On the Analytical Modeling of Concurrency Control Algorithms for Software Transactional Memories: the Case of Commit-Time-Locking

Pierangelo Di Sanzo[a], Bruno Ciciani[a], Roberto Palmieri[a], Francesco Quaglia[a], Paolo Romano[b]

[a] *Dipartimento di Informatica e Sistemistica, Sapienza Universita' di Roma, Italy*
[b] *INESC-ID, Lisbon, Portugal*

Abstract

We present an analytical performance modeling approach for concurrency control algorithms in the context of Software Transactional Memories (STMs). We consider a realistic execution pattern where each thread alternates the execution of transactional and non-transactional code portions. Our model captures dynamics related to the execution of both (i) transactional read/write memory accesses and (ii) non-transactional operations, even when they occur within transactional contexts. 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. The presented analytical model has been validated against simulation results based on workload profiles derived by tracing applications proper of the STAMP benchmark suite, running on top of the TL2 transactional memory layer.

Keywords: Software Transactional Memories, Performance Evaluation,

Email addresses: disanzo@dis.uniroma1.it (Pierangelo Di Sanzo), ciciani@dis.uniroma1.it (Bruno Ciciani), palmieri@dis.uniroma1.it (Roberto Palmieri), quaglia@dis.uniroma1.it (Francesco Quaglia), romanop@gsd.inesc-id.pt (Paolo Romano)

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]. As an example, if blindly ported to STM environments, database oriented concurrency control schemes based on explicit wait (sleep) phases, actuated on top of operating system supported mutex and/or semaphores, would induce excessive overhead and non-negligible thread (re-)schedule delay. These costs were instead affordable in databases by amortizing them across (subsequent) stable storage interactions for both read operations and log writes.

According to the above considerations, the wide set of database oriented performance analysis results (see, e.g., [9, 10, 11, 12, 13, 14]) 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, 15, 16]) is the definition 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 thread-level model 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 transaction-level 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 transaction-level model for the case of the Commit-Time-Locking (CTL) concurrency control algorithm, which is adopted by several popular STM systems, such as TL2 [15]. 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 provide a brief introduction to STMs. In Section 3 we discuss literature results related to our contribution. The analytical modeling methodology, together with the specific model instantiation for CTL is provided in Section 4. The comparison between model and simulation results is provided in Section 5. Section 6 concludes the paper.

2. Brief Overview on STMs

While early proposals for (S)TM architectures date back to 90s [18, 2], the research on this topic has been largely dormant till 2002, when the advent of multi-core processors made parallel programming exit from the niche of scientific and high-performance computing and turned it into a mainstream concern for the software industry. One of the main challenges posed by parallel programming consists in synchronizing concurrent access to shared memory by multiple threads. Programmers have traditionally used locks, but lock-based synchronization has well-known pitfalls. Simplistic coarse-grained locking does not scale well, while more sophisticated fine-grained

locking risks introducing deadlocks and data races. Furthermore, scalable libraries written using fine-grained locks cannot be easily composed in a way that retains scalability and avoids deadlock and data races [1].

By bringing the proven abstraction of atomic transaction, used for decades in databases, to parallel programming, (S)TMs allow freeing the programmers from the burden of designing and verifying complex fine-grained lock synchronization schemes. By avoiding deadlocks and automatically allowing fine-grained concurrency, transactional-language constructs enable the programmer to compose scalable applications safely out of thread-safe libraries.

Unlike ACID database transactions, STMs' transactions do not ensure durability, but encompass operations accessing (reading/writing) data stored in volatile memory. Aside from this, transactions are atomic and isolated. Atomicity ensures that, if a transaction commits, then all of its memory operations appear to take effect; if a transaction aborts, conversely, all of its data manipulations are rolled back, as if the transaction had never been executed. The isolation property, on the other hand, provides the illusion that transactions are executed in a serial fashion, which allows programmers to reason serially on the correctness of their applications. Of course, (S)TM systems do not really execute transactions serially. Instead, "under the hood" they allow multiple transactions to execute concurrently as long as they can still ensure atomicity and isolation for each transaction.

Another significant difference between transactions in database and STM environments is related to their lifetime. As already mentioned, STM transactions only encompass in memory operations, not entailing any access to persistent storage. Additionally, transactional access to memory is mediated by lightweight language primitives (typically the *atomic* {} construct) that do not suffer of the overheads for SQL parsing and plan optimization typical of database environments. As a direct consequence, even when considering complex benchmarking applications, the average execution time of STM transactions is typically two or three orders of magnitude smaller than in database environments [19].

Finally, it has been shown [20] that the effects of observing inconsistent states can be much more severe in STMs than in databases. In STMs, in fact, transactions can be used to manipulate program variables whose state directly affects the execution flow of user applications. As such, observing arbitrarily inconsistent memory states (as it is allowed, for instance, by the optimistic concurrency control algorithm [21], a classic and widely used solution in database environments) could lead applications to get trapped in

infinite loops or in exceptions that could never be triggered in any sequential execution. This is not the case for database systems, where transactions are triggered via an SQL interface with precisely defined (and more restricted) semantic, and are executed in a sandboxed component (the DBMS) which is designed not to suffer from crashes or hangs in case the concurrency control scheme allowed observing inconsistent data snapshots.

3. Related Work

Perhaps unsurprisingly, the wide majority of existing performance studies on STMs are based on the empirical comparison of different implementation choices, e.g., [7, 15], and on a (very limited) number of simulation-based studies, [22, 23] (the latter one being actually targeted to hardware-implemented transactional-memory systems).

The work in [24] 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 [24] 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. Second, the model in [24] 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. 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. Additionally, we explicitly model the relation between the final perceived performance and the time interval spent by each thread outside transactional contexts.

A queuing theory based analytical model is proposed in [25] to evaluate and compare the performance of lock-based and STM-based synchronization schemes. The main limitation of this model is due to the assumption that all

transactions (or critical sections) access the same identical memory locations. Conversely, our model captures accesses to distinct locations.

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., [11, 12, 13, 26, 27, 28] for the case of centralized database systems, and in [14, 29] for the case of distributed/replicated databases. However, as already mentioned, the execution time of STM transactions is typically several orders of magnitudes smaller than the counterpart in DBMS scenarios [19], 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 [15]). 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.

Finally, concurrency control protocols for database systems, and their impact on performance, have been extensively studied via simulation [30, 31, 32, 33, 34], which is a technique orthogonal to the analytical approach provided in this paper.

4. The Analytical Model

4.1. Basics

As typical of STM applications/benchmarks [17, 35, 36] we consider a fixed number k of threads, each of which executes on a distinct CPU-core. Threads alternate 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 a number of transactional operations (namely, either *read* or *write* operations) on shared data items and finally ends by issuing a *commit* operation. Overall, 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 on

shared data items, thus exclusively operating within its private workspace, e.g., by accessing its own stack.

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., [24]), 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 denote as t_{tcb} and t_{ntcb}, respectively, the expected duration of *tcb* and *ntcb*. 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 a time interval whose average value is denoted as $t_{backoff}$, at the end of which it gets restarted. Conforming to common implementations/settings [35, 36], the thread taking care of the execution of the transaction gets temporarily suspended, and resumes right upon the end of the back-off period.

