Towards White-Box Modeling of Hardware Transactional Memory Systems

Daniel Castro
INESC-ID & Instituto Superior Técnico, University of Lisbon
daniel.castro@tecnico.ulisboa.pt

Diego Didona
EPFL
diego.didona@epfl.ch

Paolo Romano
INESC-ID & Instituto Superior Técnico, University of Lisbon
romano@inesc-id.pt

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's processors since 2013 (Haswell family).

An inherent difficulty that lies along the path to build white-box models of commercially available HTM systems is that their internal 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 through 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.

Keywords transactional memory, hardware, performance modeling, concurrency control

1. Introduction

Transactional Memory (TM) [25] is an emerging paradigm aimed to simplify concurrent programming by bringing the familiar abstraction of atomic and isolated transactions, originally proposed in the DBMS's context [31], to the domain of parallel computing.

Over the last two decades, the research on TM has led to many different designs and implementations, either in software [21, 20, 8], hardware [26, 30], or combinations thereof [7]. Software based TM (STM) systems rely on software instrumentation to trace memory accesses and detect the concurrent execution of conflicting transactions. Due to its purely software-based nature, STM allows for supporting a broad range of alternative concurrency control algorithms. However, the overheads resulting from software-based tracking of transactions' data accesses can, at least in some workloads, severely hinder application's performance [5].

These instrumentation 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 [26, 30] are built as relatively non-intrusive extensions of the cache coherency algorithm and, as such, have a best effort nature: if a transaction performs a larger number of accesses that the processor's cache can accommodate, the transaction is never going to be successfully executed in hardware, and a conservative, lock-based fall-back path has to be used to ensure progress.

Because of its inherently restricted nature, HTM does not represent the definitive solution to the TM performance problems. Recent work has shown, in fact, that for some applications/workloads STM can deliver superior performance than HTM [19, 23, 29]. Furthermore, the performance and effectiveness of HTM has been shown to be largely dependent on the correct tuning of the software logic that governs the retry policy of hardware transactions in presence of different types of aborts (e.g., due to conflicts or capacity exceptions) [18, 12]. Ultimately, these works indicate that the performance dynamics of HTM can be strongly affected by a number of workload dependent parameters and architectural design choices, which makes the problem of predicting the performance achievable by HTM-based applications a very challenging task.

This paper makes a step towards clarifying our understanding of HTM's performance dynamics by developing the first, to the best of our knowledge, white-box analytical model of the HTM system employed in a mainstream processor by Intel, i.e., the Xeon CPU of the Haswell family. In particular, we focus on the modelling of two mechanisms that play a key role in determining the performance of HTM systems: the concurrency control scheme employed to detect conflicting memory accesses , and the management of transactional meta-data in the processor's cache.

We assess the accuracy of the proposed analytical model via a validation based on a real system and a set of synthetic micro-benchmarks that generate heterogeneous workloads. The experimental results show that the model can predict application's throughput and abort rate with high accuracy.

2. Background on HTM

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[1]. This is due

[1] The only notable exceptions being IBM zEC12's HTM implementation, that guarantees that transactions are eventually committed, provided that they meet some constraints on the instructions they execute and on the memory regions that they access

to the fact that existing HTM implementations use the processor's cache hierarchy to buffer transactional reads and writes, 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 what we call *capacity* aborts. Indeed, in existing HTM implementations, a transaction can also experience other type of spurious (i.e., not imputable to conflicting accesses) aborts, because of external events like page faults, context switches and system calls.

HTM implementations must, thus, rely on an additional fall-back mechanism in order to guarantee that a transaction eventually succeeds in committing. The default approach is to allow transactions to execute in a software fall-back execution path, guarded by a single *global* lock. When a hardware transaction aborts, it can acquire the global lock instead of retrying its execution in hardware. Hardware transactions must read the lock as free upon starting, as well as before starting, in order to avoid the, so called, lemming effect [11]. This means that as soon as a transaction T starts executing in the fall-back path, ongoing hardware transactions abort and no hardware transaction can (re)start until T releases the lock. Naturally, the global lock also serializes the execution of multiple transactions trying to execute in the fall-back path.

The policy governing the retry logic of a transaction (upon an abort event) can be implemented either in hardware (as for the case of Intel's HLE interface) or in software. The latter approach provides more flexibility, allowing for tuning not only the maximum number of attempts to execute in hardware, which we call *budget*, but also how such budget should be consumed in presence of different abort types, e.g., by exhausting it immediately, or halving it, upon a capacity abort [4, 18]. In the presented model we consider that the budget is decreased linearly, i.e., upon an abort, of any type, the budget is reduced by 1.

