An analytic framework for performance modeling of software transactional memory

Armin Heindl a,*, Gilles Pokam b

a Computer Networks and Communication Systems, University of Erlangen-Nuremberg, Germany
b Microprocessor Technology Lab, Intel Corporation, Santa Clara, CA, United States

A R T I C L E   I N F O

Article history:
Available online 27 February 2009

Keywords:
Optimistic and pessimistic STM
Inplace memory update/write buffering
Eager locking
Discrete-time Markov chains
Analytical modeling

A B S T R A C T

Analytic models based on discrete-time Markov chains (DTMC) are proposed to assess the algorithmic performance of Software Transactional Memory (TM) systems. Base STM variants are compared: optimistic STM with inplace memory updates and write buffering and pessimistic STM. Starting from an absorbing DTMC, closed-form analytic expressions are developed, which are quickly solved iteratively to determine key parameters of the considered STM systems, like the mean number of transaction restarts and the mean transaction length. Since the models reflect complex transactional behavior in terms of read/write locking, data consistency checks and conflict management independent of implementation details, they highlight the algorithmic performance advantages of one system over the other, which – due to their at times small differences – are often blurred by implementation of STM systems and even difficult to discern with statistically significant discrete-event simulations.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

As a parallel programming model for shared-memory chip multi-processors (CMPs), the concept of transactional memory (TM, [1,2]) has received a lot of attention as an alternative to synchronizing parallel applications by means of classical locking mechanisms to coordinate access to shared data. Recently, interest has grown to explore the potential of TM also in the domain of clusters [3–5], where expensive communication messages between the nodes affect the execution behavior of TM.

Writing parallel or distributed multi-threaded programs based on classical locking is a challenging task. Fine-grained locking may become very complex, enhancing the risks of deadlocks or otherwise incorrect program behavior. Coarse-grained locking may result in inefficient code. Moreover, scalability is limited. TM tries to overcome these drawbacks by providing primitives to the programmer to label critical section code with shared memory accesses, so-called transactions, and resolves resulting conflicts between speculatively started concurrent transactions at runtime. Outside of the TM system, transactions (including several read and write operations to transactional memory) appear to be executed atomically, while internally the TM system ensures a consistent view of the concurrent transactions on the transactional data. Non-conflicting transactions may execute unaffectedly until they successfully finish (i.e., commit). Conflicting data accesses are detected and resolved in different ways ranging from delaying to aborting and restarting one or more transactions.

Many different mechanisms for meta-data organization, conflict detection, contention management and consistency check policy have been proposed and can be combined to various different TM systems. Some variants have been implemented by the research community in order to better understand TM behavior – either in software [6–8], in hardware [1,9–13] or in a hybrid way [14]. Apart from shared-memory parallel architectures, few implementations for clusters also exist [3,4]. Surely, these implementations are very important triggers for pushing forward TM advancement. We believe, however, that they should be complemented with other techniques that
facilitate the decision making in the multitude of TM design choices [2]. Implementation-specific features often make it difficult to compare results across different platforms. Therefore, we propose to abstract away TM implementation details in order to focus on modeling the algorithmic aspects of TM. To this end, we develop performance models of different TM algorithms on the basis of discrete-time Markov chains (DTMCs, see e.g., [15]), whose current states encode the progress of a representative transaction in the system, henceforth called tagged transaction. As this tagged transaction executes in the system, it is influenced by other concurrent transactions via aggregate parameters, which characterize the system state in terms of meta-data like read and write set sizes. As a result, performance characteristics, like the mean number of restarts of a transaction or the mean total number of read/write requests, are obtained very efficiently in a fixed-point iteration over a set of algebraic equations.

This paper presents a novel framework for modeling TM behavior and to reason about the performance of different TM design choices. The methodology may be adapted for different settings. In this paper, we focus on TM algorithms prevalently applied in Software TM (STM) systems, which avoid most of the HTM design complexity and may therefore be considered a more attractive alternative. Three fundamental variants of STM

- pessimistic STM;
- optimistic STM with inplace memory updates;
- optimistic STM with write buffering and encounter-time locking;

are investigated in this paper. Their differences are outlined in Section 2.

