ANALYTICAL MODELING OF COMMIT-TIME-LOCKING ALGORITHMS FOR SOFTWARE TRANSACTIONAL MEMORIES

Pierangelo Di Sanzo, Bruno Ciciani, Roberto Palmieri, Francesco Quaglia
DIS, Sapienza Universita' di Roma

Paolo Romano
INESC-ID, Lisbona, Portugal

We present an analytical performance modeling approach for concurrency control algorithms in the context of Software Transactional Memories (STMs). Unlike existing approaches, we consider a realistic execution pattern where each thread alternates the execution of transactional and non-transactional code portions. Also, our model captures dynamics related to the execution of both transactional read/write memory accesses and non-transactional operations, even when they occur within transactional contexts. Further, we rely on a detailed approach explicitly capturing key parameters, such as the execution cost of transactional and non-transactional operations, as well as the cost of begin, commit and abort operations. The proposed modeling methodology is general and extensible, lending itself to be easily specialized to capture the behavior of different STM concurrency control algorithms. In this work we specialize it to model the performance of Commit-Time-Locking algorithms, which are currently used by several STM systems.

1 Introduction

Software Transactional Memories (STMs) [1, 2, 3, 4] are emerging as a highly attractive and potentially disruptive programming paradigm. Leveraging on the proven concept of atomic and isolated transactions, STMs spare the programmers from the pitfalls of conventional handcrafted synchronization, thus significantly simplifying the development of concurrent applications. Further, STMs have been recently identified as an ideal candidate to simplify the programming of distributed applications deployed in large scale data centers [5] or in cloud computing environments [6].

Compared to traditional transactional systems, such as database systems, STMs are based on (and require) innovative design/development approaches, where the optimization focus is shifted on aspects that historically had less importance. Among them we can mention hardware-cache aware design (see, e.g, [7]) as well as tailoring of the design to the specific instruction set offered by the target computing architecture (see, e.g., [8]). At the same time, concurrency control schemes commonly adopted in database environments are not likely to fit all the requirements of fine grained, volatile memory operations typical of STM-based applications [4].

According to the above considerations, the wide set of database oriented performance analysis results (see, e.g., [11, 12, 13, 14, 15, 16]) cannot be (fully) representative of the actual performance levels provided by STM systems. Hence, a major issue to address when dealing with innovative concurrency control algorithms specifically tailored to STM environments (see, e.g., [3, 9, 10]) is the lack of analytical models able to reliably capture their actual dynamics. Such models would be helpful since they could provide indications on aspects of interest for both designers of STM layers and developers of STM-based applications. As an example, it would be extremely important to assess how well a given STM concurrency control algorithm scales vs the degree of parallelism, namely the number of CPU-cores available within the underlying computing platform.

In this paper we address such a lack by providing a two-layered analytical modeling methodology well suited for STM systems. In our approach, a top modeling-layer predicts the system performance as a function of the degree of concurrency within the system (e.g. the number of worker threads in charge of executing transactional memory operations, and the probability that they are executing transactional vs non-transactional code portions), independently of the specific scheme adopted for regulating memory accesses by concurrent conflicting transactions. The latter aspect is instead demanded to the bottom layer of the model, which can be specialized in a way to determine commit/abort probabilities on the basis of the specific choices determining the actual synchronization (concurrency control) scheme among threads executing conflicting transactional code portions.

In this work we provide an instantiation of the bottom layer which models the behavior of the Commit-Time-Locking (CTL) concurrency control algorithm, which is adopted by several popular STM systems, such as TL2 [9]. The performance model has been also validated against simulation results obtained considering data access patterns based on the well known STAMP benchmark [17].

The remainder of this paper is structured as follows. In Section 2 we discuss literature results related to our contribution. The analytical modeling methodology, together with the specific model instantiation for CTL is provided in Section 3. The comparison between model and simulation results is provided in Section 4. Section 5 concludes the paper.

2 Related Work

The work in [18] provides an analytical model for STM systems. However, the provided modeling approach suffers from two key limitations, which are overcome by the approach we present in this paper. First, the model in [18] assumes that applications are constantly executing transactions, while real STM-based applications rely on threads that alternate the execution of transactional and non-transactional code, with data conflicts being possible exclusively during the execution of transactional code portions. Second, the model in [18] abstracts over time by describing the execution of a transaction as a sequence of steps whose duration is left unspecified. This restricts the usage of the model exclusively to qualitative comparisons among different STM algorithms, making it infeasible for forecasting fundamental time-related performance metrics, such as response time or throughput (unless when assuming that all the phases of the execution of any transaction have identical, constant duration). Differently, our analytical modeling approach is able to capture the advancement of time according to a continuous timeline, which is achieved via a Continuous Time Markov Chain. Also, it relies on a detailed workload characterization model, which includes key cost parameters related to both STM internals and STM-based applications, such as the duration of transactional operations (read/write accesses to transactional memory locations, as well as begin/commit/abort operations), and explicitly accounts for the time-interval in between two transactional operations.