3. Dissecting Intel's HTM implementation

We now focus on investigating two key aspects of Intel's HTM implementation: i) how it manages conflict among concurrent transactions, and ii) how transaction's metadata are maintained in the processor's cache and what impact this has on the capacity limitations perceived by transactions generating different types of workloads.

It should be noted that Intel has not disclosed details on the internal mechanisms employed by its HTM implementation. So the information reported in the following is either based on previous external studies [29, 34], or inferred via new experiments designed explicitly to shed light on the internals of Intel's HTM implementation. All the experimental results reported in this section and in the remainder of this paper are based on a Xeon E3-1275 v3 running at 3.5GhZ, equipped with 32 GB Ram and Ubuntu 12.04.02LTS.

3.1 Conflict detection and resolution.

Existing literature [29, 34] has already pointed out that Intel's HTM implementation relies on an eager conflict detection scheme, i.e., when a conflict between two transactions materializes, one of the two transactions is immediately aborted. This is in contrast to some STM implementations, which detect (certain types of) conflicts only at commit time, in a, so called, lazy fashion. Another relevant aspect of the conflict detection schemes integrated by existing HTM implementations is that, since they are built on top of a pre-existing cache coherency protocol, the conflict detection granularity coincides normally with a cache line, which is, for the case of our target Intel processor, 64 bytes long.

The conflict resolution policy used in Intel's HTM implementation, i.e., which transaction is aborted in presence of a conflict, is an aspect that, to the best of our knowledge, is undocumented by Intel and has not been investigated by previous external studies. In order to tackle this issue, we have designed a simple experimental

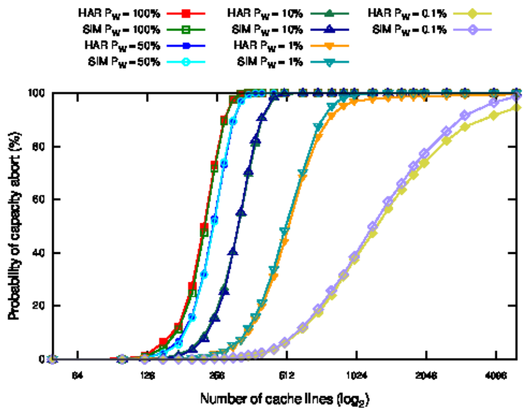


Figure 1: Probability of incurring a capacity abort when accessing a different number of cache lines. Comparing a real Intel system (HAR) against a simulator modelling only L1 (SIM).

test 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. Our experimental test showed that, at least in the Intel Xeon processor targeted by our study, the conflict resolution uses a "Last requester wins" policy, i.e., if two concurrent transactions T_1 and T_2 conflict on a memory address (i.e., both transactions access it and at least one of the two accesses is a write), the first transaction to have accessed that address is consistently the one to be aborted.

3.2 Capacity limitations.

Intel has not disclosed how transactional metadata (e.g., addresses of the values read and written by transaction) is maintained by its HTM implementation, but several previous studies [29, 34] have already partially answered this question.

The conclusions of these studies, which have been also confirmed with our experiments, can be summarized as follows:

- The writes of transactions 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 full L1 data cache: around 450 cache lines vs a total of 512. Nguyen [34] hypothesized that this reduction of the effective capacity of L1 cache could be explained by considering that a transaction must also have sufficient space to store other program's 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 around half of the total cache lines available in L3 (actually around one third in our experiments). In the light of these observations, Nguyen [34] hypothesised that the transactional reads are maintained in L3. In fact, unlike L1, is shared among all the cores of the same processor, as well as by programs' code and data — which may justify why the transaction's read capacity is smaller than the full L3's size.

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

In order to answer these questions we built a simulator of a L1 cache that uses the same geometry of our reference Intel's processor (8-way associative, 64 sets, 64 bytes cache lines, 32KB capacity) and implements a Least Recently Used (LRU) eviction policy. We will use the simulator to validate our assumptions on the internal mechanisms employed by the considered HTM implementation, by comparing 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 program.

Size of the additional transactional metadata. To determine the size of the additional metadata stored by transactions, we designed the following experiment. We occupy P subsequent cache lines, starting from a position at random in the simulated cache, so 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 best matching between simulations and real execution is 2, a value that appears reasonable especially if one considers that the transactional metadata may not be cache line aligned and hence span 2 cache lines even if occupying much less than 128 bytes.

Capacity with mixes of read/write operations. Previous works characterizing the transactional capacity of HTM implementations, e.g., [34, 29] have focused on workloads composed solely by either read-only or write-only transactions. In Figure 1 we report 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, and $1 - P_W$ of being a read. The data in Figure 1 reveals that halving the number of writes issued by a transaction (P_W=0.5) does not lead to doubling 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 has to be, for non-negligible values of P_W, largely dependent on dynamics taking place at the L1. More in detail, our hypothesis is the following. 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 read or written by the same transaction (as well as of any additional transactional metadata stored in L1). If the evicted cache line corresponds to an address written by the transaction, then this undergoes a capacity exception. If, instead, the evicted cache line had instead been read, the the transaction does not have to abort, since the metadata for tracking that read are still stored in L3.