1.1. Related work

In this paper, we pursue a systematic comparison of various design decisions in STM by means of analytic models. We refine our model for optimistic STM with inplace memory updates presented in [16] by defining the DTMC states such that repeated read and write operations to the same data within a transaction are tolerated. The models for the other optimistic variant and for pessimistic STM are proposed here for the first time. Other analytic models of performance for transactional processing have been proposed in the literature [17–20], but are mostly devoted to database transaction systems and take a very approximative approach to describe the specific state of a transaction. Important details like the number of data held in shared/exclusive state can be omitted for database systems, but are determinant to STM performance. The execution model in [21] does not consider transactional memory execution, but rather inter-dependencies between sets of potential parallel tasks. As a result, the execution model, which is limited to optimistic concurrency, cannot provide insight into the dynamics of TM algorithms.

Apart from the independence of implementation details, our analytic performance models have another advantage over experiments on implemented systems – and also over simulation studies: results and trends are usually obtained much faster and, moreover, allow to identify qualitative performance differences more reliably. Some design decisions may only imply minor performance differences, which may be hard to discern with statistic significance by simulation or implementation experiments. We have observed the latter in own STM simulation studies [22]. A systematic comparison, however for HTM, has also been attempted in the simulation study [23], which (tailored to the SPARC architecture) evaluates important HTM tradeoffs for (idealized) base HTM systems. Performance studies on implemented TM systems (with specific sets of TM features) have been conducted for both HTM [9,13,11,24,25] and STM [26–29]. Different design alternatives are compared in [25,27,28,8]. Closest to the base STM algorithms considered in this paper are [27] (with eager conflict detection with either optimistic or pessimistic concurrency control for reads implemented in Happyville) and [8,28], which compare optimistic STM with write buffering and inplace memory update. Still, differences do exist in details related to conflict detection, version management and conflict resolution, e.g., regarding a global version lock.

The rest of the paper is organized as follows: in Section 2, we describe the base STM systems, for which we construct the performance models in Section 3. In this section, the closed-form expressions to be used in the evaluation are elaborated. The corresponding algorithms are compiled in Section 4. Section 5 presents numerical results, which show the behavior of critical STM performance measures. The analytical data is validated against simulation results. Finally, we conclude in Section 6.

2. Considered STM variants

According to the categorization given in [29], the common features of the STM systems discussed in this paper can be characterized as obstruction-free and object-based with per-object meta-data. However, regarding contention management and validation strategy, the three STM systems studied here differ fundamentally. Diverse names can be found in the literature for the two optimistic STM variants mentioned above: write buffering is sometimes referred to as write-back or lazy version management, memory inplace update as write-through, eager version management or undo logging. Since these terms still leave several design decisions open, e.g., with respect to locking and versioning rules, we clarify in this section upon which operational rules the DTMC-based models of the next section will be constructed.

For both optimistic and pessimistic STM systems, concurrent transactions may or may not be granted access to some transactional data they want to read or write depending on whether a conflict is detected or not. Typically, such situations are described by means of locks, where we distinguish between read and write locks. In this paper, each data object may be locked separately. In principle, locks may be acquired any time between the initial access to the data (or its meta-data descriptor) and the release of all locks the transaction holds (either at commit or abort time). Except when stated otherwise, we assume that locks are acquired with the initial access, i.e., at open or
encounter-time, which is also called eager locking. With the exception of speculative readers (see below for optimistic STM), a write lock usually corresponds to exclusive access to transactional data. Read locks are commonly associated with shared access to transactional data, at least tolerating parallel read accesses. For optimistic STM, write-after-read (WAR) dependencies may occur across transactions. Then different transactions may hold read and write locks on the same transactional data. Strictly speaking, read locks in the context of optimistic STM should be replaced more appropriately by recorded reads (in a read set). However, we stick to the terminology above to ease the flow reading (without loss of correctness).

Different STM systems require a different degree of visibility of locks distributed for transactional data, especially when conflicts may be resolved by one transaction actively aborting another transaction. In this paper, we do not consider such active aborts and therefore locks cannot be stolen by one transaction from other ones. Thus, it suffices that the meta-data for transactional data indicates if the data is write-locked or not (and additionally for pessimistic STM, how many transactions currently have a read lock on the data). Otherwise, transactions are invisible to other transactions. In particular, an accessing transaction need not know which other transactions hold locks on a specific transactional data. The data structures required for corresponding meta-data organization, e.g., in terms of transaction records and transaction descriptors, is beyond the scope of this paper and is described elsewhere (see e.g., [29]). The possible validation strategies required for optimistic STM, including the versioning, will be discussed in the respective subsections.