4.2. Modeling Approach Overview

As discussed above, 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 thread-level model, is presented in Section 4.3. It allows determining 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 transaction-level model, one instance of which, tailored to CTL, is presented in Section 4.4. The later modeling component 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 transaction-level model, 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 thread-level model 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 [15] (see Section 4.4), with alternative ones either relying on different modeling approaches and/or targeting different concurrency control algorithms.

4.3. Thread-level 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) [37]. Each state of the CTMC is marked and identified by a couple of integers (i, j) representing, respectively, the number of threads

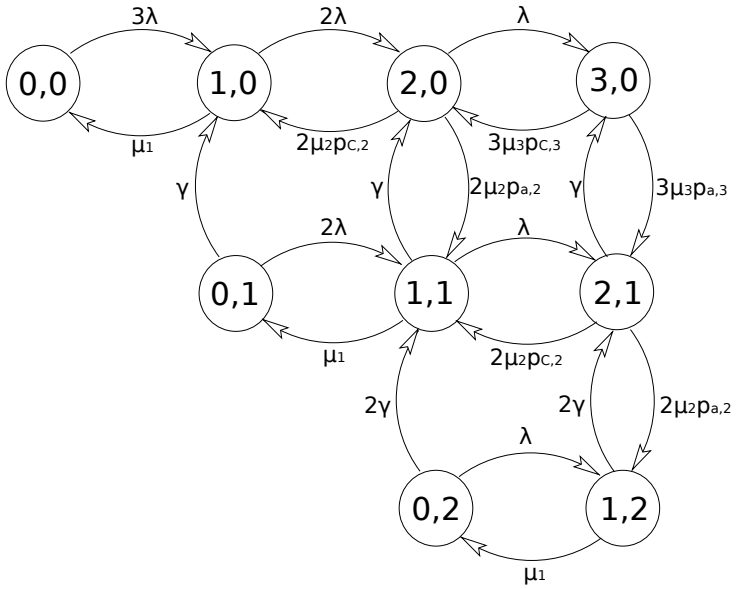

Figure 1: State transition diagram of the CTMC for $k = 3$.

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 constraint

$i + j \leq k$.

For each state $(i, \cdot)$, with $i > 0$, the model takes as input parameters (i) the rate μ_i according to which transactions are run within the system (independently of whether the transaction run gets aborted or committed), and (ii) 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 transaction-level model 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 note that modeling the system via a CTMC maps onto assuming that *ntcb*, the interarrival times of transactions to the commit phase, and the back-off interval have an exponentially distributed duration. Possible

extensions of the model to cope with cases where the values of λ and μ_i represent the mean of generic distributions will be discussed in Section 4.7.

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 new transaction run after the completion of the execution of a non-transactional code block, is equal to $\lambda \cdot (k - i - j)$;
- 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 $i \cdot \mu_i \cdot p_{c,i}$;
- 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 $i \cdot \mu_i \cdot p_{a,i}$;
- 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 $j \cdot \gamma$, where $\gamma = \frac{1}{t_{backoff}}$.

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. An example of the CTMC for the case of three threads (namely $k = 3$) is depicted in Figure 1.

As typically expected in any real system, assuming for any state where $i \in [1, k]$ that $\mu_i > 0$, $p_{c,i} \neq 0$ 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), the CTMC is irreducible, and is formed by an ergodic set of states. Thus the stationary probability vector $\mathbf{v}$ 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 [38]. 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 solving 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)

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

(where S' is the subset of S containing any state where $i > 0$). The overall transaction commit and abort probabilities, denoted as p_c and p_a, can be accordingly evaluated, using the below expressions

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

and

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

4.4. Transaction-level model: the CTL Case

In this section we introduce an analytical model of Commit-Time-Locking (CTL) concurrency control, focusing on the version implemented within the TL2 STM layer [15]. This version is considered as one of the best performing concurrency control algorithms for typical STM workloads. We will start by overviewing such a target version of the CTL algorithm, and then we will move to the presentation of its analytical model.

4.4.1. Algorithm Overview

Unlike, e.g., strict 2PL [39], CTL schemes do 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. Manipulation of the GVC typically relies on a Compare-and-Swap (CaS) operation directly exploiting atomic sequences of machine instructions (e.g., via the LOCK prefix in IA-32 compliant processors). In other words, each transaction updates the GVC as an acyclic, one shot operation, which does not require software spin-locking for accessing the corresponding critical section. Hence, any delay in the access to the GVC is only related to the underlying firmware protocol used to support the atomicity of the machine instruction pattern implementing the CaS.

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 greater 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 read operations and, as we shall discuss below, also at commit time. Accordingly, the *opacity* property [20] is guaranteed, which ensures that the snapshot observed by any transaction (including transactions that are eventually aborted), is equivalent to the one that would have been observed according to some serial execution history. As discussed in [20], 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, updating their WVCs with the value of the GVC. All the acquired locks are released at the end of the commit phase, or upon the abort.

By the above description, we have that a read operation on a data item that was previously written by the transaction gives rise to an access to the transaction private workspace. Thus it is not subject to the previously depicted read validation mechanism. In other words, the validation mechanism is used for read operations associated with any data item that has not already been accessed in write mode by the transaction.

4.4.2. Analytical Model

In order to simplify presentation, we present the model in an incremental fashion. We start by presenting a model relying on the following set of assumptions:

- the accesses (both in read or write mode) to data items in the transactional shared memory are uniformly distributed;
- all the transactions encompass the same amount n of transactional accesses;
- the sequence of read operations issued on shared data items form a Poisson process.

A discussion on how to relax the above assumptions will then be provided in Section 4.4, Section 4.6 and Section 4.7.

As previously discussed, the transaction-level model computes the transaction run 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 $1 \leq i \leq k$. We analyze the case $i = 1$ and $i \neq i$ separately.

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 access operation on a shared data item, is equal to

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

where we denote with n the number of transactional operations on shared data items within a transaction, with p_{write} the probability that the access is in write mode, and with $(1 - P_{write})$ the probability that the access is in read mode.

As already discussed in Section 4.4.1, if the transaction accesses a data item x in write mode, producing a new version, any subsequent read on x by the same transaction will return the previously written version, retrieving it from the transaction private workspace. Analogous considerations apply for subsequent writes over the same data item x, which will simply update the copy of x buffered within the private workspace. Hence read/write operations issued on previously updated data items are simply not taken into account by the parameter n. On the other hand the cost of the corresponding accesses within the private workspace is encapsulated in t_{tcb}.

By the previous notation, we have that

$$n_{write} = n \cdot p_{write} \tag{7}$$

is the average number of shared data items accessed by the transaction in write mode, and

$$n_{read} = n \cdot (1 - p_{write}) \tag{8}$$

is the average number of read operations occurring on distinct shared data items that were not already accessed by the transaction in write mode.

