_f p_t$	A thread completes a TCB in the fall-back path and starts another TCB
$[\theta_B, .., \theta_1, \theta_0, \theta_n]$	$[\theta_B, .., \theta_1, \theta_0 - 1, \theta_n + 1]$	$\mu_f (1 - p_t)$	A thread completes a TCB in the fall-back path and starts a NTCB

Table I: State transition diagram.

sions that consider abort events due exclusively to conflicts between transactions, while neglecting cascading abort events caused by the abort of transactions with only 1 retry left. In fact, even in presence of multiple, concurrent cascading aborts, the transition in the CTMC is triggered by the abort (caused by a conflict or by a capacity exception) that triggered the domino effect in the first place. Also, the transition rates in Table I are computed assuming the independence of the abort events affecting different threads, and cascading abort events are clearly not independent.

Once the CTMC is instantiated with the transition rates, we obtain its stationary probability vector $\vec{\pi}$[1]. Based on $\vec{\pi}$, we compute the global average throughput and abort probability as the weighted average of the probabilities $\vec{\pi}_s$ of being in state s and the corresponding throughput/abort probability in that state. It is at this stage that we account for the effects of cascading aborts triggered when the fall-back path is acquired, by computing adjusted values for $\mu_{t,s}$ and $p_{a,s}$, denoted as $\mu'_{t,s}$ and $p'_{a,s}$. This allows for accurately reflecting these cascading abort dynamics in the computation of the target KPIs.

C. Modelling aborts due to conflicts

As discussed in Section II, TSX employs an eager conflict detection and a "last requester wins" conflict resolution policy. After a hardware transaction T accesses the i-th granule and until its $i+1$-th access, there is an average time interval of length C/L. During this time, T can be aborted because of conflicting accesses by concurrent transactions on any of the i data items T has accessed. If we assume that the sequence of accesses to granules issued by concurrent transactions forms a Poisson process, we can express the probability density function corresponding to the event that a conflicting access is generated at time t, with $t \in [0, C/L]$ as:

$$f_{c,s}(i,t) = H_s(i) e^{-H_s(i)t}$$

$H_s(i)$ is the rate at which concurrent transactions conflicts on any of the i granules accessed by T in state s, and it can be computed as $H_s(i) = \lambda_s P_h(i) P_I$, where: λ_s is the rate at which concurrent threads issue accesses to memory words; $P_h(i)$ is the probability that a concurrent access targets one of the i granules previously accessed by T; P_I is the probability that an access by a concurrent transaction T' to a granule previously read/written by T results in a conflict.

During the execution of a transaction, a thread issues a memory access every C/L time units. Before starting a transaction, however, a thread incurs a cost T_B to initialize the transaction. Similarly, when committing (resp., aborting), the thread incurs a cost T_C (resp., T_A). A transaction does not issue any memory access during these lapses of time. Hence, assuming that $T_C \approx T_A$, a transactional thread executes a memory access, on average, every $(C + T_A + T_B)/L$ time units. Hence, the model computes $\lambda_s = \frac{\theta_{t,s} - 1}{(C + T_A + T_B)/L}$, where $\theta_{t,s}$ is the number of transactional threads that are active in a given CTMC state. Because we are assuming that memory granules are chosen uniformly at random from a pool of cardinality D, $P_h(i) = i/D$.

Finally, two concurrent accesses yield a conflict if at least one of the two is a write: $P_I = 1 - (1 - P_W)^2$.

D. Response time and abort probability

We now compute the mean response time R_t of a single execution of a transaction T using HTM, assuming that transactions can only be aborted because of conflicts. This response time does not include multiple re-executions of the same transaction: R_t is the average time since the (re)start of T and its completion, independently of whether it is successful.

We first introduce the probability that a transaction T manages to successfully perform i memory accesses in state s, $P_{R,s}(i)$. $P_{R,s}(i)$ has a recursive expression, because it is given by the product of the probability that T manages to access $i - 1$ granules and the probability

[1]This can be achieved by solving the set of linear equations expressed by $\pi \cdot Q$=0, where Q is the infinitesimal generator matrix of the CTMC [26]. We use a numerical solver [25] that relies on the QR decomposition algorithm [33], which has $O(n^3)$ time complexity and $O(n^2)$ space complexity, n being the number of states of the CTMC.

that T then manages to access the i-th granule without experiencing aborts due to a data conflict.