Leveraging on the common notion of atomic transactions, STM algorithms and DBMS concurrency control schemes are naturally closely related. The analytical modeling of concurrency control in database environments has been widely investigated over the last three decades. Analytical modeling approaches have been presented in, e.g. [13, 14, 15, 19, 20, 21] for the case of centralized database systems, and in [16, 22] for the case of distributed/replicated databases. However, database transactions encompass accesses to stable storage and incur in the overhead of SQL parsing and plan optimization. Conversely, in STM environments, transactions are used as a programming-language construct for the manipulation of in-memory data structures. This makes their execution time several orders of magnitudes smaller than the counterpart in DBMS scenarios [29], which amplifies the impact of the overhead associated with the STM-specific internal schemes for the management of low-level data-structures (e.g. CTL [9]). These schemes do not have a direct counterpart in the database literature so, consequently, they are not covered by the literature on analytical modeling of concurrency control schemes for database systems. Also, existing performance models of concurrency control schemes do not capture the behavior of applications alternating the execution of transactional and non-transactional phases, as it is conversely typical of STM-based applications.

3 The Analytical Model

3.1 Assumptions and Considerations

As typical of STM applications/benchmarks [17, 23, 24] we assume a fixed number k of threads. Also, we assume that each thread executes on a distinct CPU-core. Each thread alternates the execution of transactional and non-transactional code blocks. A non-transactional code block is formed by a sequence of machine instructions which we denote as $ntcb$. Each transaction starts with a $begin$ operation, then executes n transactional operations (namely, either $read$ or $write$ operations) and finally ends by issuing a $commit$ operation. After the $begin$ operation and after each transactional operation, the thread executes a code block, denoted as tcb, during which it does not perform transactional read/write operations.

We denote with t_{begin}, t_{read}, t_{write} and t_{commit}, the expected time required by a thread to execute, respectively, $begin$, $read$, $write$ and $commit$ operations. Note that, in practice, these durations are affected by both the speed of the underlying hardware platform and the internals of the underlying STM layer. Compared to existing approaches (see, e.g., [18]), the choice of capturing the above costs through ad-hoc parameters enhances the flexibility of our model, thus allowing it to be employed for what-if analysis aimed at forecasting the performance for diverse scenarios and/or workloads. As an example, the model can be used to assess the performance of STM-based applications when deployed on different hardware platforms (which might give rise to different machine instruction patterns) or when changing the internals of the underlying STM layer (e.g. via the exploration of trade-offs between alternative implementation strategies).

We assume the duration of tcb and $ntcb$ to be exponentially distributed, with mean t_{tcb} and t_{ntcb}, respectively. Whenever a transaction is aborted, an $abort$ operation is executed, whose handling has an expected duration t_{abort}. After experiencing an abort, a transaction is temporarily held in a back-off state for an exponentially distributed time period, denoted as $t_{backoff}$, at the end of which it gets restarted. The probability for a transactional operation to be a memory write is denoted as p_{write}, otherwise it is a memory $read$ operation. To simplify the model, we will assume that the n read/write operations occurring within a transaction access distinct data items, uniformly distributed across the whole set of d data items maintained within the memory layout. In other words, we assume that two distinct read/write operations never access the same data item.

3.2 Modeling Approach Overview

As hinted, we logically structure our model in two distinct parts, each one capturing complementary aspects of the execution dynamics of STM-based applications. The first part of the model, which we name top modeling-layer, is presented in Section 3.3. It exploits a Continuous Time Markov Chain (CTMC) to determine how the various threads in the system alternate among the following three phases: (i) execution of a non-transactional code block, (ii) execution of an STM transaction, (iii) blocked in back-off.

By allowing the determination of the probability distribution of the number of threads in each of these three phases, this layer of the model can be used to output standard performance metrics such as throughput and execution time. This part of the model is de-facto oblivious of the specific algorithm used by the STM to regulate concurrency, over which it abstracts via two key input parameters: (a) the average transaction execution time (independently of its final outcome) and (b) the commit probability, given a number $i \in [1, k]$ of threads concurrently executing transactions. Instead, these parameters are computed by what we refer to as bottom modeling-layer, one instance of which, tailored to CTL, is presented in Section 3.4. This layer is focused on capturing proper dynamics associated with the specific conflict detection and resolution schemes adopted by the STM layer, assuming a constant, albeit parametric, number of threads simultaneously executing transactions.

By decoupling the modeling of the dynamics associated with thread alternation among the various phases from the modeling of the concurrency control algorithm, our two-layered modeling methodology provides the below reported benefits:

1. It simplifies the modeling stage of the concurrency control algorithm, delegated to the bottom modeling-layer, since this model does not require to explicitly consider dynamic variations of the number of threads concurrently executing transactional code blocks. The model only requires to provide performance predictions under the assumption that exactly i threads are concurrently executing transactions. Then, it will be the responsibility of the top modeling-layer to exploit the independent performance forecasts associated with different values of i in order to generate the final performance predictions.