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_{write} of write-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 in case of succesful commit attempts. Thus, to simplify, we ignore the cost of aborts when we evaluate the average lock holding time, by assuming that if a transaction successfully completes the lock acquisition phase, it holds the locks for an average time equal to t_{commit}.

Let us now compute the probability for a transaction to abort while executing a read operation on a shared data item x, given that it finds the corresponding write-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 write-locks for all the

data items in its write-set. Given that we are assuming uniformly distributed accesses to distinct data items within a transaction, it follows that the probability for a committing transaction to have a specific data item within its write-set is n_{write}/d. Exploiting the aforementioned assumption of Poissonianity of the arrival process of read operations, we can rely on the PASTA property (Poisson Arrivals See Time Averages) [40] to compute the probability to incur in a raised write-lock during a read operation as

$$p_{lock} = l_r \cdot t_{commit} \cdot \frac{n_{write}}{d} \tag{9}$$

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

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

where p_{avf} is the probability for a transaction to abort during the read-set validation phase. Such a transaction contributes anyway to the lock-acquisition rate since read set validation occurs after write-locks are acquired at commit time over any written data item. 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. The rate u_r at which a data item is updated by transactions is equal to

$$u_r = c_r \cdot \frac{n_{write}}{d} \tag{11}$$

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

$$c_r = \frac{1}{r_{t,i}} \cdot p_{c,i} \cdot (i-1) \tag{12}$$

Upon the l-th operation by transaction T, 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 shared data item that has been updated by some successfully committing transaction after T started can be expressed as

$$p^o_{u,l} = 1 - e^{-u_r \cdot t_{b,l}} \tag{13}$$

In the above expression, in order to avoid overcomplicate the model, we decided not to capture the case of repeated transactional read operations on the same data item. In this case, in fact, the invalidation window for a data item x would no longer correspond to the time elapsed since the beginning of the transaction (namely $t_{b,l}$), but would be equal to the (average) time elapsed since the last occurrence of a read on x. Clearly, the error introduced by this modeling choice depends on the actual frequency of occurrence of repeated read operations on the same data item during the same transaction. On the other hand, the model captures faithfully the effects of a frequent optimization technique (possibly implemented at the compiler level), which allows sparing subsequent read operations issued within the same transaction on the same data item from the cost of validation. To this end, it is sufficient to copy the values read from the shared transactional memory to thread local variables, and to redirect subsequent reads on these data items (within the same transaction) towards the thread local variables. Note that, since with this optimization subsequent read operations on a data item do not target the shared transactional memory, they do not even need to be accounted for while computing the value of the parameter n.

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

$$p^o_{a,1} = (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^o_{u,1}) \tag{14}$$

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

$$p^o_{a,l} = p^o_{na,l} \cdot (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^o_{u,l}) \tag{15}$$

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

$$p^o_{na,1} = 1 \tag{16}$$

and

$$p^o_{na,l} = (1 - p^o_{a,l-1}) \cdot p^o_{na,l-1} \tag{17}$$

In equations (14)-(15) we have assumed that the event of finding a write-lock raised on the shared data item by a concurrent transaction currently

attempting to commit, and the event that the same data item was previously updated by a different (already committed) concurrent transaction are independent. Overall, independence is related to that these events belong to commit time activities across distinct transactions.

The probability p_{alc} for a transaction T to abort at commit time due to lock contention while acquiring the write-locks 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, as in [9], we approximate this last probability, namely p_{wlc}, with an upper bound value, that is

$$p_{wlc} = 1 - (1 - p_{lock})^{n_{write}} \tag{18}$$

Thus we have

$$p_{alc} = p^{o}_{na,n+1} \cdot p_{wlc} \tag{19}$$

where we recall that $p^{o}_{na,n+1}$ 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

$$p^{la}_{na} = p^{o}_{na,n+1} \cdot (1 - p_{wlc}) \tag{20}$$

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 one data item x belonging to the read-set of T has the corresponding write-lock raised by another transaction, or if a new version of x has been committed after the validation executed by T during its last read operation on x. We denote with $p^{r}_{u,l}$ the probability that the shared data item accessed in read mode at the l^{th} operation by T has been updated after the last validation (occurred upon the corresponding last read operation on x). We calculate this probability as follows

$$p^{r}_{u,l} = 1 - e^{-u_r \cdot t_{v,l}} \tag{21}$$

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

$$t_{v,l} = (t_{tcb} + t_{op}) \cdot (n - l + 1) + t_{commit} \tag{22}$$

Analogously to what we did in equation (15), we evaluate the abort probability due to failure in the validation of the data item associated with the l^{th} transactional access of T as follows

$$p^r_{a,l} = p^r_{na,l} \cdot (1 - p_{write}) \cdot (p_{lock} + (1 - p_{lock}) \cdot p^r_{u,l}) \tag{23}$$

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

$$p_{avf} = p^{la}_{na} \cdot p_{rvf} \tag{24}$$

where

$$p_{rvf} = \sum_{l=1}^{n} p^r_{a,l} \tag{25}$$

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

$$p_{c,i} = p^{la}_{na}(1 - p_{rvf}) \tag{26}$$

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 during that phase. Let us denote with

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

$$t_{a,l} = t_{begin} + l \cdot (t_{tcb} + t_{op}) + t_{abort} \tag{27}$$

- $t_1 = t_b + t_{tcb} + t_{abort}$ 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

$$t_b = t_{begin} + n \cdot (t_{tcb} + t_{op}) \tag{28}$$

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

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

Overall, the average transaction execution time can be expressed as

$$r_{t,i} = \sum_{l=1}^{n}(p^{o}_{a,l} \cdot t_{a,l}) + p_{alc} \cdot t_1 + p_{avf} \cdot t_2 + p_c \cdot t_3 \tag{29}$$

Now let us evaluate the time t_{GVC} spent by any committing transaction while updating the GVC. We consider this time logically included in t_{commit}, thus t_{commit} is the sum of two terms, namely t'_{commit} and t_{GVC}, where t'_{commit} is the time to execute all the other operations, distinct from GVC manipulation, during the commit phase. As explained in Section 4.4.1, the atomic operations required for the update of GVC typically rely on firmware level protocols offered by modern SMP and/or multi-core machines. Assuming fairness by these protocols vs different CPU/core requests, we model the delay for performing an atomic increment of the GVC, denoted as t_{GVC}, by means of an M/D/1 queue [37] with service rate $\mu = \frac{1}{t^{inc}_{GVC}}$ (where t^{inc}_{GVC} expresses latency for the updating machine instructions, once the firmware has granted access to the corresponding critical section) and arrival rate $\beta = l_r$ (note that the increment of the GVC is performed by any transaction that successfully acquired all the requested locks). According to this modeling approach, t_{GVC} corresponds to the residence time within the M/D/1 queue, namely

$$t_{GVC} = (1 + \frac{\rho}{2 \cdot (1 - \rho)}) \cdot t^{inc}_{GVC}, \tag{30}$$

where $\rho = \frac{\beta}{\mu}$.