The average time between two memory accesses is C/L. The probability that a transaction that has accessed i granules is not aborted in such lapse of time is:

$$\begin{aligned} P_{R,s}(i) &= P_{R,s}(i-1)\left(1 - \int_0^{C/L} f_{c,s}(i-1,t)dt\right) \\ &= P_{R,s}(i-1)\left(e^{-H_s(i-1)C/L}\right) \end{aligned} \tag{1}$$

$P_{R,s}(i)$ is used to obtain $R_{t,s}$. $R_{t,s}$ corresponds to the cost of starting a transaction plus the sum of two contributes: i) one corresponding to the case in which T commits ($R^C_{t,s}$); and, ii) one corresponding to the abort case ($R^A_{t,s}$). Namely $R_{t,s} = T_B + R^C_{t,s} + R^A_{t,s}$. A transaction T commits if it successfully performs L memory accesses and it is not aborted during the final commit operation. In this case, its average execution time, including the final validation phase, is equal to $C + T_C$. The probability that the validation phase is successful is computed as 1 minus the probability that T is aborted in a time window of duration T_C, after having accessed L granules. Hence,

$$R^C_{t,s} = P_{R,s}(L)(C + e^{-H_s(L)T_C}T_C) \tag{2}$$

We now compute $R^A_{t,s}$. Let us first compute the probability density function (PDF) corresponding to the event that T successfully accesses i granules and aborts at time t, with $t \in [0, C/L]$, before accessing the $i+1$-th granule. By leveraging the assumption of independent accesses to granules, this PDF is $P_{R,s}(i)f_{c,s}(i,t)$. The response time corresponding to the event is $iC/L + t$.

The probability that T successfully accesses all the L granules and then is aborted at time t, $t \in [0, T_C]$, during the final validation phase is $P_{R,s}(L)f_{c,s}(L,t)$. The corresponding response time is $T_B + C + t$.

The execution time of T if T manages to perform i accesses and is aborted at time t after the i-th access is equal to $T_B + iC/L + t$. $R^A_{t,s}$ is computed as the weighted average that T is aborted after having accessed i granules and while trying to access the $i+1$-th, with i ranging from 1 to $L-1$. T can also abort during the commit phase, because of a conflicting access towards any of the L accessed granules. Using the shorthand $W = C/L$, we can express $R^A_{t,s}$ as:

$$\begin{aligned} R^A_{t,s} = &\sum_{i=1}^{L-1} P_{R,s}(i) \int_0^W iWtH_s(i-1)e^{-H_s(i)t}dt \\ &+ CP_{R,s}(L)(1 - e^{-H_s(L)T_C}) \end{aligned}$$

We can now compute the abort probability $p_{a,s}$ and average rate $\mu_{t,s}$ at which transactions complete, in a state s of the CTMC, as:

$$p_{a,s} = 1 - P_{R,s}(L)e^{-H_s(L)T_C}, \qquad \mu_{t,s} = R^{-1}_{t,s} \tag{3}$$

E. Modeling capacity aborts

The model presented so far captures only the aborts triggered by conflicts between transactional operations. We now extend the model to capture also capacity aborts.

We note $P(c \leq i)$ the probability that a transaction experiences a capacity abort in any of its first i memory accesses, conditioned to that it does not experience a data conflict. Assuming to know how to obtain $P(c \leq i)$, we can compute an updated version of the probability that T successfully manages to access i granules as

$$P''_{R,s}(i) = P_{R,s}(i)(1 - P(c \leq i)) \tag{4}$$

$P''_{R,s}(i)$ takes into account data conflicts, aborts due to the activation of the fall-back path and capacity aborts. By replacing $P_{R,s}$ with $P''_{R,s}(i)$ in Equations 2-3, we can also update the variable $p_{a,s}$ and $\mu_{t,s}$, which, we recall, are used in the definition of the CTMC's transition rates.

In the light of the findings reported in Section III-B, our model assumes that a capacity abort can only be triggered by the eviction of a cache line that was written by a transaction. In this section we present a model to calculate the probability of capacity exception at the i-th accessed granule assuming a N-way associative cache with a given number of sets. To compute the probability that a transaction experiences a capacity abort at its i-th access we compute the probability that two events jointly happen: i) the corresponding granule is stored in a full set of the L1 cache, and ii) the cache line selected for eviction corresponds to a written granule.

We cast the problem of finding this probability to a variation of the balls-into-bins problem. In our settings, a ball is an accessed granule, the bins (β) are the sets of the cache, and the capacity of each bin (γ) is the associativity of the cache.