2. It allows seamless replacement of the model of the STM concurrency control scheme presented in this paper, namely the CTL model [9] (see Section 3.4), with alternative ones either relying on different modeling approaches and/or targeting different concurrency control algorithms.

3.3 Top Modeling-Layer: Thread Execution Model

We model the alternation of the various phases for the execution of the different threads (inside a transaction, executing a non-transactional code block or blocked in back-off after an abort) via a Continuous Time Markov Chain (CTMC) [25]. Each state of the CTMC is marked and identified by a couple of integers (i, j) representing, respectively, the number of threads running transactions and the number of threads in back-off. Since the total number of threads in the system is equal to k, the only admissible states in the CTMC are those for which the corresponding (i, j) pair respects the constraints: $i, j \in [0, k]$ and $i + j \leq k$.

For each state $(i, \cdot)$, with $i > 0$, the model takes as input parameters the transaction execution rate, denoted as μ_i, and the probability $p_{c,i}$ for a transaction to successfully commit, in case of i threads simultaneously executing transactions. These need to be provided by the bottom modeling-layer in charge of capturing the effects of the specific concurrency control scheme. In the following we will denote with $p_{a,i} = 1 - p_{c,i}$ the probability for a transaction to experience an abort, when considering that i threads are concurrently executing transactions. Also, we will denote with $\lambda = \frac{1}{t_{ntcb}}$, the rate according to which a thread executes a non-transactional code block (in between two transactions).

We can now list the rules defining the transition rates from any two states of the CTMC:

- for $i + j < k$, the transition rate from state (i, j) to state $(i + 1, j)$, associated with the activation of a transaction after the completion of the execution of a non-transactional code block, is equal to $\frac{k-i-j}{t_{tcb}}$;
- for $i > 0$, the transition rate from state (i, j) to state $(i - 1, j)$, associated with transaction commit events and the subsequent activation of a non-transactional code block, is equal to $\mu_i \cdot p_{c,i} \cdot i$;
- for $i > 0$, the transition rate from state (i, j) to state $(i-1, j+1)$, associated with transaction abort events and the start of the back-off period, is equal to $\mu_i \cdot p_{a,i} \cdot i$;
- for $j > 0$, the transition rate from state (i, j) to state $(i + 1, j - 1)$, associated with the termination of back-off periods and transaction restart, is equal to $\frac{1}{t_{backoff}} \cdot j$.

We exclude state $(0, k)$ as a possible one since, (i) the CTMC characterizing our model does not express state transitions where multiple transactions get simultaneously aborted due to (mutual) conflicts, and (ii) adopting whichever literature STM concurrency control algorithm, if a single thread is currently executing a transactional code block, then the corresponding transaction cannot be aborted. It is easy to show that the set of states of the CTMC, denoted as S, has cardinality equal to $\frac{(k+1)\cdot(k+2)}{2} - 1$. Note also that, if $i + j < k$, it follows that $k - (i + j)$ threads are executing non-transactional code blocks.

As typically expected in any real system, if we assume $\mu_i > 0$, $p_{c,i} \neq 0$ and $p_{c,i} \neq 1$ (the cases of $p_{c,i} = 0$ or $p_{c,i} = 1$ express, respectively, a pathological scenario with no possibility of transaction progress and a trivial scenario entailing no data contention), given that $i \in [1, k]$, the CTMC is irreducible, and is formed by an ergodic set of states. Thus the stationary probability vector v is unique and satisfies the typical equation

$$\mathbf{v} \cdot Q = \mathbf{0} \tag{1}$$

where Q is the infinitesimal generator matrix of the CTMC [26]. Assuming that the system is in steady-state, and that we are provided with μ_i and $p_{c,i}$ values ($\forall i \in [1, k]$), we can compute the probability to be in each state $(i, j) \in S$ by resolving equation (1). We can then evaluate the system throughput τ as the sum of the transaction commit rates in the different states, weighted according to the probability for the system to be in each state (i, j)

$$\tau = \sum_{(i,j)\in S} v_{i,j} \cdot i \cdot \mu_i \cdot p_{c,i} \tag{2}$$

The overall transaction commit and abort probabilities, denoted as p_c and p_a, can be accordingly evaluated, using the below expressions

$$p_c = \sum_{(i,j)\in S} v_{i,j} \cdot p_{c,i} \tag{3}$$

$$p_a = (1 - p_c) \tag{4}$$

3.4 Bottom Modeling-Layer: CTL Model

In this section we introduce an analytical model of Commit-Time-Locking (CTL) concurrency control, considering the version implemented within the TL2 STM layer [9]. This version is considered as one of the best performing concurrency control algorithms for typical STM workloads. We will start by over-viewing such a target version of the CTL algorithm, and then we will move to the presentation of its analytical model.