We tested our hypothesis using the same L1 simulator mentioned above, and, as it can be observed in Figure 1, we obtain a very close match between simulation and real system for values of P_W as small as 1%. Below this value, as expectable, the likelihood of incurring evictions of cache lines in the transaction's readset 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 shared nature of L3 in typical multi-core architectures.

4. Analytical model

This section is devoted to derive a white-box, analytical model of the concurrency control mechanism adopted by the HTM system of Intel Xeon processors (Haswell family).

We start, in Section 4.1, by presenting the key model's parameters and its underlying assumptions. Next, in Section 4.2, we illustrate the methodology we adopted to derive the proposed analytical model. In Section 4.3 we discuss how the model is solved to produce performance predictions. Finally, Section 4.4 discusses how the presented concurrency control model is extended and coupled with models aiming at capturing the probability of capacity aborts.

4.1 Key model parameters and assumptions

We consider a HTM system with θ threads that execute in closed loop either a transactional code block (TCB) or a non-transactional code block (NTCB). A TCB has an average service time, i.e., CPU demand, of C time units, a NTCB has an average service time C_n, and a serialized TCB has an average service time C_f. The time to complete a TCB in the software fall-back path may also account for the extra cost of acquiring the global lock. The hardware path has additional costs for starting (T_B) and committing (T_C) a transaction.

Transactions re-execute as many times as necessary to successfully complete. When a transaction starts for the first time, it is assigned a budget B of attempts to execute in hardware. This budget is decreased by one whenever a transaction aborts. When the transaction reaches the fall-back path it runs serially until it commits and releases the lock. When a thread completes a NTCB or successfully completes a transaction, it starts a new transaction with probability p_t and a new NTCB with probability $1 - p_t$.

A transaction accesses on average L distinct memory words, or *granules*. The timing of such accesses is spread uniformly at random during a transaction's lifetime. Namely, 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 noted P_W. The probability that a memory access is a read is, hence, $1 - P_W$. Table **??** in Appendix summarizes the input parameters of the model.

The model only considers conflicts stemming from concurrent accesses to the same granule. This means that the model does not encompass cache aliasing effects, and also that the memory accesses issued by threads running NTCBs do not interfere with the execution of transactional threads. The only notable exception to this last assumption is represented by conflicts caused by the concurrent acquisition of the global lock used on the fall-back path (which is acquired by the fallen back transaction outside the scope of a hardware transaction). The model also does not consider aborts caused by asynchronous interrupts or page faults. This last assumption simplifies the development of the model without significantly impacting its predictive accuracy.

Capacity aborts involving read-only transactions occur after thousands of granules are accessed, as is explained in Section 2. Hence, it is more likely transactions to abort due to time constraints than due to capacity. For this reason we only consider workloads in which the probability of writing is larger than 1%. This allows us to produce a model that considers the dynamics of L1 cache exclusively. Our preliminary experimental study and previous work [19], in fact, show that the incidence of spurious aborts is typically much lower than capacity and conflict-induced aborts.

The model further assumes that no hardware transaction (re)starts when there is at least one transaction waiting to acquire the lock for

Source state	Destination State	Transition Rate	Corresponding Event
$[t_B, .., t_0, nt]$	$[t_B, .., t_0, nt]$	$nt\mu_n(1 - p_t)$	A thread finishes a NTCB and starts another NTCB
$[t_B, .., t_0, nt]$	$[t_B + 1, .., t_0, nt - 1]$	$nt\mu_n p_t$	A thread finishes a NTCB and starts a TCB
$[t_B, .., t_i, .., t_1, 0, nt]$	$[t_B + 1, .., t_i - 1, .., t_1, 0, nt]$	$t_i\mu_t(1 - p_a)p_t$	A thread with $i > 0$ retries left commits a TCB and starts another TCB
$[t_B, .., t_i, .., t_1, 0, nt]$	$[t_B, .., t_i - 1, .., t_1, 0, nt + 1]$	$t_i\mu_t(1 - p_a)(1 - p_t)$	A thread with $i > 0$ retries left commits a TCB and starts a NTCB
$[t_B, .., t_i, .., t_1, 0, nt]$	$[t_B, .., t_i - 1, t_{i-1} + 1, .., t_1, 0, nt]$	$t_i\mu_t p_a$	A thread with $i > 1$ retries left aborts a TCB and restarts
$[t_B, .., t_i, .., t_1, 0, nt]$	$[0, t_B, .., t_{i+1}, .., t_2, t_1, nt]$	$t_1\mu_t p_a$	A thread with 1 retry left aborts a TCB and falls-back to the software path
$[t_B, .., t_1, t_0, t_n]$	$[t_B + 1, .., t_1, t_0 - 1, nt]$	$\mu_f p_t$	A thread completes a TCB in the fall-back path and starts another TCB
$[t_B, .., t_1, t_0, nt]$	$[t_B, .., t_1, t_0 - 1, t_n + 1]$	$\mu_f(1 - p_t)$	A thread completes a TCB in the fall-back path and starts a NTCB