Each memory access of a transaction is a ball thrown at a bin chosen uniformly at random. The variation with respect to the classic bin-into-balls-problem is two-fold: i) a ball can be a write or a read ball (with probability P_W, resp. $1 - P_W$); ii) if a bin is full, a read ball can be removed from it (if selected by the eviction policy) to make room for another ball.

Let us start by considering the simpler case in which only write accesses are performed. We define a valid sequence of length i, a sequence of i ball throws such that no bin overflows, i.e., no bin receives more than γ balls. The total number of possible sequences of length i with β bins is β^i. These sequences also include invalid ones, i.e., sequences in which bins can have been assigned more than γ balls. We note $M_{\beta,\gamma,i}$ the number of valid sequences after i balls have been thrown. Then, the probability that at least one bin experiences an overflow after throwing I balls, $P(c \leq i)$, is:

$$P(c \leq i) = 1 - \frac{M_{\beta,\gamma,\mathrm{i}}}{\beta^i} \tag{5}$$

We compute $M_{\beta,\gamma,\text{i}}$ as follows. Assume that exactly x bins have been filled after i balls. The number of combinations of balls-to-bins allocations is given by the product of a) the number of ways in which the x bins can be filled with $x\gamma$ balls and b) the number, ν, of ways in which the the remaining $i - x\gamma$ balls can be assigned to the remaining $\beta - x$ bins without fully filling them. It follows that ν can be computed as $M_{\beta\text{-x},\gamma\text{-1},\text{i-x}\gamma}$, i.e., the number of ways in which the remaining $i - x\gamma$ balls can be thrown in $\beta - x$ bins in such a way that, at most, every bin is filled with $\gamma - 1$ balls.

The minimum value for x is the number of bins that are filled if balls are assigned to bins in a round-robin fashion: $\min_\gamma = \max(0, i - \beta(\gamma - 1))$. The maximum value for x is the number of bins that get filled if the balls are thrown to the same bin until it gets full: $\max_\gamma = \lfloor i/\beta \rfloor$. These x bins can be chosen out of the total β possible in $\binom{\beta}{x}$ ways. Finally, the number of ways in which γx balls can be thrown in x bins in such a way that all the x bins are filled is $\prod_{y=0}^{\text{x-1}} \binom{\text{i-y}\gamma}{\gamma}$. The resulting equation for $M_{\beta,\gamma,\text{i}}$ is then

$$M_{\beta,\gamma,\text{i}} = \sum_{x=\min_\gamma}^{\max_\gamma} M_{\beta\text{-x},\gamma\text{-1},\text{i-x}\gamma} \binom{\beta}{x} \prod_{y=0}^{\text{x-1}} \binom{\text{i-y}\gamma}{\gamma} \quad (6)$$

We now describe how we extend the model to take into account scenarios in which transactions issue a mix of reads and writes. In this case, the number of valid sequences of a given length i is larger than for the case of $P_W = 1$, since if a full bin contains at least a read ball b, it can still accommodate an additional (read/write) ball, provided that b is selected by the eviction policy. Given the combinatorial nature of the problem, the number of scenarios to be accounted for in order to derive an exact probabilistic solution increases dramatically for the case of $P_W \neq 1$, along with the complexity and computational cost of the resulting model.

We propose therefore an approximate solution technique that is based on the following approach. Let us introduce the notations: $a)$ $P(c \leq i_{P_W})$, to refer to the probability of having a capacity abort upon during any of the first i accesses of a transaction that executes writes with probability P_W; $ii)$ $P(c = i_{P_W} \wedge \neg c < (i-1)_{P_W})$, to refer to the probability of having a capacity abort exactly at the i-th access and of not incurring capacity aborts during the previous i-1 operations, where each operation is a write with probability P_W.

We express $P(c = i_{P_W} \wedge \neg c < (i-1)_{P_W})$ as:

$$P(c = i_{P_W} | \neg c < (i-1)_{P_W}) P(\neg c < (i-1)_{P_W}) \quad (7)$$

Next we observe that the probability of having a capacity exception at operation i is not affected by whether this operation is a read or write , but only by whether the corresponding ball i hits a full bin and causes the "eviction" of a write ball. Hence:

$$P(c = i_{P_W} | \neg c < (i-1)_{P_W}) = P(c = i | \neg c < (i-1)_{P_W})$$

Next, we introduce the following approximation:

$$P(c = I | \neg c < (I-1)_{P_W}) \approx P(c = I | \neg c < (I-1)) P_W$$

namely, we approximate the conditioned probability of having a capacity after i *read/write* accesses with the conditioned probability of having a capacity after i *write* accesses scaled down by a factor P_W. The latter scaling factor reflects the fact that $P(c = i | \neg c < i - 1)$ is computed assuming that all the full bins after i-1 balls contain exclusively write balls. Conversely, if transactions issue write operations with probability P_W, on average the full bins after $i - 1$ throws will contain only a fraction of write ball equal to $P_W \gamma$ over a total of γ balls. This is an approximation, which, as we will show in Section V, yields good accuracy for P_W values larger than 1%. In fact, as discussed in Section III-B, assuming P_W values larger than 1% is also a necessary condition for modelling accurately the cache dynamics by modelling solely the L1 dynamics.

$P(c = i | \neg c < i - 1)$ can be computed by expressing it as $(P(c \leq i) - P(c \leq i - 1))/(1 - P(c < i - 1))$ and exploiting Eq. 5, using the definition of conditioned probability. $P(\neg c < (i-1)_{P_W})$, in Eq. 7, can be expressed as:

$$1 - \sum_{j=1}^{i-1} P(c = j_{P_W} \wedge \neg c < (j-1)_{P_W})$$

and can be computed recursively by setting $P(c = 1_{P_W} \wedge \neg c < 0_{P_W}) = 1$.

Finally, $P(c \leq i_{P_W})$ can simply be expressed as the sum of the probabilities of having a capacity abort exactly at operation j, and not earlier, for all $j < i$:

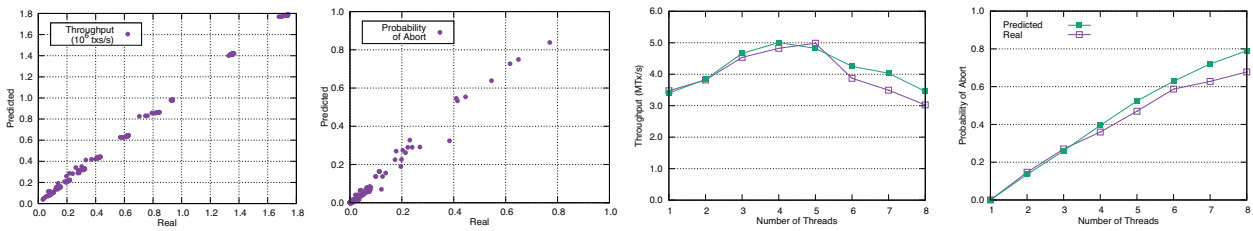
$$P(c \leq i_{P_W}) = \sum_{j=0}^{i} P(c = j_{P_W} \wedge \neg c < (j-1)_{P_W})$$

F. Computing the Target KPIs

We now describe how to obtain the average throughput X, average transaction response time R_t^* and average abort probability P_A. Unlike the response times computed in the former section, R_t^* represents the execution of a transaction since its first begin to the time it commits. Namely, R_t^* includes possible multiple re-executions of a transaction and the possible final execution in the fall-back path.

We start by computing the state transitions for the CTMC (using the values of $p_{a,s}$ and $\mu_{t,s}$ derived in Section IV-E that encompass both conflict and capacity aborts), and by solving it to obtain the vector $\vec{\pi}$ of the states probabilities.

Next, we compute the adjusted abort probability, $p'_{a,s}$, and average transaction execution rate, $\mu'_{t,s}$, in each state s, keeping into account that a transaction can abort, either directly due to a conflict or capacity abort, or, indirectly,



(a) Throughput (10^6 txs./sec.) MAPE = 8.12%, R = 0.9989.
(b) Abort probability MAE = 4.94 %, R = 0.9923.
(c) Throughput L=8, B=4, D=128, P_W=0.5
(d) Abort probability L=8, B=4, D=128, P_W=0.5

Figure 2: Real vs predicted KPIs in workloads with negligible capacity aborts.

because of the abort of a concurrent transaction with only 1 retry left. Such transactions, which we call *dangerous* because they can trigger the cascading abort of all concurrent transactions, are not accounted for in $p_{a,s}$ and $\mu_{t,s}$ (as these variables are used solely to derive the CTMC's transition rates, which need to account only for independent abort events), but they do have an impact on the actual abort probability and average transaction execution time of the system.

To ease the presentation, we postpone the discussion on how to compute $p'_{a,s}$ and $\mu'_{t,s}$, and explain first how to compute P_A, X and R^*_t.