3.4.1 Algorithm Overview

Unlike, e.g., strict 2PL [27], CTL does not acquire locks upon accessing data items. Instead, lock acquisition is delayed to commit time, and only involves written data items (write-locks). This choice enhances concurrency with respect to conventional lock-based schemes by, e.g., avoiding to block transactions issuing a write operation on a data item that has already been read/written by a concurrent transaction.

Given the absence of read-locks, consistency is ensured via a validation mechanism used to notify transaction T, which speculatively read a data item x, about the fact that x was overwritten by some concurrent transaction T' preceding T in the commit order. To this end, a versioning scheme is employed which associates a timestamp value with each data item, referred to as Write-Version-Clock (WVC). The generation of WVC values relies on a unique Global-Version-Clock (GVC), which is read by any transaction upon startup, and is atomically increased upon transaction commit. The updated value is used as the new WVC value for all the data items written by the committing transaction.

When validating a transaction against a read data item x, two actions are performed:

1) it is checked whether there is a write-lock being held on x (which implies that another transaction has written x and is currently within its commit phase);

2) it is checked whether the current timestamp associated with x is grater than the timestamp read by the transaction upon starting up (which indicates that some concurrent transaction has overwritten x and has already been committed).

If one of the previous checks fails, the transaction gets aborted. This validation scheme is used upon each read operation and, as we shall discuss below, also at commit time. Accordingly, CTL concurrency control schemes guarantee the *opacity* property [28], which ensures that the snapshot observed by any transaction is equivalent to the one that would have been observed according to some serial execution history. As discussed in [28], this property is crucial since for several categories of STM-based applications, transactions observing an inconsistent snapshot may be trapped within infinite loops, or may even cause the application program to crash (e.g. due to an invalid memory reference).

As far as write operations are concerned, in CTL they are buffered within a private workspace until the commit phase. When a transaction attempts to commit, it first acquires the write-locks for all the data items within its write-set. If any of these lock acquisitions fails (due to lock holding by some other transaction), the transaction is aborted. Otherwise, the transaction increments the GVC and tries to validate all the data items it has within its read-set (according to the aforementioned validation procedure). If the validation fails for at least one item within the read set, the transaction gets aborted. If no abort occurs, the data within the write-set are copied back to their original memory locations, together with the updated values for their WVCs, reflecting the updated value of the GVC. All the acquired locks are released at the end of the commit phase, or upon the abort.

3.4.2 Analytical Model

As previously discussed, the bottom modeling-layer computes the transaction execution rate $\mu_i = 1/r_{t,i}$ (where $r_{t,i}$ is the average transaction execution time) and the transaction commit probability $p_{c,i}$ under the assumption that there are i threads simultaneously processing transactions, with $1 \leq i \leq k$. If $i = 1$, a single thread is currently executing transactional code, thus no data conflict can arise. This also means that the currently executed transaction can not be aborted and it follows that $p_{c,1} = 1$. Therefore, for the average transaction execution time we have that

$$r_{t,1} = t_{begin} + n \cdot t_{op} + (n+1)t_{tcb} + t_{commit} \tag{5}$$

where t_{op}, namely the average time to execute an operation, is equal to

$$t_{op} = t_{read}(1 - p_{write}) + t_{write} \cdot p_{write} \tag{6}$$

For $i \neq 1$ we proceed as follows. Once fixed i, we use a procedure that iteratively recalculates the values of $p_{c,i}$ and $r_{t,i}$. Upon starting the iterative procedure, the initial values can be selected as $p_{c,i} = p_{c,i-1}$ and $r_{t,i} = r_{t,i-1}$ for commodity. The output values by an iteration step are used as the input values for the next step. We conclude the iterative procedure as soon as the corresponding input and output values for $p_{c,i}$ and $r_{c,i}$ differ by at most an ϵ. In all the configurations that we have experimented, using ϵ=1%, the procedure has always converged in at most fifteen iterations.

In each iteration step the following set of parameters, captured by our model, are re-evaluated:

- $p^o_{a,l}$, namely the probability for a transaction to abort while executing its l^{th} operation due to validation fail (recall that a transaction can abort while executing an operation only if the operation is a read);
- p_{alc}, namely the probability for a transaction to abort at commit time due to lock contention experienced in the commit-time lock acquisition phase;
- p_{avf}, namely the probability for a transaction to abort at commit time due to validation failure of its read-set.

In order to model these parameters, we consider that the expected system state *seen* by any of the i active transactions is determined by the activities associated with the other $i - 1$ transactions currently within the system. Thus we use the following approach.

When a transaction successfully commits, an average number $n \cdot p_{write}$ of locks are first acquired, and then released after read-set validation and write-back phases. Actually, the duration of the lock acquisition and release phases are typically negligible with respect to the duration of validation and write-back phases (recall that, during lock acquisition, transactions do never block, even if they experience contention). Hence, for simplicity, we assume lock acquisition and release to be instantaneous and to occur, respectively, at the beginning and at the end of the commit phase. Also, if a transaction is aborted, no real rollback operation is required for undoing the effects of the corresponding write operations since transaction write-sets are reflected to memory only if transactions successful commit. Thus, to simplify, we ignore the cost of aborts when we evaluate the average lock holding time, assuming that if a transaction successfully completes the lock acquisition phase, it holds the locks for an average time equal to t_{commit}.