Table 1: State transition diagram

the software fall-back path. In the real system, instead, it is possible for a thread T to start a hardware transaction before another thread T' waiting for the global lock manages to acquire it. This scenario results into the early abort of T as soon as T' acquires the lock.

A restarted transaction is modeled as 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 [27], 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.

4.2 Modeling methodology and target KPIs

Our model is based on average value analysis [35]: 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 average probability that a transaction aborts, noted P_A, the throughput of the system, noted X, and the average response time of a transaction, R. The response time of a transaction differs from the service time in that R includes the fact that a transaction might re-execute multiple times, possibly also in the software fall-back path, before committing.

We model the evolution of the system by means of a Continuous Time Markov Chain (CTMC) [27]. A CTMC is a graph in which vertices represent the states in which the system can be and edges represent the rates at which the system transitions from one state to another. In our case, a state is a tuple $\langle t_B, t_{B-1}, \dots, t_0, nt\rangle$. $t_i,\ i = B, \dots, 1$ indicates the number of threads that are running a TCB and still have i retries of their initial budget left. t_0 is the number of threads that have exhausted their budget and have to execute using the sequential fall-back path. nt 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 $\sum_{i=0}^{B} t_i + nt = \theta$.

The system transitions from a state to another upon the completion of a NTCB, and upon the commit or abort of one or more transactions. When $t_0 = 0$, threads executing hardware transactions can run in parallel. When $t_0 > 0$, threads executing hardware transactions are stalled until the single 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 note μ_t the rate at which a thread completes a transactional code block, either successfully or prematurely because of an abort, and $\mu_n = 1/C_n$ the rate at which a NTCB is completed. We note $\mu_f = 1/C$ the rate at which a thread completes a TCB in the fall-back path. In general, let the system be in a state when there are t hardware transactions running concurrently and nt 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 nt 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 probability that the completion is caused by a commit times the number of concurrent transactions, i.e., $T\mu_t(1 - p_a)$. Following a similar reasoning, the rates at which the second and third events happen are, respectively, $T\mu_t p_a$ and $N\mu_n$. If a transaction aborts and fall-backs to acquiring the global lock, it induces the abort of all the other $t - 1$ transactions and decreases their budget by one.

We describe in more details the transition rates between pairs of states in Table 1. The transitions in the table are specific instantiations of the general reasoning provided above.

Figure **??** in Appendix depicts an example CTMC for the case of two threads and an initial budget B=2.

4.3 Computing the KPIs

Once the CTMC is instantiated with the transition rates, it can be solved by means of standard techniques [27] to obtain the KPIs of interest. The transition rates, however, are not provided as input and thus they have to be computed by the analytical model.

Computing the transition rates is a challenging task because of the recursive definition of the abort probability. In fact, in HTM, a transaction T can abort not only because of conflicting accesses or capacity exceptions. T can also abort because another transaction aborts, exhausts its budget and acquires the single global lock.

We tackle this issue as follows. We first assume transactions can abort only because of conflicts with other transactions. Thus, we obtain the transition rates that allow us to solve the CTMC. Then, we solve the model a second time, using the abort probability computed in the previous iteration to model the effects of transactions falling back to the software path.

4.3.1 Computing the state transition rates

Modeling conflicts. When a transaction T accesses a granule g at time t, it opens a so-called "vulnerability window". Namely, T becomes vulnerable to concurrent accesses to g by other transactions. As discussed in Section 2, we model an eager conflict detection and a "Last requester wins" conflict resolution policy. Hence, if T first reads g, and then another transaction writes g before T commits, then T aborts. Similarly, if T has written g, if another transaction then reads or writes g before T commits, then T aborts.

Transactional threads perform memory accesses every C/L time units on average. Assuming there are θ hardware transactions running, the rate at which they issue granules accesses is $\theta L/C$. Then, the rate at which transactions other than T issue potentially conflicting accesses is $\lambda = (\theta - 1)L/C$. The probability that any of those accesses results in the abort of T because of a conflicting access on g during the vulnerability window of g depends on the probability that at least one of such accesses is directed to g and that this access is incompatible with T's access on g. As we assume that granules are accessed uniformly at random by transactions, the probability that a given granule g is accessed is $1/D$. The probability P_I that another access to g is incompatible with T's access to g corresponds to one minus the probability that both accesses are read accesses, i.e., $P_I = 1-(1-P_W)^2$. Therefore, the rate at which incompatible accesses to g are produced by the other $\theta - 1$ active threads is $P_I\lambda/D$. If T has accessed i granules, then the rate at which concurrent threads generate incompatible memory accesses towards any of these i granules can be approximated as $H(i) = P_I\lambda i/D$.