4.5. Coping with Multiple Transaction Classes

In this section we extend the analytical model by considering the case of q different transactional classes, associated with different transaction profiles. A transaction of class m, with $m \in [1, q]$ executes n^m operations and each operation is a write operation with probability p^m_{write}. Hence $n^m \cdot (1 - p^m_{write})$ expresses the total amount of distinct transactional read accesses. A thread runs a transaction of class m with probability P^m. Also, t^m_{commit} and t^m_{abort} are the expected commit time and abort time for a transaction of class m, respectively.

4.5.1. Multi-class Thread-level Model

For q transactional classes, the state of the CTMC can be identified by $2q$ integers $(i_1, .., i_q, j_1, .., j_q)$ where i_m and j_m (with $m \in [1, q]$) represent the

number of threads running transactions of class m and the number of threads in backoff due to an abort of a transaction of class m, respectively. Note that $i_1 + .. + i_q + j_1 + .. + j_q \leq k$ for each state of the CTMC.

For any state $(i_1, .., i_q, j_1, .., j_q)$, the average transaction execution rate and the transaction commit probability for a transaction of class m depend on the mix of active transactions in that state. Thus we denote them as $\mu^m_{i_1,..,i_q}$ and $p^m_{c,i_1,..,i_q}$, respectively. Also, the abort probability for a transaction of class m while residing in state $(i_1, .., i_q, j_1, .., j_q)$ is denoted as $p_{a,i_1,..,i_q} = 1 - p_{c,i_1,..,i_q}$.

The rate according to which a thread executes a new transaction of class m is $\lambda_m = P^m / t_{ntcb}$. The rules defining the transition rates from any two states of the CTMC are the following:

- for $i_1 + ... + i_q + j_1 + ... + j_q < k$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_q)$ to state $(i_1, ..., i_m + 1, ..., i_q, j_1, ..., j_q)$, associated with the activation of a run of a transaction of class m is equal to $\lambda_m(k - i_1 - ... - i_q - j_1 - ... - j_q)$;

- for $i_m > 0$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_q)$ to state $(i_1, ..., i_m - 1, ..., i_q, j_1, .., j_q)$, associated with a successful commit event of a transaction of class m is equal to $i_m \mu^m_{i_1,...,i_q} p^m_{c,i_1,...,i_q}$

- for $i_1 + ... + i_m + ... + i_q \geq 2$ and $i_m \geq 1$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_m, .., j_q)$ to state $(i_1, ..., i_m - 1, .., i_q, j_1, .., i_m + 1, ..., j_q)$, associated with an abort event of a transaction of class m is equal to $i_m \mu^m_{i_1,...,i_q} p^m_{a,i_1,...,i_q}$

- for $j_m > 1$, the transition rate from state $(i_1, ..., i_m, ..., i_q, j_1, ..., j_m, ..., j_q)$ to state $(i_1, ..., i_m + 1, ..., i_q, j_1, ..., j_m - 1, ..., j_q)$, associated with the termination of a back-off period of an aborted transaction of class m is equal to $\gamma \cdot j_m$.

The cardinality of the set S of states of the CTMC can be evaluated by considering that at given time a thread can be in one of $2 \cdot q + 1$ different states (namely, q states corresponding to the execution of a transaction of class m, q states corresponding to the back-off period after an abort event of a transaction of class m, and 1 state corresponding to the execution of a non-transactional code block). As a consequence, the number of states of the CTMC is the k-combination with repetition of $2 \cdot q + 1$ elements, that

is the binomial coefficient $\begin{pmatrix} (2 \cdot q + 1) + k - 1 \\ k \end{pmatrix}$, where we have to exclude the states where all the threads are in back-off.

By relying on the same Poissonianity assumptions made in Section 4.3, and by equation (1) applied to the multi-class CTMC, we can evaluate the stationary probability vector v. Hence, the execution rate τ_m of transactions of class m can be expressed as

$$\tau_m = \sum_{(s') \in S'} v_{s'} \cdot i_m \cdot \mu^m_{s''} \cdot p^m_{c,s''} \tag{31}$$

where we used s' in place of $i_1, .., i_q, j_1, .., j_q$ and s'' in place of $i_1, .., i_q$, and where S' is the subset of S containing any state where $i_m > 0$. The overall system throughput is

$$\tau = \sum_{m=1}^{q} \tau_m \tag{32}$$

The commit probability for a transaction of class m is

$$p^m_c = \frac{\sum_{(s') \in S'} v_{s'} \cdot p^m_{c,s}}{\sum_{(s') \in S'} v_{s'}} \tag{33}$$

4.5.2. Multi-class Tread-level Model for CTL

Fixed a configuration of active transactions $i_1, .., i_q$, the thread-level model is in charge of evaluating for each transactional class m the transaction run rate $r^m_{t,i_1,..,i_q}$ and the transaction commit probability $p^m_{c,i_1,..,i_q}$. As for the single-class models, if there is just one active transaction, that is $i_m = 1$ and $i_w = 0$ for each $w \neq m$, the average transaction execution time of the transaction of class m is

$$r^m_{t,i_1,..,i_q} = t_{begin} + n^m \cdot t_{op} + (n^m + 1) t_{tcb} + t^m_{commit} \tag{34}$$

where t^m_{op}, namely the average time to execute an access operation on a shared data item for a transaction of class m, is equal to

$$t^m_{op} = t_{read}(1 - p^m_{write}) + t_{write} \cdot p^m_{write} \tag{35}$$

When the number of active transactions is greater that one we use the same iterative approach as in Section 4.4.2, by stopping the iterations when two consecutive values of the commit probability for transactions of each

class m (if $i_m \geq 1$) differ by at most an ϵ. Also, in what follows we use the same assumptions and considerations as in Section 4.4.2.

When a transaction of class m is active, its concurrent transactions are:

- i_x active transactions of each other class x such that $x \neq m$ and $i_x \geq 1$;

- $i_m - 1$ active transactions of the same class m, if $i_m \geq 2$.

At the start of each iterative step we evaluate the following parameters. The lock rate associated with transactions of each class x, expressed as

$$l_r^x = \frac{1}{r^x_{t,i_1,..,i_q}} \cdot (p^x_{c,i_1,..,i_q} + p^x_{avf}) \tag{36}$$

where p^x_{avf} is the probability for a transaction of class x to abort during the read-set validation phase. The probability for a transaction of class m to find a write-lock raised while issuing a read operation, which is expressed as

$$p^m_{lock} = \sum_{x=1,x\neq m}^{q} l_r^x \cdot i_x \cdot t_{commit} \frac{n^x \cdot p^x_{write}}{d} + l_r^m \cdot (i_m - 1) \cdot t_{commit} \frac{n^m \cdot p^m_{write}}{d}. \tag{37}$$

The commit rate associated with transactions of class x, which is expressed as

$$c_r^x = \frac{1}{r^x_{t,i_1,..,i_q}} \cdot p^x_{c,i_1,..,i_q} \tag{38}$$

Finally, the update rate by concurrent transactions of a transaction of class m, which is expressed as

$$u_r^m = \sum_{x=1,x\neq m}^{q} c_r^x \cdot i_x \frac{n^x \cdot p^x_{write}}{d} + c_r^m \cdot (i_m - 1) \frac{n^m \cdot p^m_{write}}{d}. \tag{39}$$

After solving the previous equations, we evaluate in each iterative step all the parameters we list below. The probability $p^{o,m}_{u,l}$ for a read operation, executed as the l-th operation of a transaction T of class m, to access a data item that has been updated by some successfully committing transaction after T started, which can be expressed as

$$p^{o,m}_{u,l} = 1 - e^{-u_r^m \cdot t^m_{b,l}} \tag{40}$$

where $t^m_{b,l}$ is the average elapsed time since the validation performed on the data item upon the read access by the transaction of class m, which can be evaluated the same way as the single-class case.