From now on we make the assumption that transactions arrive to the commit phase according to a Poisson process. Let us now compute the probability for a transaction to abort while executing a read operation on a data item x, given that it finds the corresponding lock currently busy. For this case to be possible, there must exist another transaction that has written x, that is currently in its commit phase and that has successfully acquired the locks for all the data items in its write-set. Given that we are assuming uniformly distributed accesses to distinct data items in each of the n operations issued within a transaction, it follows that the probability for a committing transaction to have a specific data item within its write-set is $n \cdot p_{write}/d$. Exploiting the aforementioned assumption of Poissonianity of the arrival rate of transactions to the commit phase, we can compute the probability to incur in lock contention while issuing a read operation as:

$$p_{lock} = l_r \cdot t_{commit} \cdot \frac{n \cdot p_{write}}{d} \tag{7}$$

where l_r is the rate according to which the remaining $i-1$ transactions in the system successfully execute the lock acquisition phase. This rate can be evaluated as follow

$$l_r = \frac{1}{r_{t,i}} \cdot (p_{c,i} + p_{avf}) \cdot (i-1) \tag{8}$$

where p_{avf} is the probability for a transaction to abort during the read-set validation phase. Note that $p_{c,i} + p_{avf}$ is the probability for a transaction to successfully execute the lock acquisition phase. We will evaluate p_{avf} later in this subsection.

Now we determine the probability $p^o_{a,l}$ for a transaction T to abort while executing the l-th operation due to the fact that the corresponding data item is accessed in read mode and was updated since T started its execution (that is, some other transaction has written the data item and has successfully committed after T started). The rate u_r at which a data item is updated by transactions is equal to

$$u_r = c_r \cdot (n \cdot p_{write})/d \tag{9}$$

where c_r expresses the rate at which the other $i-1$ transactions successfully commit, and can be evaluated as

$$c_r = \frac{1}{r_{t,i}} \cdot p_{c,i} \cdot (i-1) \tag{10}$$

Equation (9) is obtained by relying again on the assumption that each operation of a transaction accesses a distinct data item and that each of the d data items within the transactional-memory system is accessed with the same probability. If the l-th operation by transaction T is a read operation, the average time $t_{b,l}$ elapsed since T started its execution can be expressed as $t_{begin} + t_{tcb} \cdot l + t_{op} \cdot (l-1)$. As we are assuming that the arrival of transactions to the commit phase forms a Poisson process, the probability $p^o_{u,l}$ for a read (executed as the l-th operation of T) to access a data item that has been updated by some successfully committing transaction after T started can be expressed as

$$p^o_{u,l} = (1 - e^{-u_r \cdot t_{b,l}}) \tag{11}$$

Hence, the probability for a transaction to abort during the execution of its first operation (i.e., when l=1), namely $p^o_{a,1}$ can be evaluated as

$$p^o_{a,1} = (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^o_{u,1}) \tag{12}$$

Since the abort of a transaction T during its l-th operation (where $2 \leq l \leq n$) implies that T did not abort during its previous $l-1$ operations, it follows that

$$p^o_{a,l} = p^o_{na,l} \cdot (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^o_{u,l}) \tag{13}$$

where $p^o_{na,l}$ is the probability of not aborting until the completion of the $(l-1)^{th}$ operation. For this last probability we have

$$p^o_{na,1} = 1 \tag{14}$$

and

$$p^o_{na,l} = (1 - p^o_{a,l}) \cdot p^o_{na,l-1} \tag{15}$$

In equations (12)-(13) we have assumed that the aborts due to lock conflict and the aborts due to validation failures are independent events.

The probability p_{alc} for a transaction T to abort at commit time due to lock contention can be evaluated as follow. T can experience contention while requesting the lock on a data item x only if, at the time in which T starts its commit phase, some other transaction that has written x has successfully completed its lock acquisition phase, and is still executing the commit procedure. Considering that T aborts only if *at least one* of the data items in its write-set is locked, then we approximate the abort probability as

$$p_{wlc} = c_r \cdot t_{commit} \cdot (1 - (1 - \frac{n \cdot p_{write}}{d})^{n \cdot p_{write}}) \tag{16}$$

where the term $1 - (\frac{n \cdot p_{write}}{d})$ in the above equation represents the probability for a data item not to be locked by a committing transaction that successfully completes its lock acquisition phase. Thus we have

$$p_{alc} = p^{o}_{na,n+1} \cdot p_{wlc} \tag{17}$$

where we recall that $p^{o}_{na,n}$ is the probability for a transaction not to be aborted until the completion of its n^{th} operation, that is until it enters its commit phase. Consequently, the probability p^{la}_{na} for a transaction not to be aborted during its execution and to succeed in its commit-time lock acquisition phase is equal to