We now compute the probability that a transaction T successfully acquires i granules, which we note $P_R(i)$. If T has not accessed any memory word, then it cannot be aborted because of conflicting accesses. Hence, $P_R(1) = 1$. By assumption, T will perform its second memory access C/L time units after the first one. Assuming that $H(i)$ is exponentially distributed, we can compute the likelihood that T is aborted at any time t before accessing the second memory word as $H(1)e^{-H(1)t}$. Hence, the probability that T manages to perform its second memory access is

$$P_R(2) = 1 - \int_0^{C/L} H(1)e^{-H(1)t}dt = 1 - e^{-H(1)C/L} \quad (1)$$

The general probability that T successfully manages to acquire its i-th granule can, then, be computed recursively:

$$P_R(i) = P_R(i-1)(1 - e^{-H(i-1)C/L}) \quad (2)$$

We can now compute the mean response time R_t of one execution of a transaction T when running in the hardware path, assuming that transactions can only be aborted because of conflicting accesses. This response time does not include multiple re-executions of the same transaction: it is the average time since the (re)start of T's execution and its completion, independently of whether it is successful or not.

R_t is, thus, the weighted sum of two contributes, one corresponding to the case in which T commits (R_t^C) and one corresponding to the abort case R_t^A. R_t^C is given by the probability that T manages to access all the L granules without aborting, times the cost of executing a TCB, i.e., $P_R(L)C$. In addition to C, a successful execution also takes T_B to execute the begin statement and T_C to commit. During the commit phase, T is still vulnerable to contention from other concurrent transactions. The probability that T survives this last vulnerability window is $e^{-H(L)T_C}$. Summing all these contributes,

$$R_t^C = P_R(L)(T_B + C + e^{-H(L)T_C}T_C) \quad (3)$$

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_t^A 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 final commit phase, because of a conflicting access towards any of the L accessed granules. Hence, using the shorthand $W = C/L$,

$$R_t^A = T_B + \sum_{i=1}^{L-1} P_R(i)\int_0^W iWtH(i-1)e^{-H(i)t}dt +$$
$$+CP_R(L)(1 - e^{-H(L)T_C}) =$$
$$= T_B + \sum_{i=1}^{L-1} P_R(i)\left(iW\left(1 - e^{-H(i)W}\right) + \frac{1}{H(i)} - e^{-H(i)W}\left(W + \frac{1}{H(i)}\right)\right) +$$
$$+CP_R(L)(1 - e^{-H(L)T_C}) \quad (4)$$

We can now compute the per-state p_a and μ_t. The average abort probability in one state is one minus the commit probability:

$$p_a = 1 - P_R(L)e^{-H(L)T_c}$$

μ_t, instead, is computed as the inverse of R_t.

Modeling aborts due to fall-backs. We now use the abort probability and response times that we derived so far in order to capture the dynamics stemming from the aborts of transactions with only 1 retry left. We refer to these as *dangerous* transactions, because their abort causes all other transactions to abort, decrease their budget and wait for the global lock to become free.

Let us consider a state with n non-dangerous transactions and d dangerous ones. We are interested in computing the probability that a transaction T is aborted not only because of a direct conflict, but also because of the conflict experienced by a dangerous transaction.

We model the increase in the abort probability of T by computing an adjusted rate at which T can abort. Such rate, thus, does not encompass anymore only the rate at which other transactions can issue conflicting accesses with T, but also the rate at which dangerous transactions abort because of a conflict they are experiencing. To this end, we assume for simplicity that the set of granules accessed by dangerous transactions is disjoint from the set of granules accessed by T.

Then, we can express the adjusted rate as the previous rate $H(i)$ plus the rate at which dangerous transactions abort. The rate at which a dangerous transaction aborts because of a conflict, is computed as $\mu_t p_a$, which we obtain from the previous analysis. The adjusted rate at which a non-dangerous transaction can abort after having accessed i granules is then $H^n(i) = H(i) + d\mu_t p_a$. 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, $H^d(i) = H(i) + (d-1)\mu_t p_a$.