The average abort probability P_A is computed as the weighted average of $p'_{a,s}$ across all the states with no threads in the fall-back path (as no transaction can execute and, hence, abort in such a state):

$$P_A = \sum_{s(i,f=0,n)\in S} \vec{\pi}_s p'_{a,s} \tag{8}$$

where S denotes the set of states of the CTMC and the notation $s(i,f,n)$ indicates the state corresponding to i active hardware transactions, f in the fall-back path and n non-transactional active threads.

Let us denote with X^t the transactional throughput, i.e., the rate at which transactions commit, and with X the global throughput, i.e., the rate at which any thread in the system completes a code block (either transactional or not). Both throughputs are computed as the weighted average of the system being in a state s_i times the corresponding throughput in s_i. On its turn, the global throughput in s_i is the sum of the rates corresponding to the completion of a NTCB or the commit of a TCB.

$$X = \sum_{s(i,f=0,n)\in S} \vec{\pi}_s (i\mu'_{t,s}(1-p'_{a,s})+n\mu_n) + \sum_{s(i,f\geq 1,n)\in S} \vec{\pi}_s (n\mu_n+\mu_f)$$

This equation captures the fact that in a state in which there is at least one transaction in the fall-back path there is only one transaction contributing to the transactional throughput, by committing with a rate $\mu_f = \frac{1}{C}$. In a state s in which $f = 0$, instead, the i hardware transactions all contribute to the throughput of the system, with a rate $i\mu'_{t,s}(1-p'_{a,s})$. The transactional throughput is simply: $X^t = Xp_t$.

We exploit Little's law [27] to obtain the response time of a transaction, R^*_t. We first express X as the product of the number of active threads θ and the inverse of the average response time of a code block, whether transactional or not, R^*. Once we obtain R^* we note that it corresponds to the weighted average of the response time of a transactional code block R^*_t and of a non-transactional code block R^*_n. Because the system is stable, the probability that a successfully executed code block is (non) transactional corresponds to the probability that a (non) transactional code block is started. Hence, $R^* = p_t R^*_t + (1-p_t)R^*_n$. Because R^*_n is equal to C_n and it is given as input to the model, we can solve the equation and obtain R^*_t.

Modeling aborts due to fall-backs. Finally, we describe how to compute the per-state abort probability, $p'_{a,s}$, and average transaction execution rate, $\mu'_{a,s}$, which we have introduced and used above to compute the model's KPIs. Let us consider a state with n non-dangerous transactions and d dangerous ones. We model the increase in the abort probability of T due to concurrent dangerous transactions by computing an adjusted rate at which T can abort. Such rate does not encompass only the rate at which other transactions can issue conflicting accesses with T, but also the rate at which T aborts due to the abort of some dangerous transaction.

We express the adjusted rate as the previous rate $H_s(i)$ (see Section IV-C) plus the rate at which dangerous transactions abort. The rate at which a dangerous transaction aborts (due to a conflict or a capacity exception) is computed as $\mu_{t,s}p_{a,s}$, obtained as discussed in Section IV-E. The adjusted rate at which a non-dangerous transaction can abort after having accessed i granules is then $H^n_s(i) = H_s(i) + d\mu_{t,s}p_{a,s}$. The adjusted rate is different for a dangerous transaction, since it can only abort because of the conflict of $d-1$ other dangerous transactions. Hence, $H^d_s(i) = H_s(i) + (d-1)\mu_{t,s}p_{a,s}$.

The first step to compute $p'_{a,s}$ is to derive the probability that a transaction successfully accesses i granules, despite both direct aborts and cascading aborts due to dangerous transactions. We again distinguish between the case of non-dangerous transactions ($P'^{n(i)}_{R,s}$) and

dangerous transactions ($P'^{d(i)}_{R,s}$). Both probabilities take the value 1 for $i = 1$. For $i > 1$, following the same reasoning applied when computing $P_{R,s}(i)$ and $P''_{R,S}$

$$P'^{n}_{R,s}(\text{i}) = P'^{n}_{R,s}(\text{i-1})e^{-H^{n}_{s}(\text{i-1})\,C/L}(1 - P(c \leq i))$$

$$P'^{d}_{R,s}(\text{i}) = P'^{d}_{R,s}(\text{i-1})e^{-H^{d}_{s}(\text{i-1})\,C/L}(1 - P(c \leq i))$$

Taking into account the vulnerability window T_c corresponding to the commit operation, we obtain $p'_{a,s}$:

$$p'_{a,s} = 1 - \Big(\frac{nP'^{n}_{R,s}(L)e^{-H^{n}_{s}(L)T_c} + dP'^{d}_{R,s}(L)e^{-H^{d}_{s}(L)T_c}}{n+d}\Big) \quad (9)$$