$$p^{la}_{na} = p^{o}_{na,n} \cdot (1 - p_{wlc}) \tag{18}$$

Let us now show how we can evaluate p_{avf}, namely the probability for a transaction T to abort at commit time due to validation failure for its read-set. The validation fails if at least a data item x belonging to the read-set of T is locked by another transaction, or if a new version of x has been committed after the validation executed by T during its read operation on x. We denote with $p^{r}_{u,l}$ the probability that the l^{th} data item of T's read-set has been updated after the original validation (occurred upon the corresponding read operation). We calculate this probability as follows

$$p^{r}_{u,l} = (1 - e^{-u_r \cdot t_{v,l}}) \tag{19}$$

where $t_{v,l}$ is the elapsed time since the original validation, that is

$$t_{v,l} = (t_{tcb} + t_{op}) \cdot (n - l + 1) + t_{commit} \tag{20}$$

Analogously to what we did in equation (13), we evaluate the abort probability due to failure in the validation of the l^{th} data item within the read-set of T as follows

$$p^{r}_{a,l} = p^{r}_{na,l} \cdot (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^{r}_{u,l}) \tag{21}$$

where $p^{r}_{na,1} = 1$ and, for $l > 1$, $p^{r}_{na,l} = (1 - p^{r}_{a,l}) \cdot p^{r}_{na,l-1}$. Then, we can express p_{avf} as

$$p_{avf} = \sum_{l=1}^{n} p^{r}_{a,l} \tag{22}$$

At the end, successful commit probability for the case of i active threads can be evaluated as

$$p_{c,i} = p^{r}_{na,n+1} \tag{23}$$

The average execution time of a transaction $r_{t,i}$ can now be computed as the sum of the average time for a transaction to reach a different execution phase, weighted with the probability for the transaction to abort exactly in that phase. Let us denote with

- $t_{a,l}$ the average duration of a transaction that aborts during its l-th operation, that is:

$$t_{a,l} = t_{begin} + l \cdot (t_{tcb} + t_{op}) + t_{abort} \tag{24}$$

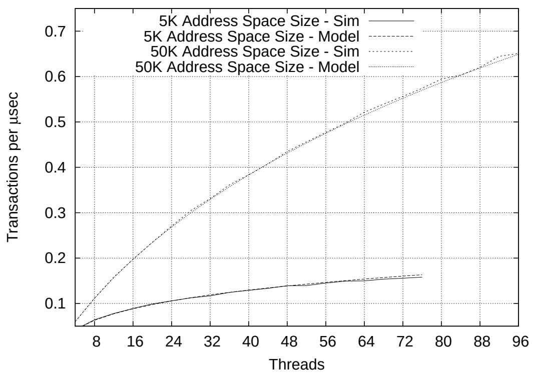

Figure 1: Throughput vs the address space size d.

- $t_1 = t_b + t_{tcb} + t_{abort}$ the average duration of a transaction that aborts during its commit phase due to contention while acquiring locks for the data items in its write-set, where

$$t_b = t_{begin} + n \cdot (t_{tcb} + t_{op}) \tag{25}$$

- $t_2 = t_b + t_{tcb} + t_{commit} + t_{abort}$ the average duration of a transaction that aborts during its commit phase due to failure in validating its readset;

- $t_3 = t_b + t_{commit}$ the average duration of a transaction that successfully commits.

Overall, the average transaction execution time can be expressed as

$$r_{t,i} = \sum_{l=1}^{n} (p^o_{a,l} \cdot t_{a,l}) + p_{alc} \cdot t_1 + p_{avf} \cdot t_2 + p_c \cdot t_3 \tag{26}$$

4 Validation

In this section we provide the results of an evaluation study aimed to verify the accuracy of the proposed modeling methodology, and of the presented CTL model. The study is based on the comparison between some key performance parameters determined via our analytical model and the corresponding values as obtained by means of a discrete event simulator. The latter relies on a detailed, high fidelity simulation model of the internals of an STM layer, which has been developed by also taking into account the internals of TL2. The simulation model mimics the execution of a closed system entailing k concurrent worker threads, whose conflicts when executing transactional code portions are regulated by CTL.

The workload parameters for this study have been selected on the basis of measurement and tracing activities, carried out for the STAMP benchmark, and in particular for the Intruder application specified by the benchmark. To this end, we have exploited an implementation of TL2 which we have instrumented to trace the data access pattern and the costs associated with the corresponding operations, as well as the internal operations performed by the STM layer. Measurements have been carried out using a quad-core 2.4 GHz machine equipped with 4 GB of RAM and running the Suse Linux operating system (kernel 2.6.17).

By the tracing process for Intruder, we gathered the following values: $t_{begin} = 0.2\mu sec$, $t_{read} = 0.25\mu sec$, $t_{write} = 0.2\mu sec$, $t_{commit} = 2\mu$ sec, $t_{abort} = 1\mu sec$. Also, according to the traces, the transactional workload has been modeled by fixing the number n of operations per transaction to the value 70, with $p_{write} = 0.3$. The other adopted settings are: $t_{backoff} = 30$,$t_{tcb} = 0.5\mu sec$, $t_{ntcb} = 10\mu sec$.