We can now compute the adjusted value for p_a. To this end, we first compute the adjusted probability that a transaction successfully accesses i granules. We again distinguish between the case of non-dangerous transactions ($P_R^n(i)$) and dangerous ones ($P_R^d(i)$). Both probabilities take the value 1 for $i = 1$. For $i > 1$, following the same reasoning applied when computing $P_R(i)$, we have:

$$P_R^n(i) = P_R^n(i-1)e^{-H^n(i-1)C/L} \quad (5)$$

$$P_R^d(i) = P_R^d(i-1)e^{-H^d(i-1)C/L} \quad (6)$$

Taking also into account the vulnerability window T_c corresponding to the final transaction validation, the average value for the adjusted p_a is:

$$p'_a = 1 - \left(\frac{n}{n+d}P_R^n(L)e^{-H^n(L)T_c} + \frac{d}{n+d}P_R^d(L)e^{-H^d(L)T_c}\right) \quad (7)$$

We also obtain adjusted values for the response time of an execution of a transaction, again depending on whether the transaction is dangerous (R^d) or not (R^n). To compute them, we use Equation 3 and 4, where we substitute $H(i)$ accordingly.

Thus, we compute the average response time of a single hardware execution of a transaction:

$$R'_t = \frac{n}{n+d}R_d + \frac{d}{n+d}R_n \quad (8)$$

The adjusted μ'_t is, hence, its inverse $\mu'_t = 1/R'_t$.

4.3.2 Computing Target KPIs

Once we have the transition rates for every edge of the CTMC, we can obtain average throughput X, average transaction response time R_t^* and average abort probability P_A.

To this end, we first solve the CTMC by means of standard methods [27] to obtain the vector $\vec{\pi}$ of the states probabilities. The i-th entry of this vector, noted $\vec{\pi_i}$, represents the probability of the system being in a given state $s_i \in S$, where S is the set of all the states of the CTMC. We use the notation $s(t_i, ft, nt)$ to indicate the index of the state corresponding to t_i active hardware transactions, ft in the fall-back path and nt non-transactional active threads.

The throughput of the system is defined as the the rate at which any thread in the system completes a code block (NTCB or TCB). It is computed as the weighted average of the system being in a state s_i times the throughput in s_i. On its turn, the throughput in s_i is the sum of the rates corresponding to the completion of a NTCB or the commit of a TCB. We refer to the values of μ'_t computed in state s as $\mu'_{t,s}$.

$$X = \sum_{s(t_i,ft=0,nt)\in S} \vec{\pi}_s(t_i\,\mu'_{t,s}{+}nt\,\mu_n){+}\sum_{s(t_i,ft\geq 1,nt)\in S} \vec{\pi}_s(nt\,\mu_n{+}\mu_f) \tag{9}$$

We note that in a state s in which there is at least one transaction in the fall-back path, this equation captures the fact that there is only one transaction contributing to the throughput, by committing with a rate $\mu_f = \frac{1}{C}$. In a state s in which $ft = 0$, instead, the ti hardware transactions all contribute to the throughput of the system, with a rate $ti\mu_{t,s}$.

To obtain the response time of a transaction, we exploit Little's law [28]. 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 a 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^*.

The average abort probability P_A is computed as the weighted average of the abort probability values obtained in each state s, noted $p'_{a,s}$. Because a transaction cannot abort when running in the fall-back path, we do not consider the states in which $ft > 0$: $P_A = \sum_{s(i,f=0,n)\in S} \vec{\pi}_s p'_{a,s}$.

4.4 Modelling capacity aborts

In this section we discuss how the model presented so far can be extended, in a modular way, with an additional model aimed solely at predicting the probability, noted $P_C(i)$, that a transaction incurs a capacity abort when it issues its i-th operation.

The integration of these two models is indeed straightforward as it suffices to observe that the the probability to reach operation i when both aborts due to conflicts and capacity exceptions are possible, noted $P''_R(i)$, can be expressed as:

$$P''_R(i) = P_R(i)(1 - P_C(i)) \tag{10}$$

5. Validation

This section reports the results of a validation study that compares the KPIs predicted by the model presented in the previous sections with those achieved when executing on our target experimental platform (see Section 3).

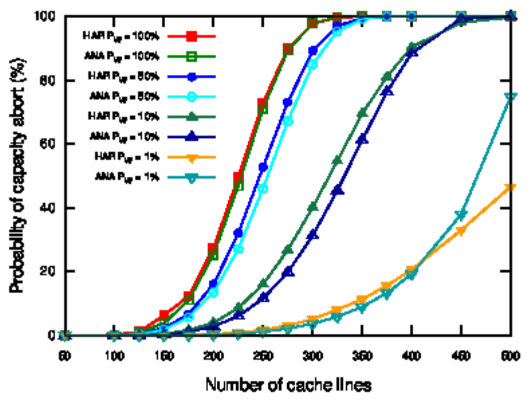


Figure 2: Validating the analytical model (ANA) for the probability of capacity aborts vs a real system (HAR)

We start by validating the accuracy of our model of capacity aborts, since it is a building block on which the overall performance model is built. To this end we run several experiments in which transactions perform N distinct memory accesses with a write probability $0.01 \leq P_W \leq 1$. We then measure the probability that a transaction incurs a capacity aborts before successfully completing the N memory accesses. Such probability is calculated as the ratio between the number of capacity aborts and the total number of started transactions (excluding the ones failed because of spurious aborts).