We also obtain adjusted values for the response times of an execution of dangerous (R^d) and non-dangerous (R^n) transactions. To compute them, we use Equation 2 and Equation IV-D, where we substitute $H(i)$ accordingly. Thus, we compute the average response time of a single hardware execution of a transaction:

$$R'_{t,s} = \frac{n}{n+d}R_{d,s} + \frac{d}{n+d}R_{n,s}$$

The adjusted $\mu'_{t,s}$ is obtained as $\mu'_{t,s} = 1/R'_{t,s}$.

V. Model Validation and Evaluation

This section reports the results of a validation study that compares the KPIs predicted by the proposed model with those achieved when executing on our target experimental platform (see Section III). In order to stress the prediction accuracy of the presented model, we use a synthetic benchmark that generates diverse workloads. The micro-benchmark launches θ concurrent threads bounded to different physical cores, hence, not sharing private caches and other resources. These threads start transactions that perform L accesses uniformly at random over a granule pool of size D.

We first focus the study on validating the accuracy of the proposed model to predict the contention dynamics among transactions and due to the fallback path activation. To minimize the probability of capacity aborts, we initially consider short transactions and generate about 380 workloads/configurations varying the model's parameters as follows: $L \in \{2, 5, 10, 20\}$, $D \in \{512, 2048, 8192, 32768\}$, $\theta \in \{2, 4, 8\}$, $B \in \{2, 4, 6\}$, $P_W \in \{0.5, 1\}$. Figures 2a and 2b report a scatter plot comparing the real and predicted KPI values for the considered workloads. The reported data highlights the accuracy of the proposed model in predicting both the throughput and abort probability of the system: the Mean Absolute Error (MAE) for the abort rate is less than 5% and the Mean Percentage Absolute Error (MAPE) for the throughput around 8%; the Pearson correlation factor (R) is in both case larger than 0.99. Figures 2c and 2d report the predicted/real throughput and abort probability values while varying the thread count in a conflict intensive workload. Also in such a challenging workload,

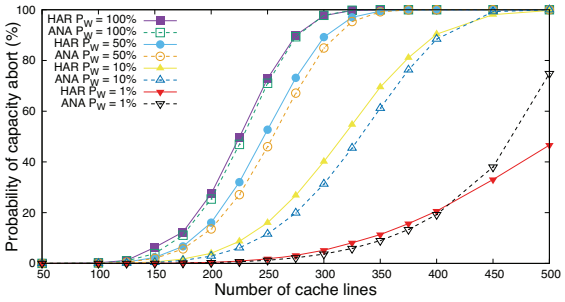


Figure 3: Validating the analytical model (ANA) for the probability of capacity aborts vs a real system (HAR)

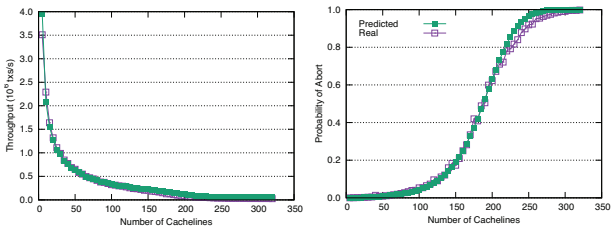
where the abort probability spikes up to approx. 70%, we can appreciate the high accuracy of the model in predicting the actual system's performance.

Next, we focus on validating the modelling of the probability capacity aborts ($P(c \leq i)$) in absence of contention among transactions. To this end we consider a single threaded execution, where we varied the number of distinct granules (L) accessed by transactions, set $D = 8192$ and vary $P_W \in [0.01, 1]$. The results of this study are reported in Fig. 3 and confirm the high accuracy of the model, which attained a MAE of 2.12%. As expected, the larger errors are introduced for smaller values of P_W. This is due to the approximation introduced in Section IV-E, which we use to compute $P(c \leq i)$ when $P_W < 1$. Nonetheless, this study shows that the proposed approximation achieves good accuracy even for P_W values as small as 1%.

Next, we assess the model's accuracy with workloads that generate aborts induced both by capacity exceptions and conflicts (see Fig. 4). To this end, we set Θ =4, D =100000, P_W =0.5 and varied L ∈[5, 320]. Also in this case, the experimental data confirms the high accuracy of the model.