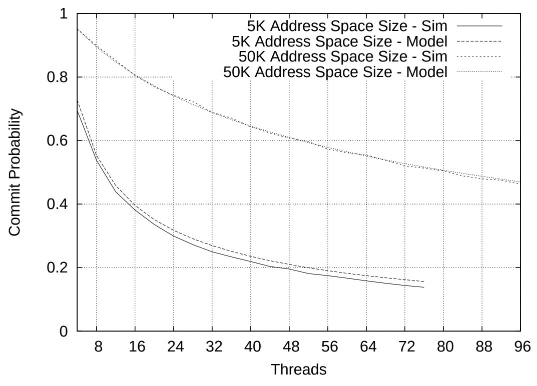

Figure 2: Commit probability vs the address space size d.

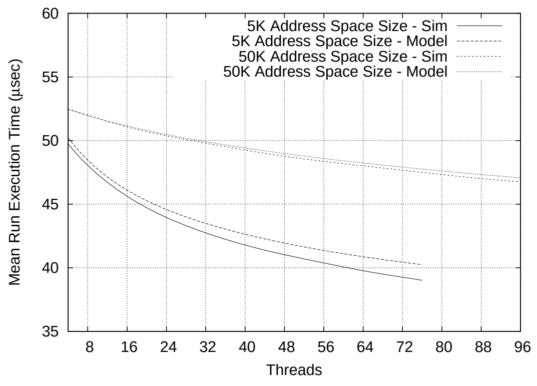

Figure 3: Mean run execution time vs the address space size d.

The independent parameters in the study are the total number of data items d (namely, the transactional memory address space) and the number of worker threads k. Actually, once fixed the number of worker threads, variation of d allows capturing settings with differentiated levels of contention which, in their turn, determine different probabilities for a transaction to be aborted during a run. In this validation we chose two different values for the parameter d. The first one represents a memory layout composed by 5K data items (corresponding to the lower value of d within the benchmark specification). Instead, the second one represents a memory with 50K data items (corresponding to the highest benchmark specified value). Clearly, higher levels of data contention are achieved when the memory is configured with lower values of d, since transactional memory accesses by the worker threads are distributed on a relatively reduced number of memory objects (i.e. a relatively reduced address space). On the other hand, the parameter k has been selected to span form 4 to 76, in case of restricted address space size, and from 4 to 96 for the greater value of d. The comparison between analytical and simulation results is based on the following three parameters: (A) system throughput (Figure 1), (B) commit probability (Figure 2) and (C) mean execution time evaluated over each single transaction run, independently of whether the run is committed or aborted (Figure 3). The plots point out the accuracy of the presented analytical model, highlighting how analytical and simulation results coincide across the whole considered region of the parameters' space, namely low vs high number of worker threads, as well as large vs small address space.

By Figure 2, in correspondence with the lower value of d, we can appreciate the accuracy of the analytical model even in high contention scenarios (namely, for very reduced values of the transaction commit probability). By Figure 3, we remark how, for reduced size of the address space, the relatively high contention probability often leads transactions to be early aborted (they are aborted as soon as the first conflicting memory reference is issued), thus contributing to a reduction of the mean value for the run execution time. (Recall that the mean run execution time is evaluated over both committed and aborted run instances.) Hence, we observe an increase of the mean run execution time in the configuration with larger address space, where the weight of aborted run instances becomes lower. We also note that, by the aforementioned early abort phenomenon, high contention likely generates the case of increased variance for the mean run execution time. The above phenomenon, and their effects on the observed mean value, are correctly captured by our analytical model with very limited error, which is an additional support of the high fidelity of our analytical approach.

5 Conclusions

In this paper we have addressed the issue of analytical modeling of concurrency control schemes in Software Transactional Memories (STMs). Compared to their counterpart in the context of database systems, concurrency regulation approaches for STMs are different in nature, given that the focus is on optimizing design/implementation aspects that have been traditionally treated as less relevant for databases. The provided modeling methodology is general, and can be used to capture differentiated mechanisms within the concurrency regulation layer. We have also specialized our approach to the case of Commit-Time-Locking (CTL) concurrency control algorithms, and we have evaluated the accuracy of the presented CTL performance model against simulation results based on execution patterns resembling the behavior of the STAMP STM benchmark.