The probability to abort while executing the 1-st operation on a shared data item for a transaction of class m, expressed as

$$p_{a,1}^{o,m} = (1 - p_{write}^{m}) \cdot (p_{lock}^{m} + (1 - p_{lock}^{m}) \cdot p_{u,l}^{o,m}), \tag{41}$$

and the probability to abort while executing the l-th operation with $l \geq 2$ for a transaction of class m, expressed as

$$p_{a,l}^{o,m} = p_{na,l}^{o,m} \cdot (1 - p_{write}^{m}) \cdot (p_{lock}^{m} + (1 - p_{lock}^{m}) \cdot p_{u,l}^{o,m}) \tag{42}$$

where $p_{na,l}^{o,m}$ is the probability of not aborting until the completion of the $(l-1)^{th}$ operation, for which we have

$$p_{na,1}^{o,m} = 1 \tag{43}$$

and

$$p_{na,l}^{o,m} = (1 - p_{a,l-1}^{o,m}) \cdot p_{na,l-1}^{o,m} \tag{44}$$

The contention probability during write-lock acquisition phase for a transaction of class m can be then approximated as

$$p_{wlc}^{m} = 1 - (1 - p_{lock}^{m})^{n^m \cdot p_{write}} \tag{45}$$

Hence the probability for a transaction of class m to abort at commit time due to write-lock contention is

$$p_{alc}^{m} = p_{na,n^m+1}^{o,m} \cdot p_{wlc}^{m} \tag{46}$$

The probability for a transaction of class m not to be aborted during its execution and to succeed in its commit-time lock acquisition phase is

$$p_{na}^{la,m} = p_{na,n^m+1}^{o,m} \cdot (1 - p_{wlc}^{m}) \tag{47}$$

The probability that a data item in the read-set of a transaction belonging to class m, which is accessed at the l-th transactional operation, has been updated when T executes the read-set validation can be expressed as

$$p_{u,l}^{r,m} = 1 - e^{-u_r^m \cdot t_{v,l}^m} \tag{48}$$

where $t_{v,l}^{m}$ is the elapsed time since the original validation, which can again be computed teh same way as for the single-class case.

Thus the abort probability due to failure in the validation of the l^{th} data item within the read-set can be evaluated as follows

$$p_{a,l}^{r,m} = p_{na,l}^{r,m} \cdot (1 - p_{write}^{m}) \cdot (p_{lock}^{m} + (1 - p_{lock}^{m}) \cdot p_{u,l}^{r,m}) \tag{49}$$

where $p_{na,1}^{r,m} = 1$ and, for $l > 1$, $p_{na,l}^{r,m} = (1 - p_{a,l-1}^{r,m}) \cdot p_{na,l-1}^{r,m}$. Hence, the probability for a transaction of class m to abort during the read-set validation phase can be expressed as

$$p_{avf}^{m} = p_{na}^{la,m} \cdot p_{rvf}^{m} \tag{50}$$

where

$$p_{rvf}^{m} = \sum_{l=1}^{n^m} p_{a,l}^{r,m} \tag{51}$$

Finally we can evaluate the probability of successful commit when residing within state
$(i_1, .., i_q)$ as

$$p_{c,i_1,..,i_q}^{m} = p_{na}^{la,m}(1 - p_{rvf}^{m}) \tag{52}$$

For brevity we do not detail the equations for the evaluation of average transaction execution time and t_{GVC} because they can be simply derived by using the same approach we have show at the end of Section 4.4.2. In fact, the evaluation of the average transaction execution time for a transaction of class m can be done by using the already provided set of equations, by substituting the parameter values that depend on the specific transactional class with the ones we calculated in this section. Regarding the evaluation of t_{GVC}, by using the approach discussed in Section 4.4.2, we have just to evaluate λ as the sum of the lock rate due to all the active transactions across the different classes, namely

$$\lambda = \sum_{m=1}^{q} l_r^m \cdot i_m \tag{53}$$

4.6. Hints on Model Extension for Non-uniform Data Access

By relying on the approach in [9], which has been proposed for the case of concurrency control algorithms in database systems, our model could be extended to cope with non-uniform data accesses. We provide hints on how the extension could be realized in this section.

The proposed approach considers the whole set of d shared data items as grouped in s disjoint subsets, possibly exhibiting different cardinalities. The

set of n operations executed by a transaction are grouped in s different subsets, possibly exhibiting different cardinalities, where each operation accesses a data item belonging to a different data subset. The accesses executed on each subset of data items by a transaction are uniformly distributed over the subset.

Different subsets of data items exhibit different access frequencies. As a consequence, the probability to find a lock raised on a data item and the data item update rate are different for each specific subset of data items. To evaluate them for a given subset we can use the same equations (9) and (11) by considering, in place of n, only the subset of operations executed by the transactions on that specific subset of data items. Consequently, the subsequent equations, expressing the abort probability for a transaction, can be determined by considering the probability of finding the lock raised and the data item update rate as differentiated for each subset, and then weighting the corresponding effects by the fraction of operations executed on the specific subset.

4.7. On Removing Exponential Assumptions

In the model presented so far we have exploited the assumption of exponential distribution of several random variables. In this section we discuss how our modelling approach could be extended to relax these assumptions.

As for the thread-level model in Section 4.3, the reliance on a CTMC representation maps onto exponential assumptions for the frequency with which i) transactions complete their execution (either due to a commit or to an abort event), ii) transactions exit from their back-off period following an abort event, and iii) the execution of a non-transactional code block is completed. We also recall also that the output of the thread-level model is represented by the stationary probability vector v associated with the CTMC.

If one wanted to account in the model for generic, but known, distributions of the above transition rates, it would be simply sufficient to replace the CTMC with an equivalent Semi-Markov process [41]. At this point one should rely on well-known solution techniques [41] for the stationary probability vector of Semi-Markov processes.

For what concerns the transaction-level model, we exploited the assumption on the exponential distribution of the transactions' arrival rate to the commit phase to compute $p^o_{u,l}$ in equation (13) and $p^r_{u,l}$ in equation (21). Further, we exploited the assumption that the execution rate of read operations

on shared data items forms a Poisson process to derive the expression of p_{lock} in equation (9).