Finally, Figure 5 reports the memory and time required to solve the model while varying the two main factors that affect its spatial and temporal complexity, i.e., θ and B. The top plot reports the model solution time, using a single threaded implementation based on the Eigen [25] numerical library, running on an Intel E3-1270v3@3.50GHz with 32GB of RAM (Ubuntu 12.04). We observe that the model can be solved in less than 10 seconds for up to 100 threads, when $B = 1$. This result is relevant, since $B = 1$ when using TSX's in Hardware Lock Elision mode [36] and given that, currently, the largest TSX-enabled Intel's processor has 28 physical cores. As B grows, though, the solution time grows very quickly, and for $B = 5$ it requires 3 minutes when $\theta = 9$, with a $3\times$ increase with respect to $\theta = 8, B = 5$. This is expectable, since the CMTC's state space, S, grows combinatorially, as previously discussed.

The data in the bottom plot evaluates the spatial



(a) Throughput (10^6 txs./sec.) Θ=4, B=4, D=100000, P_W=0.5

(b) Abort probability Θ=4, B=4, D=100000, P_W=0.5

Figure 4: Workloads generating aborts due to both contention and capacity.

complexity of the model, which is dominated by the CTMC's generator matrix, Q. An interesting observation that we can draw from this plot is that Q, despite having size $|S| \times |S|$, is actually very sparse. This is clearly illustrated by the bottom plot, which reports the total number of elements in Q and the actual non-zero entries of Q: for the largest models (e.g., $B = 5, \theta = 9$) less than 4% of the matrix holds non-zero values, i.e.,$\tilde{1}$50K cells, for a memory occupation of less than 10 MB. This suggests that the model's solution is CPU-bound, rather than memory-bound, and that it should be therefore possible to boost it using parallel implementations, possibly exploiting hardware accelerators (like GPUs).

VI. Related Work

The most closely related works lie in the area of analytical modelling of the performance of transactional systems. A number of analytical models of concurrency control for database management systems have been proposed in the literature [35], [1], [37], [9]. More recently, several analytical models have been proposed for the concurrency control algorithms adopted by software implementations of TM [38], [21], [8], [14], [12], [32], [13]. The key difference with respect to these approaches is that in our model we consider peculiar characteristics of the concurrency control of HTM, including the co-existence of optimistic techniques (i.e., speculative execution of parallel transactions) and of a sequential/pessimistic fallback path. Indeed, to the best of our knowledge, the analytical model presented in this work is the first one to target HTM systems.

Black box techniques for throughput prediction are present in the literature for the case of STM [5], [31], and also in HTM either to predict its throughput [29] or to improve its performance by tuning the TM parameters [11], [15], [10]. Unlike the white-box analytical model presented in this model, which can be instantiated by simply providing a few parameters as input, these black box models require an extensive training phase. Indeed, the accuracy of black-model is known to be

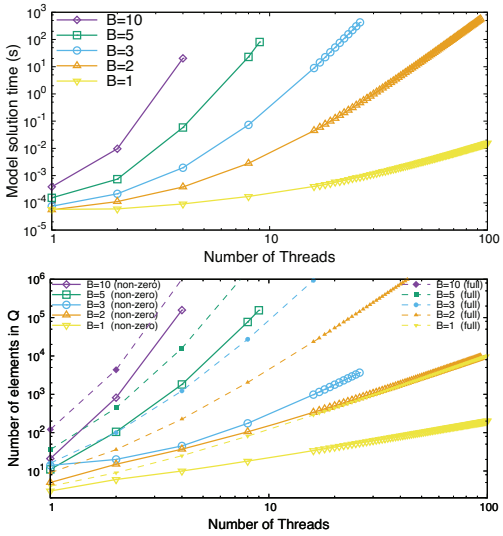


Figure 5: Model solution time (top) and memory consumption (bottom).

strongly influenced by the representativeness of the data collected during the training phase [2], [12].

VII. Conclusions

This paper introduces the first analytical model of an HTM system. The presented model targets TSX, a mainstream HTM implementation included since 2013 in Intel's processors, and captures complex, non-linear performance dynamics reflecting the joint impact of architectural choices (e.g., cache size and geometry), workload characteristics (e.g., number of accessed memory words) and specific features of the employed conflict resolution scheme (e.g., the co-existence of optimistic and pessimistic execution modes). The model has been validated using a real system, achieving high accuracy in a broad range of workloads.