2.1. Pessimistic STM

Pessimistic STM pursues a conservative approach in that it prevents transactions from entering an inconsistent state a priori. Therefore, validation checks to detect such inconsistencies become unnecessary.

To achieve this, write and read locks on the same data by different transactions are not granted. More precisely, a write lock on some data item is only granted to a transaction,

- if this data item is not locked by any other transaction, neither by a write lock nor by read lock(s).

The above definition implies that if the accessing transaction has a unique read lock on the data item, this read lock may be converted to a (unique) write lock.

On the contrary, a read lock on some data item is only granted to a transaction,

- if this data item is not write-locked by any other transaction. However, the data item may be locked by any number of concurrently active read locks.

Whenever a transaction encounters a situation in which one of its read/write lock requests is not granted, it aborts and restarts. When the transaction aborts or finishes successfully, it releases all collected read and write locks.

2.2. Optimistic STM

Optimistic STM provides more opportunities for parallel execution than pessimistic STM at the risk that transactions are aborted/restarted at a later instant (leading to more wasted operations). The key difference is that now a write lock on some data item is only granted to a transaction,

- if this data item is not write-locked by any other transaction.

Obviously, other transactions may hold a read lock on the data item at the instant the write access is granted, which leads to the above-mentioned WAR dependency with the involved transactions being in a potential conflict. Under certain circumstances, typically if a reading transaction finishes before the writing transaction, this potential conflict does not impact the behavior of neither transaction. They may finish successfully after all.

Admitting such speculative readers requires a versioning mechanism to indicate to readers that transactional data has been written upon and along with it a validation procedure to detect states of data inconsistency (e.g., to detect that a reading transaction is working with outdated transactional data). We assume that such a validation check based on version numbers of the transactional data is performed right at the end of every read lock acquisition attempt and at the final commit operation.

Versioning essentially means to associate a global counter (visible to any transaction) with each transactional data object. Write operations eventually increment these counters to indicate that the transactional data has been modified. When a transaction first reads transactional data and successfully obtains a read lock, it records the value of its counter (i.e., the current global version number of the transactional data) locally. In every validation procedure, the transaction compares all locally stored version numbers with the respective current global version numbers. If any global version number has been incremented by another transaction in the meantime (i.e., any locally stored version number is smaller than the corresponding global version number), the validation check fails and the validating transaction aborts and restarts. Otherwise the transaction may continue with the next operation.

The point in time when global version numbers are to be incremented also depends on the instant when changes of the write operations are actually performed to the transactional data. In this paper, we distinguish two cases: inplace memory updates and write buffering.

- With inplace memory updates, the transaction directly manipulates the transactional data at lock acquisition time, but saves its original value in an undo log in case the transaction needs to be rolled back due to an abort. In this case, the logged original value is written back to the global memory location of the transactional data.
- With write buffering, transactions manipulate local copies of the transactional data until the changes are made visible by a successful commit operation.

Transactions must keep track of which read and write locks they have obtained since their (re)start. We also refer to these two sets of locks as the read set and write set, respectively. The elements in the write set indicate which version numbers will have to be incremented by the transaction, while the version numbers of the elements in the read set have to be compared during validation.

### 2.2.1. Optimistic STM with inplace memory updates

One might suggest to increase the version numbers already at encounter-time, i.e., when the write locks are acquired (eager locking) and the global memory location is modified. However, speculative readers in WAR dependencies (i.e., the transactions with read locks on the data) would be forced to abort earlier than necessary (due to incremental validation). Therefore, with inplace memory updates for writes, a transaction increments the version numbers of transactional data in its write set only later, namely both at commit and abort time. (More precisely, at commit time means at the end of a successful commit operation, which in turn means after a successful validation of the read set.)

Since the newly acquired write lock blocks novel accesses and repeated read requests from other transactions, written data becomes effectively visible only after a successful commit.

Incremental validation checks on the read-set data (i.e., on every read request) ensure a locally consistent view of the transactional data from its (re)start until commit, even though the global memory location might have been manipulated.