As for equations (13) and (21), they could be extended to account for arbitrary distributions of the transaction arrivals to the commit phase. We could in fact write them as

$$p^o_{u,l} = \Phi(t_{b,l}, u_r) \tag{54}$$

$$p^r_{u,l} = \Phi(t_{v,l}, u_r) \tag{55}$$

where $\Phi(t, \eta), t \in (0, \infty)$ expresses the generic cumulative distribution function of the arrival process to the commit phase, having as $\eta = r_{t,i} = 1/E[\Phi(t, \eta)]$ its average arrival rate, and u_r could be computed as before using equation (11) and equation (12).

More problematic would be, instead, relaxing the assumption that the rate of read operations forms a Poisson process. In equation (9), in fact, we exploited directly the PASTA property [40] of Poisson arrival processes to compute the probability of finding a write-lock busy during a read on a data item x, as the probability for x to be locked at a random instant. However, if one were to assume that the arrival process of read operations on x formed a generic renewal process, one should explicitly account for the dynamic of interleaving between the arrival process of read operations on x and the stochastic process associated with the arrival of transactions that updated x to the commit phase. This would require determining the conditioned probability that, given an arbitrarily small interval $[t-h, t]$, there is a transaction T that is locking the data item x during its commit phase given that a transaction T' issues a read on x in the same time interval, or more formally:

$$lim_{h \to 0} Pr\{X(t-h) = 1 | N(t-h) \geq 1\} \tag{56}$$

where $X(t)$ expresses the number of transactions (that updated x) to be in the commit phase at time t and $N(t)$ is the counting process associated with the arrival of read operations (on x).

5. 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 pre-

sented 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 simulation model of CTL as implemented by the TL2 STM layer [15]. 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 simulation results were obtained by repeating a number of independent runs (with different initial seeds for the random generators) until the amplitude of the 90% confidence intervals on the throughput (committed transactions per second) became smaller than 10% of the average throughput value.

The workload parameters for this study have been selected on the basis of measurement and tracing activities, carried out for the STAMP benchmark [17]. 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).

In our study we focus on two of the applications included in the STAMP benchmark, namely Intruder and Vacation. Intruder is a signature-based network intrusion detection system which processes network packets in parallel via a user-tunable number of threads that concurrently update two main data structures, namely a FIFO queue and a self-balancing tree. In this benchmark, each thread spends around 33% of the time executing transactional code, and generates relatively short transactions, belonging to three different classes (capture, reassembly, and detection), the 90% percent of which exhibit a read plus write set made of up to $n = 71$ items, 30% of which are accessed in write mode. Based on our measurements, we set $t_{tcb} = 0.5\mu sec$, $t_{ntcb} = 5\mu sec$ and $t_{commit} = 2\mu sec$.

Vacation, on the other hand, implements an on-line transaction processing system emulating a travel reservation system. The system is implemented as a set of trees that keep track of customers and their reservations for various travel items. Client threads perform a number of sessions, each one enclosed in a coarse-grained transaction (compared to Intruder), which are again differentiated into three classes (reservations, cancellations, and updates), all interacting with the travel system's data layer. In this application, client threads spend almost all their execution time (92%) executing transactions,

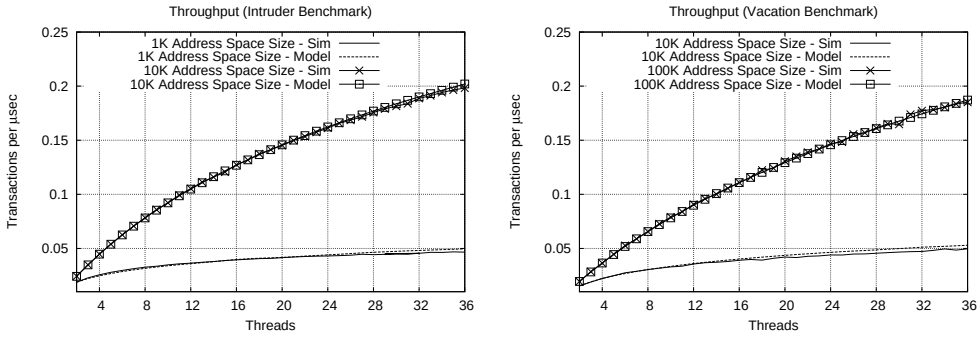

Figure 2: Throughput.

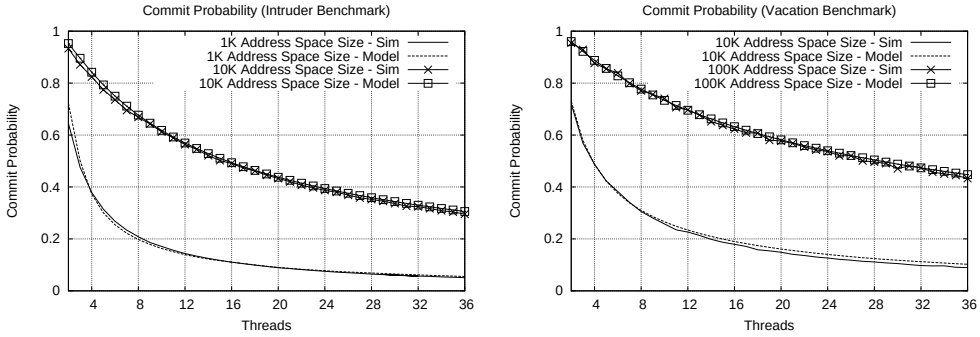

Figure 3: Commit probability.

the 90% percent of which exhibit a read plus write set made of up to $n = 200$ items, 12% of which are accesses in write mode. Based on our measurements, we set $t_{tcb} = 0.2\mu sec$, $t_{ntcb} = 5\mu sec$ and $t_{commit} = 5\mu sec$

In addition to the above parameters, we used our tracing facility to determine also the following set of parameters: $t_{begin} = 0.2\mu sec$, $t_{read} = 0.25\mu sec$, $t_{write} = 0.2\mu sec$, $t_{abort} = 1\mu sec$. Finally, the back-off period, $t_{backoff}$, was set to $2\mu sec$.

By the above description, both the selected benchmark applications entail multi-class transactions. Hence the tracing process and the related outcomes have been used in a differentiated manner depending on whether the target is the validation of the single-class or the multi-class model.

To validate the single-class model, we configured the simulator to generate durations of the above mentioned timing activities based on exponential distributions. On the other hand, the validation of the multi-class version

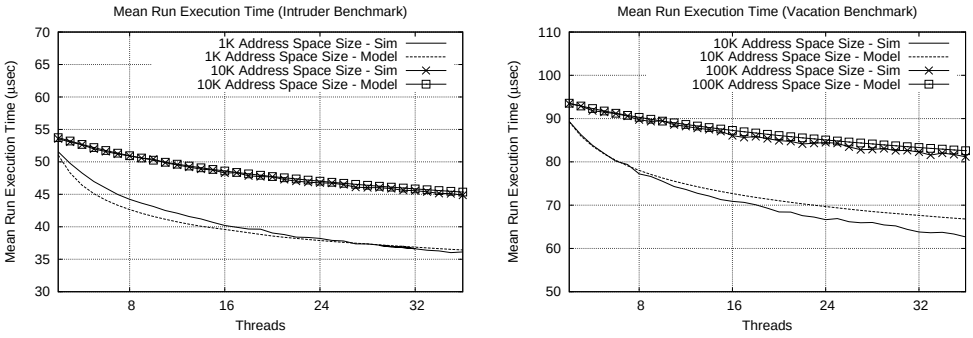

Figure 4: Mean run execution time.

of the model, which captures more in detail the execution dynamics of the STM system, has been performed by replaying within the simulator the exact timing of actions as logged in the execution traces.