Another relevant contribution of our work consists in having shed lights, via a set of ad-hoc experiments and simulations, on several internal and undisclosed mechanisms of TSX: besides determining the conflict detection and resolution schemes employed in TSX, we have also investigated the cache capacity limitations in presence of mixed read/write workloads, inferring undisclosed information on how transactional meta-data is stored and managed. Not only the insights gained through this study allow for a better understanding of performance dynamics of TSX; they were also crucial to enable the development of analytical models capturing both capacity and conflict aborts.

Acknowledgments

This work was supported by Portuguese funds through Fundação para a Ciência e Tecnologia via projects UID/CEC/50021/2013 and PTDC/EEISCR/1743/2014.

REFERENCES

[1] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Transactions on Database Systems*, 12(4):609–654, 1987.

[2] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., New York, NY, USA, 2006.

[3] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[4] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: why is it only a research toy? *Queue*, 6(5):46, 2008.

[5] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra, and J. F. Méhaut. A machine learning-based approach for thread mapping on transactional memory applications. *18th International Conference on High Performance Computing, HiPC 2011*, 2011.

[6] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '11*, pages 39–51, 2011.

[7] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78, 2010.

[8] P. Di Sanzo, B. Ciciani, R. Palmieri, F. Quaglia, and P. Romano. On the analytical modeling of concurrency control algorithms for Software Transactional Memories: The case of Commit-Time-Locking. *Performance Evaluation*, 69(5):187–205, 2012.

[9] P. Di Sanzo, B. Ciciani, F. Q. Sapienza, and P. Romano. A performance model of multi-version concurrency control. *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS*, 2008.

[10] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, and P. Romano. Proteustm: Abstraction meets performance in transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 757–771, New York, NY, USA, 2016. ACM.

[11] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker. Identifying the optimal level of parallelism in transactional memory applications. *Computing*, 97(9):939–959, Sept. 2015.

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

[13] D. Didona and P. Romano. On Bootstrapping Machine Learning Performance Predictors via Analytical Models. In *ICPADS*, 2015.

[14] D. Didona, P. Romano, S. Peluso, and F. Quaglia. Transactional Auto Scaler: Elastic Scaling of In-memory Transactional Data Grids. *ACM Transactions on Autonomous and Adaptive Systems*, 9(2):125–134, 2014.

[15] N. Diegues and P. Romano. Self-tuning Intel Restricted Transactional Memory. *Parallel Computing*, 50:25–52, dec 2015.

[16] N. Diegues, P. Romano, and L. Rodrigues. Virtues and Limitations of Commodity Hardware Transactional Memory. *Pact*, pages 3–14, 2014.

[17] A. Dragojevic, R. Guerraoui, and M. Kapalka. Stretching transactional memory. *ACM SIGPLAN Notices*, 44:155, 2009.

[18] P. Felber, C. Fetzer, and T. 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.

[19] W. Feller. *An Introduction to Probability Theory and Its Applications.*, volume 1. Wiley, 2nd edition, 1950.

[20] B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. 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.

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

[22] M. Herlihy, M. Herlihy, J. E. B. Moss, and J. E. B. Moss. Transactional memory. *ACM SIGARCH Computer Architecture News*, 21(2):289–300, 1993.

[23] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

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

[25] B. Jacob and G. Guennebaud. Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. *http://eigen.tuxfamily.org/index.php?title=Main_Page*, 2017-06-26.

[26] L. Kleinrock. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.

[27] J. D. C. Little. A Proof for the Queuing Formula: L = λ W. *Operations Research*, 9(3):383–387, jun 1961.

[28] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, pages 144–157, New York, New York, USA, 2015. ACM Press.

[29] A. Nguyen. Investigation of Hardware Transactional Memory. Master's thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2015.

[30] Peter Bergner, Alon Shalev Houfater, Madhusudnanan Kandeasamy, David Wendt, Suresh Warrier, Julian Wang, Bernhard King Smith, Will Schmidt, Bill Schmidt, Steve Munroe, and Tullo Magno. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, 2015.

[31] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. 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.

[32] D. Rughetti, P. D. Sanzo, B. Ciciani, and F. 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.

[33] J. Stoer and R. Bulirsch. *Introduction to numerical analysis*, volume 12. Springer Science & Business Media, 2013.

[34] Y. Tay. Analytical Performance Modeling for Computer Systems. *Synthesis Lectures on Computer Science*, 2(1):1–116, apr 2010.

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

[36] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013.

[37] P. S. Yu, D. M. Dias, and S. S. Lavenberg. On the analytical modeling of database concurrency control. *Journal of the ACM*, 40(4):831–872, 1993.

[38] 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.