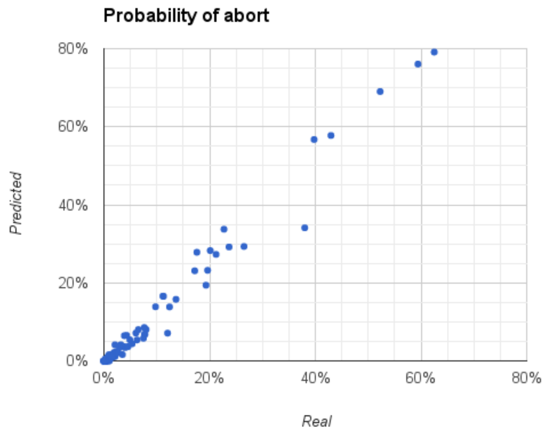
To control as much as possible the transaction footprints, we need to minimize the amount of auxiliary data structures used to generate the random access path. To this end, transactions can only access memory addresses belonging to a large set of D candidates. Each memory location $d \in D$ is pre-initialized with a random address belonging to D. After accessing d, a transaction accesses the granule at the address encoded in d. In this way, we generate a random access path over the D possible addresses using minimal auxiliary memory during the experiment.

Figure 2 reports the results of the experiments and contrasts them with the predictions output by our analytical model of capacity aborts. The plot shows that the model is able to predict well the probability of a capacity abort as a function of the number of (tentatively) accessed granules and write access probability, attaining a MAE of 2.12%. The highest error is incurred by the model for $P_W = 0.01$. This is an effect of the approximation that we have introduced in Section 4.4 to obtain a closed form solution for the capacity abort probability. Such approximation, in fact, works better as P_W tends to 1 and, with $P_W = 0$ would yield to a null probability of incurring a capacity abort, regardless of the number of performed accesses.

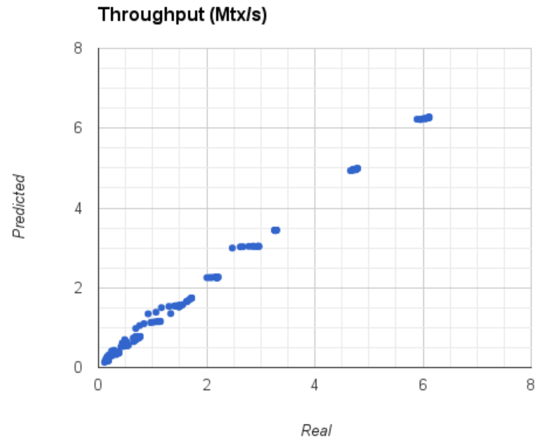
We now evaluate the accuracy of the presented analytical model as a whole. In order to stress its prediction capabilities, we use a synthetic benchmark that generates different contention levels and access patterns. In total, we consider a set of 380 workloads, obtained by varying the workload parameters as follows: $\theta \in \{1,2,3,4\}$, $B \in \{2,4,6\}$, $L \in \{2,5,10,20\}$, $D \in \{512, 2048, 8192, 32768\}$, $P_W \in \{0.5, 1.0\}$.

A micro-benchmark launches θ concurrent threads pinned 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 predefined granule pool of size D.

The memory accesses of a transaction are performed as follows. First, a random random value $0 \leq g < D$ is generated, such that g is different from previously generated accesses. Then, with



(a) Abort probability. MAE = 4.94%, R = 0.9923.



(b) Throughput (10^6 txs. per sec.) MAPE = 8.12%, R = 0.9989.

Figure 3: Validation of the predicted KPIs vs a real system.

probability P_W the transaction writes g; with probability $1 - P_W$ the transaction reads g. The granules fit an entire cache line and are aligned in memory to avoid aliasing conflicts.

The CPU demand of a transaction depends on the number L of accessed granules. For each of the considered values of L, we measure the corresponding CPU demand C, which we provide as input to the model.

In Figure 3 we report a scatterplot comparing the real and predicted probability of abort and throughput. The reported results for the real system are obtained as the average of 10000 executions, from which we removed the first and last quartile to filter out outliers. Presented error metrics are MAPE, MAE and the Pearson correlation factor R. The closer R is to 1, the better is the output prediction of the model.

The reported data confirms the high accuracy of the proposed model in predicting both the throughput and abort probability of the system: the MAE for the abort rate is less than 5% and the MAPE for the throughput around 8%; the Pearson correlation coefficient, R is in both case larger than 0.99, confirming the strong correlation between the predicted and real KPI values.