For what concerns data accesses, the simulator generates them according to a uniform distribution across the total number of d data items/memory words (in compliance with the assumptions of our analytical model). The parameter d is treated as an independent parameter of the validation study. Note that, once fixed the number of worker threads, variations of d allow capturing settings with differentiated levels of contention, which, in their turn, determine different transactions' abort probabilities. 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 smaller number of distinct memory words. We consider different values for the parameter d, associated, respectively, with reduced and increased values of the benchmarks' data-set size according to the indications provided in [17]. Specifically, for Intruder, we set d to 1,000 and 10,000, whereas, for Vacation, we set d to 10,000 and 100,000.

5.1. Single-class Case

The comparison between analytical and simulation results is based on the following four parameters: (A) the system throughput (Figure 2), (B) the commit probability (Figure 3), (C) the mean execution time evaluated over each single transaction run, independently of whether the run is committed or aborted (Figure 4) and (D) the likelihood of each of the possible causes of transaction abort (Figure 5).

The plots in Figure 2 and Figure 3 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 3, 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 4, we remark how, when considering the case of smaller address spaces, the relatively high contention probability often leads transactions to be early aborted (i.e., 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.) On the other hand, 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. Note that, due to the aforementioned early abort phenomenon, the variance of the mean run execution time grows in high contention scenarios. 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 accuracy of our analytical approach. The only exception is represented by the case of the Vacation benchmark when configured to use the smaller address space. In this case, the accuracy of the analytical model in predicting the mean run execution time is in fact subject to a slight deterioration as the number of worker threads increases. We argue that this is imputable to the fact that the Vacation benchmark comprises transactions whose execution latency is (on average) significantly longer than the Intruder benchmark. This leads to an increase of the variance of the run execution time and to a corresponding amplification of the model's prediction error.

In Figure 5 we evaluate the accuracy of the analytical model in predicting the different causes of aborts for the transactions. Specifically, we set the number of worker threads to eight and report: (i) the probability for a transaction to abort during its execution before reaching the commit phase (recall that this can only happen due to a validation failure during a read operation), denoted as $p_{a,ex} = 1 - p^{o}_{na,n+1}$ (see equations (15-17)); (ii) the probability for a transaction to abort in the commit phase during the writeset lock acquisition, namely p_{wlc} (see equation (18)); (iii) the probability for a transaction to abort in the commit phase due to read-set validation failure, namely p_{rvf} (see equation (25)). Also in this case we observe that the accuracy of the

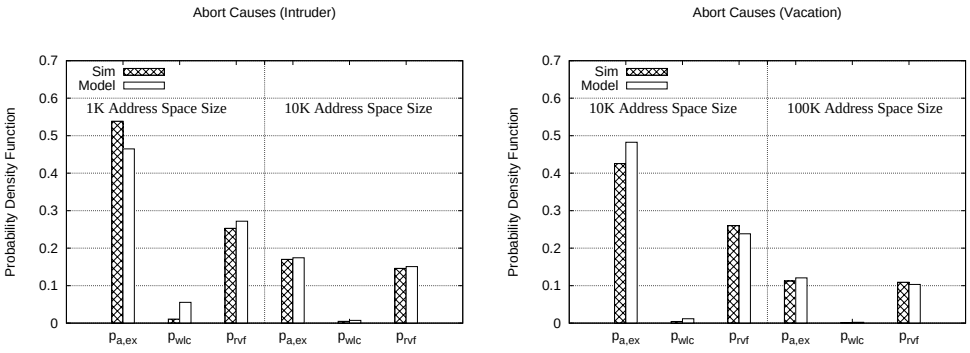

Figure 5: Abort causes.

proposed analytical model is very good for the scenarios in which the benchmarks are configured to use the larger datasets. On the other hand, with smaller datasets, namely the ones associated with very high contention rates (note that the probability of abort is around 0.7 and 0.8 in these scenarios), there is a slight degradation of the analytical model accuracy. We argue that this is imputable to the fact that the error introduced by assuming a Poisson assumption for the distribution of the transaction interarrival time to the commit phase, which remains negligible at low/medium contention levels, shows an increasing trend at very high contention levels. This phenomenon is confirmed by the plots in Figure 6, where we evaluate the goodness of this assumption in different workload scenarios by contrasting the empirical density functions of the transaction interarrival time to the commit phase, as computed by the simulator, and the exponential distribution functions whose average value has been computed via the analytical model. More in detail, the plots on the right side of Figure 6 have been obtained by considering moderate contention scenarios obtained by selecting, for each benchmark, the largest address spaces and degree of concurrency equal to eight, that give rise to probability of abort on the order of 20% and 35% for Vacation and Intruder, respectively. On the other hand, the plots on the left side of Figure 6 are associated with a very high (and, arguably, somewhat pathological in practice) contention scenario, in which we select for each benchmark the smallest address spaces and degree of concurrency equal to eight, that give rise to probability of abort on the order of 70% and 80%, for Vacation and Intruder, respectively. The reported results clearly highlight that, up to medium contention levels, there is an excellent match between the empirical and analytical distributions, thus confirming the validity of the Poissionian-

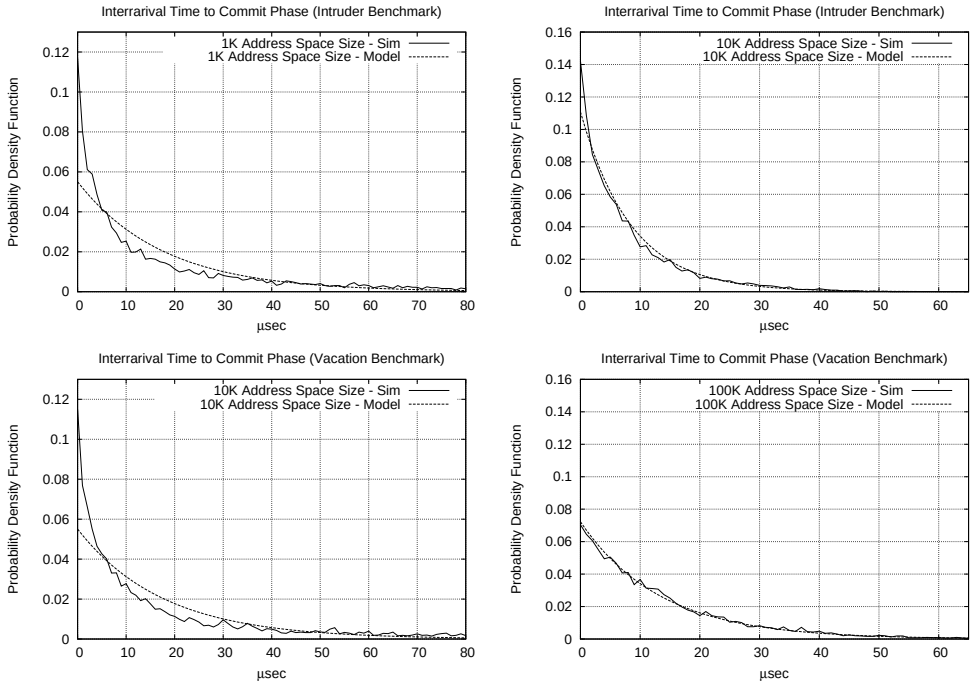

Figure 6: Distribution of the transaction interarrival times to the commit phase.

ity assumption for the commit phase arrival in case the timing of actions natively associated with the transactions follows exponential distributions. The left side plots, conversely, highlight a higher discrepancy between the empirical and analytical density functions in very high contention scenarios.