Speculative readers in WAR dependencies will not abort due to the considered WAR data, as long as they do not attempt to write access nor to (re)read access this (now otherwise write-locked) data and do finish before the transaction with the recently acquired write lock finishes or aborts.

### 2.2.2. Optimistic STM with write buffering and eager locking

With local buffers for the write-set data, global memory (apart from eager locking) is left unmanipulated in case a transaction has to abort. The global memory location only needs to be modified (i.e., the local values are copied to the global memory locations), if a transaction successfully finishes making the local changes permanent and visible to other transactions. At the same time (a compare-and-swap instruction ensures atomicity), the transaction increments the version numbers of these data items. Therefore, with write buffering, version numbers are only incremented at commit time, i.e., at the end of the lifetime of the transaction.

In WAR dependencies, speculative readers will not abort due to the considered WAR data, as long as they do not attempt to write access this data and do finish before the transaction with the recently acquired write lock finishes. While repeated read requests may be tolerated (due to the yet unmodified global memory location), transactions with a novel read request to the data will, however, be aborted due to the write lock. An aborted and restarted writing transaction does not affect the reading transactions, because an abort operation does not increment the version numbers of the write-locked data.

While concurrent transactions may operate on different values for the transactional data (due to local copies), (incremental) validation checks guarantee that each transaction has a consistent view of the transactional data. For instance, data that is only read will always have the same value for this transaction between (re)start and commit; otherwise the transaction will be aborted.

The latter also holds for inplace memory update. However, increasing version numbers only at commit time constitutes the major difference of optimistic STM with write buffering as compared to STM with inplace memory updates. The less frequent versioning updates may cause speculative readers to be aborted less often, which implies a higher degree of concurrency in WAR dependencies. Generally, the interaction of different mechanisms makes it very difficult to predict which STM variant will perform better in which situations, especially with an increasing share of write operations.

### 3. The DTMC-based models

Instead of modeling explicitly all transactions in their concurrent behavior, our model characterizes the representative behavior of a single so-called tagged transaction. The impact of the other transactions will be captured by appropriately computing the parameters of the single-transaction model. To some extent, this approach assumes that all transactions have a similar probabilistic behavior.

Since we want to study performance measures independently of the specific timing between lock requests, we consider the behavior of the tagged transaction at the instants of lock requests. In this paper and opposed to [16], we propose to model this behavior as an (absorbing) embedded Markov chain, whose states are enumerated according to the current number of read/write operations that have been successfully performed, i.e., the current progress of the transaction.

Transition probabilities of the absorbing discrete-time Markov chain (DTMC) will depend on how many read and write locks on the transactional data are currently held by the transactions. This information can in turn be inferred from the DTMC. As a consequence, a fixed-point iteration scheme arises.

All models are determined by only four key input parameters $L, N, k$ and $l_w$:

- Integer $L$ denotes the number of transactional data items in the system, i.e., the amount of data lockable by read and write locks. In STAMP programs [30], which serve as STM benchmark applications, $L$ ranges between 40,000 ($k$means) and 11 millions ($Yada$).
- Integer $N$ denotes the number of concurrent transactions competing for the transactional data. We assume that there are always $N$ active transactions on dedicated
cores, i.e., \( N \) is constant.\(^1\) This number increases as the hardware technology advances.

- The tagged transaction has to perform \( k \) subsequent read and write operations (possibly repeatedly to the same data) successfully in order to finish. Our own measurements on STAMP programs [30] resulted in values for \( k \) between 6 and 141.
- We do not assume any particular ordering of the read and write operations in a transaction. Instead, any lock request is issued as a write access with probability \( l_w \) (and as a read access with probability \( l_r = 1 - l_w \)). All transactional data objects are equally popular, i.e., have the same probability of being accessed. Transactions of the benchmark applications [30] read more frequently than they write, i.e., \( l_w \in [0.05, 0.49] \).