References

[1] *A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha, "Unlocking concurrency,"* ACM Queue*, vol. 4, no. 10, pp. 24–33, 2007.*

[2] *N. Shavit and D. Touitou, "Software transactional memory," in* Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing*. Ottawa: ACM Press, 1995.*

[3] *J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions,"* Sci. Comput. Program.*, vol. 63, no. 2, pp. 172–185, 2006.*

[4] *P. Felber, C. Fetzer, R. Guerraoui, and T. Harris, "Transactions are back—but are they the same?"* SIGACT News*, vol. 39, no. 1, pp. 48–58, 2008.*

[5] *M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in* SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA: ACM, 2007, pp. 159–174.*

[6] *P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo, "Cloud-tm: Harnessing the cloud with distributed transactional memories," in* Proceedings of the 3rd ACM SIGOPS International Workshop on Large-Sacle Distributed Systems and Middleware (LADIS)*, Big Sky Resort, Big Sky (MT), USA, Oct. 2009.*

[7] *P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in* PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 237–246.*

[8] *P. Wu, M. M. Michael, C. von Praun, T. Nakaike, R. Bordawekar, H. W. Cain, C. Cascaval, S. Chatterjee, S. Chiras, R. Hou, M. Mergen, X. Shen, M. F. Spear, H. Y. Wang, and K. Wang, "Compiler and runtime techniques for software transactional memory optimization,"* Concurr. Comput. : Pract. Exper.*, vol. 21, no. 1, pp. 7–23, 2009.*

[9] *D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in* Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, 2006, pp. 194–208.*

[10] *M. Herlihy, V. Luchangco, and M. Moir, "A flexible framework for implementing software transactional memory,"* SIGPLAN Not.*, vol. 41, no. 10, pp. 253–262, 2006.*

[11] *P. S. Yu, D. M. Dias, and S. S. Lavenberg, "On the analytical modeling of database concurrency control,"* Journal of the ACM (JACM)*, vol. 40, no. 4, pp. 831–872, September 1993.*

[12] *S. T. Leutenegger and D. Dias, "A modeling study of the tpc-c benchmark,"* SIGMOD Rec.*, vol. 22, no. 2, pp. 22–31, 1993.*

[13] *P. di Sanzo, B. Ciciani, F. Quaglia, and P. Romano, "A performance model of multi-version concurrency control," in* Proceedings of the 16th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008)*. IEEE Computer Society, 2008, pp. 41–50.*

[14] *A. Thomasian and I. Ryu, "Performance analysis of two-phase locking,"* IEEE Transactions on Software Engineering*, vol. Volume 17, no. Issue 5, pp. 386 – 402, May 1991.*

[15] *I.K.Ryu and A.Thomasian, "Analysis of database performance with dynamic locking,"* Journal of the ACM (JACM)*, vol. Volume 37, no. Issue 3, pp. pp. 491 – 523, July 1990.*

[16] *B.Ciciani, D.M.Dias, and P.S.Yu, "Analysis of concurrency-coherency control protocols for distributed transaction processing systems with regional locality,"* IEEE Transactions on Software Engineering*, vol. Volume 18, no. 10, pp. pp. 899–914, October 1992.*

[17] *C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in* IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.*

[18] *A. Heindl and G. Pokam, "An analytic framework for performance modeling of software transactional memory,"* Comput. Netw.*, vol. 53, no. 8, pp. 1202–1214, 2009.*

[19] *A. Thomasian, "Concurrency control: Methods, performance, and analysis,"* ACM Computing Surveys*, vol. 30, no. 1, March 1998.*

[20] *P. Di Sanzo, R. Palmieri, B. Ciciani, F. Quaglia, and P. Romano, "Analytical modeling of lock-based concurrency control with arbitrary transaction data access patterns," in* WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. New York, NY, USA: ACM, 2010, pp. 69–78.*

[21] *Y. C. Tay, R. Suri, and N. Goodman, "A mean value performance model for locking in databases: the no-waiting case,"* J. ACM*, vol. 32, no. 3, pp. 618–651, 1985.*

[22] *B.Ciciani, D.M.Dias, and P.S.Yu, "Dynamic and static load sharing in hybrid distributed-centralized systems,"* Computer Systems Science and Engineering*, vol. Volume 7, no. 1, pp. pp. 25–41, January 1992.*

[23] *R. Guerraoui, M. Kapalka, and J. Vitek, "STMBench7: a benchmark for software transactional memory,"* SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 315–324, 2007.*

[24] *M. Ansari, C. Kotselidis, I. Watson, C. C. Kirkham, M. Lujï£¡n, and K. Jarvis, "Lee-tm: A non-trivial benchmark suite for transactional memory." in* ICA3PP*, ser. Lecture Notes in Computer Science, A. G. Bourgeois and S.-Q. Zheng, Eds., vol. 5022. Springer, 2008, pp. 196–207.*

[25] *L. Kleinrock,* Queuing Systems (Vol1 and Vol2)*. Wiley-Interscience, 1975.*

[26] *R. Nelson,* Probability, stochastic processes, and queueing theory: the mathematics of computer performance modeling*. New York, NY, USA: Springer-Verlag New York, Inc., 1995.*

[27] *P. A. Bernstein, V. Hadzilacos, and N. Goodman,* Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.*

[28] *R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in* Proc. of PPOPP*, 2008.*

[29] *R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In* Proc. of the 15th International Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, Apr. 2010.*