However, it is interesting to highlight that the degradation of the goodness of the poissionianity assumption leads to a (slight) increase of the model's error only when predicting some internal state variables, such as the likelihood of the various abort causes. On the other hand, the model's accuracy in predicting external performance metrics, such as throughput and commit probability, remains very high across every analyzed workload, even those associated with very high contention rate (see Figure 2 and Figure 3).

5.2. Multi-class Case

In this section we validate the variant of the analytical model capturing multi-class transactional profiles. To this purpose, the timing of accesses to shared memory data items has been simulated by replaying the execution traces of the Vacation benchmark. On the other hand, we used the reduced

data set size selected for this benchmark (i.e., 10,000 data items) in order to stress the accuracy of the model when considering non-minimal contention scenarios. The parameters characterizing this workload are summarized in Table 1.

By the results shown in Figure 7, we have that the analytical model again shows a very good match vs simulative results. In particular, throughput, response time and commit probability for each individual class are evaluated by the model in a very accurate manner while increasing the number of worker threads. Also, the curves show that the matching is good up to a number of worker threads yielding towards flat throughput values.

Parameter	Class 1	Class 2	Class 3
Transaction Class Probability (P^m)	0.898	0.047	0.056
Transaction Class Length (n^m)	154	57	121
Write Probability per Class (p^m_{write})	0.046	0.117	0.080

Table 1: Parameters used for the multi-class study (Vacation benchmark)

6. 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.

Acknowledgements

This work has been partially supported by the "Cloud-TM" project (co-financed by the European Commission through the contract no. 257784), COST Action IC1001 EuroTM, the FCT project ARISTOS (PTDC/EIA-

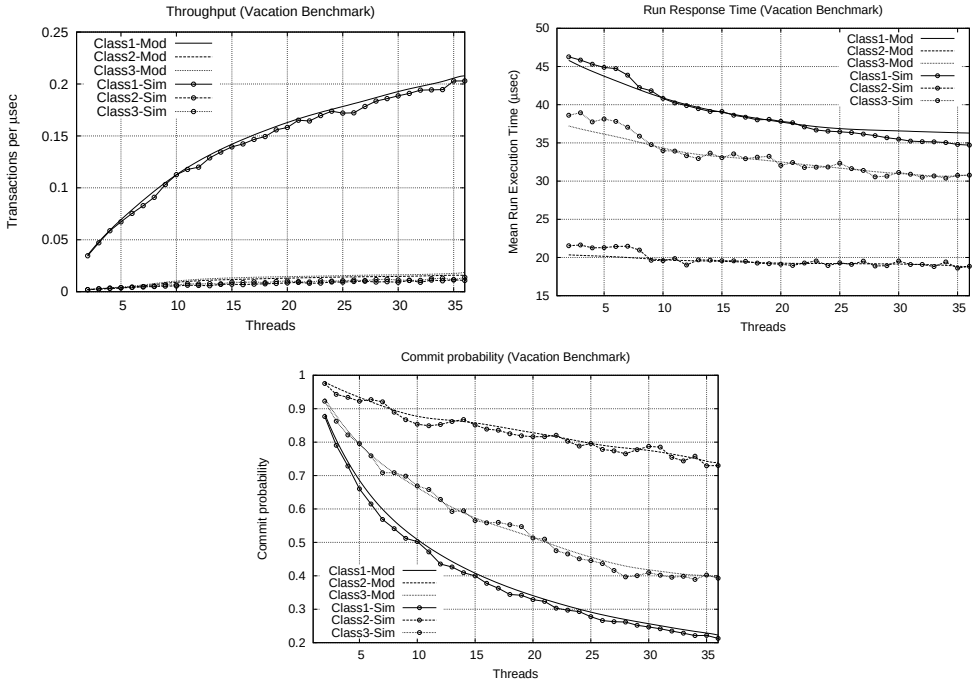

Figure 7: Analytical vs Simulative Results for the Multi-class Scenario.

EIA/102496/2008) and by FCT (INESC-ID multiannual funding) through the PIDDAC Program Funds.

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] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[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] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on Computer architecture*, ser. ISCA '93. New York, NY, USA: ACM, 1993, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/165123.165164

[19] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho, "Evaluating database-oriented replication schemes in software transactional memory systems," in *Proc. of International Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, Oct. 2006.

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

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

[22] A. Heindl and G. Pokam, "Modeling software transactional memory with anylogic," in *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, pp. 1–10.

[23] M. Moir, K. Moore, and D. Nussbaum, "The adaptive transactional memory test platform: a tool for experimenting with transactional code for rock (poster)," in *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. New York, NY, USA: ACM, 2008, pp. 362–362.

[24] 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.

[25] Z. He and B. Hong, "On the performance of commit-time-locking based software transactional memory," in *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 180–187. [Online]. Available: http://portal.acm.org/citation.cfm?id=1581383.1582122

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

[27] 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.

[28] 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.

[29] 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.

[30] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modeling: Alternatives and implications," *ACM Transactions on Database Systems*, vol. 12, no. 4, December 1987.

[31] N. Al-Jumaha, H. Hassaneinb, and M. El-Sharkawia, "Implementation and modeling of two-phase locking concurrency," *Information and Software Technology*, vol. 42, no. 4, pp. 257–273, March 2000, elsevier Science.

[32] M. J. Carey and W. A. Muhanna, "The performance of multiversion concurrency control algorithms," *ACM Transactions on Computer Systems*, vol. 4, no. 4, pp. 338–378, November 1986.

[33] D. R. Ries and M. Stonebraker, "Locking granularity revisited," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 2, 1974.

[34] ——, "Effects of locking granularity in a database management system," *ACM Transactions on Database Systems (TODS)*, vol. 2, no. 3, September 1977.

[35] 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.

[36] M. Ansari, C. Kotselidis, I. Watson, C. C. Kirkham, M. Lujn, 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.

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

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

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

[40] R. W. Wolff, "Poisson arrivals see time averages," *Operations Research*, vol. 30, no. 2, pp. 223–231, 1982. [Online]. Available: http://www.jstor.org/stable/170165

[41] A. A. Tomusyak, "Computation of an ergodic distribution of markov and semi-markov processes," no. 1, 1969.