In the following, we comment on the modeled assumptions above. The common case transactional behavior has been reported in [31] to have little variability in \( k \). This is consistent with our own experimental observations using the Happyville STM system [27]. With the exceptions of only a few cases (e.g., Genome), most applications collected from STAMP [30] have indeed exhibited very little variation in \( k \). To account for different classes of transactions of different sizes (i.e., different values of \( k \)), our framework may be extended according to ideas in [16]. In this paper, we consider saturation conditions (i.e., \( N \) active transactions on \( N \) cores) under artificial workloads (see last assumption). While the presented framework may be adapted to loosen these assumptions, they are justified by our interest in the relative performance of the STM variants (with respect to each other) independent of specific applications. If unsaturated conditions were to be considered instead, then we would have expected a (rather uniform) decrease in conflict rates and, as a result, only little insight into the behavior of the different STM variants. Also note that access patterns to transactional behavior may vary significantly from application program to application program. We expect that possible modifications in the assumptions will not invert the performance behavior of the STM variants, but rather only emphasize or mitigate certain effects, which can already be observed under the stated assumptions. For instance, the number \( L \) of lockable transactional data could be much smaller with some applications, causing more conflicts to occur under certain conditions where the value of \( k \) or \( N \) is larger. We expect this type of scenario to sharpen some of the observations made in this paper.

### 3.1. Transaction behavior

Let us observe the execution of a tagged transaction \( T^{(j)} (1 \leq j \leq N) \) at the epochs when a read or write request is issued. (In the following, we omit the superscript \( j \) to simplify notation.) With each successful read or write operation on transactional data, transaction \( T \) advances to the next epoch, i.e., we define state \( i \) of transaction \( T \) as the state in which \( T \) has performed a sequence of \( i \) successful operations.

Depending on the current state \( i \) of the tagged transaction and the behavior of the other transactions, the current read/write request will be successful with (non-zero) probability \( q_i \), i.e., with this non-conflict probability the tagged transaction moves from state \( i \) to state \( i + 1 \) \((0 \leq i < k)\). With the complementary probability \( 1 - q_i \), a conflict occurs and the transaction must restart in state 0, i.e., resumes execution from state 0.

Fig. 1 depicts this behavior in form of an absorbing discrete-time Markov chain (DTMC). Here, we assume that when the transaction is aborted, it restarts immediately. At stage \( k \), a commit operation (which includes the validation of the read set for optimistic STM) succeeds with probability \( q_k \) and fails with \( 1 - q_k \), possibly depending on factors like the size of the read and write sets of the tagged transaction and the writing behavior of the other transactions. The transaction completes when it reaches the absorbing state \( k + 1 \). For pessimistic STM, commit operations are always successful, i.e., \( q_k = 1 \).

In state \( i \) \((0 \leq i < k)\), a transaction may hold fewer than \( i \) locks due to repeated read and write accesses to the same data. The number of lock requests to distinct data corresponds to the number of held (read/write) locks. Since a transaction accesses data equiprobabilistically, we can compute the mean number of distinct locks held in state \( i \) as

\[
 n_i^{(i)} = L \left( 1 - \left(1 - \frac{1}{L} \right)^i \right)
\]

according to the general birthday problem [32]. On average, approximately \( n_{rw}^{(i)} = l_w n_i^{(i)} \) are write locks and \( n_{rw}^{(i)} = l_r n_i^{(i)} \) are read locks. We will use this and similar information about the non-tagged transactions below in order to establish expressions for the non-conflict probabilities \( q_i \).

### 3.2. Average number of read and write locks held by an arbitrary transaction

Assuming probabilistically identical behavior\(^2\), we may interpret the DTMC of Fig. 1 as the representative behavior of an arbitrary (not only the tagged) transaction. With the

\(^1\) Otherwise, we might work with an effective number \( N_{eff} \) of concurrent transactions.
\(^2\) Otherwise, we may exploit some experimental proportionality between the average current progress of a transaction and \( k \), the given number of operations (see e.g., related discussions in [16]).
DTMC fully specified, important performance measures, like the mean number $E[S]$ of steps of a transaction before absorption (i.e., all issued lock requests and the commit step before the completion of the transaction), can be computed as outlined below. $E[S]$ counts all locks requests, including those, which might have to be repeated due to a restart of the transaction, plus the final commit step, and thus corresponds to the number of steps of the transient DTMC until absorption. Let us also introduce $E[I]$ as the average current progress of an arbitrary transaction, i.e., the average state number, in which it resides in steady state.