6. Related Work

The works that are most closely related to our proposal 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 [36, 1, 37, 22, 10]. More recently, several analytical models have been proposed for the concurrency control algorithms adopted by software implementations of TM [38, 24, 9, 33].

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., the speculative execution of parallel transactions) and of a sequential, and hence inherently 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.

Among the aforementioned works, the proposed model shares a common treat with the STM model proposed in Di Sanzo et al. [9], namely the reliance on a CMTC to capture the presence of execution phases where threads execute in different modes. In the case of the work of Di Sanzo et al., though, the CMTC served to distinguish solely between threads executing transactional and non-transactional code blocks. The CMTC defined in our model, instead, captures a broader range of dynamics, i.e., also the number of threads with a given available budget and those executing/enqueued in the fall-back path.

A different line of work makes use of analytical models and formal notation to predict if it is possible to develop Hybrid TM (HyTM) solutions without instrumentation [2].

Black box techniques for throughput prediction are present in the literature for the case of STM [6, 32], and also in HTM either to predict its throughput [34] or to improve its performance by tuning the TM parameters [13, 18]. 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 techniques is known to be strongly influenced by the extent to which the data collected during the training phase is representative of the actual conditions in which they will be employed [3, 15].

Finally, recently, several gray-box modelling techniques have been applied to the case of transactional systems. These approaches combine white and black box modes in order to reduce the learning cost and/or enhance the accuracy of predictions [32, 17, 15, 33, 16, 14]. As already discussed in Section 4.4, given that the internal mechanisms used by the Intel's HTM implementation to maintain the transactional metadata are undisclosed, a natural way to extend the presented model would be to integrate it with a black-box model aimed at predicting the capacity abort probability.

7. Conclusions and future work

This paper presented an analytical model that captures the performance dynamics of the concurrency control algorithm employed in a mainstream hardware transactional memory (HTM) implementation, namely the one provided by Intel's Xeon (Haswell family) processors. The proposed model was validated using a real system and a synthetic benchmark, which allowed to confirm its high accuracy, at least in the set of considered workloads.

The model proposed in this work fills a relevant gap in the literature on performance modelling of TM, as it is, to the best of our knowledge, the first analytical model targeting the concurrency control of a HTM system. Yet, the presented model has also some limitations, which we discuss in the following.

A limitation of the current work is its reliance on the assumption that memory addresses are accessed with uniform probability, whereas many applications' workloads tend to exhibit skewed access patterns. In order to cope with this issue we plan to exploit the idea of modelling workloads that generate non-homogeneous access patterns over a data set of size D using instead a simpler uniform workload over a data set of size $D' \neq D$ that generates an equivalent contention level [36, 17, 14]. The key challenge here lies in identifying the transformation that maps D to D', as this is dependent on the concurrency control employed by the underlying system (and still unknown for the case of Intel's HTM).

Finally, the number of states associated with the continuous time Markov-chain (CTMC) used in the presented model grows exponentially with the number of threads and the budget of attempts available to execute a transaction using HTM. In order to enhance the model's scalability, we are exploring different approximation techniques, which trade-off prediction accuracy to achieve significant reduction in the number of states required by the CTMC.

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] D. Alistarh, J. Kopinsky, P. Kuznetsov, S. Ravi, and N. Shavit. Inherent limitations of hybrid transactional memory. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9363:185–199, 2015.

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

[4] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. *9th Workshop on Transactional Computing (TRANSACT'14)*, 2014.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. *ASPLOS*, 44:157, 2009.

[12] D. Didona, N. Diegues, R. Guerraoui, A.-M. Kermarrec, R. Neves, and P. Romano. ProteusTM: Abstraction Meets Performance in Transactional Memory, apr 2016.

[13] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker. Identifying the Optimal Level of Parallelism in Transactional Memory Applications. In *Networked Systems*, volume 7853, pages 233–247. 2013.

[14] D. Didona and P. Romano. Performance modelling of partially replicated in-memory transactional stores. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '14, pages 265–274, Washington, DC, USA, 2014. IEEE Computer Society.

[15] 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.

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

[17] 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.

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

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

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

[21] 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.

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

[23] 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.

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

[25] 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.

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

[27] L. Kleinrock and S. Lam. Packet Switching in a Multiaccess Broadcast Channel: Performance Evaluation. *IEEE Transactions on Communications*, 23(4):410–423, apr 1975.

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

[29] 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.

[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] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3rd edition, 2002.

[32] 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.

[33] 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.

[34] N. Shavit, A. Nguyen, and W. Hasenplaugh. *Investigation of Hardware Transactional Memory*. PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2015.

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

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

[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.