The absorbing DTMC of Fig. 1 consists of $k + 2$ states, where state $k + 1$ is the absorbing state. We denote by $p_i$ ($0 \leq i \leq k$) the probability that – at an arbitrary instant of time before absorption – the DTMC is in state $i$, i.e., the probability that the transaction has successfully processed the first $i$ read and write operations at an arbitrary instant before its completion.

For an absorbing DTMC, $p_i$ may be computed as the ratio of the mean number of visits to state $i$, $E[S]_i$, over the mean number of steps of the transient DTMC until absorption, $E[S]$. $E[S]$ is computed as [33]

$$E[S] = v_0 (I - P)^{-1} e,$$  

(2)

where row vector $v_0$ is the initial probability vector of the transient DTMC with (substochastic) transition probability matrix $P$. Matrix $I$ is the identity matrix of appropriate dimension and column vector $e$ a corresponding vector of ones. For the DTMC of Fig. 1, matrices and vectors are of dimension $k + 1$ and more specifically

$$v_0 = [1 \ 0 \ \ldots \ 0],$$

$$P = \begin{bmatrix}
1 - q_0 & q_0 & 0 & \ldots & 0 \\
1 - q_1 & 0 & q_1 & \ldots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 - q_{k-1} & 0 & \ldots & 0 & q_{k-1} \\
1 - q_k & 0 & \ldots & 0 & 0
\end{bmatrix}.$$  

Evaluating Eq. (2) and using the fact that – with the specific initial probability vector $v_0 - E[S]$ simply equals the $i$th component of the first row of the fundamental matrix $(I - P)^{-1}$, we get

$$E[S] = \frac{1}{\Pi_{j\neq i} q_j} \text{ and } E[S]_i = \frac{1}{\Pi_{j\neq i} q_j}. \quad (4)$$

If all non-conflict probabilities equal $1$, $E[S] = k + 1$ and $E[S]_i = 1$ as expected.

Hence, the probabilities $p_i = \frac{E[S]_i}{E[S]}$ for $i = 0, \ldots, k$ are

$$p_i = \frac{\Pi_{j\neq i} q_j}{\sum_{j=0}^{k-1} \Pi_{j\neq i} q_j}.$$  

(5)

We can now compute the average current progress of a transaction (i.e., the mean number of steps performed by a transaction since its last (re)start), denoted by $E[I]$, as

$$E[I] = \sum_{i=0}^{k} i p_i = \frac{\sum_{i=0}^{k} i \Pi_{j<i} q_j}{\sum_{j=0}^{k} \Pi_{j<i} q_j}. \quad (6)$$

Again using the solution of the well-known general birthday problem, we obtain the following expressions for the mean number of read/write locks held by an arbitrary transaction (on distinct data):

$$E[Q_w] = l_w L \left(1 - \left(1 - \frac{1}{L}\right)^{E[I]} \right), \quad (7)$$

$$E[Q_r] = l_r L \left(1 - \left(1 - \frac{1}{L}\right)^{E[I]} \right). \quad (8)$$

This information of an arbitrary transaction will flow into the specification of the non-conflict probabilities $q_i$ leading to a fixed-point iteration scheme to evaluate the complete model of any STM variant.

3.3. Variant-dependent non-conflict probabilities

In this section, we determine the non-conflict probabilities in response to lock requests, i.e., probabilities that, given a transaction is in state $i$ ($0 \leq i < k$ for the tagged transaction), it does not encounter a conflict upon its $(i + 1)$th lock request and continues executing from state $i + 1$ in the next step. We do this first for pessimistic STM.

3.3.1. Pessimistic STM

In case a transactional read operation is issued, a transaction $T$ moves from state $i$ to next state $i + 1$ in one step, if and only if one of the following two mutually exclusive conditions holds:

**cond-1:** The requested data object is already in the write set of this transaction (i.e., RAW within the tagged transaction)

**cond-2:** The requested data object is not in the write set of any transaction.

For successful progress with a transactional write operation, the following mutually exclusive conditions apply:

**cond-3:** The requested data object is neither in a read set nor in a write set of any transaction.

**cond-4:** The requested data object is already in the write set of the transaction (i.e., WAW within the tagged transaction).

**cond-5:** The requested data object is exclusively read-locked by the transaction (i.e., WAR within the tagged transaction).

Given that a request occurs, it may be either a write request with probability $l_w$, or a read request with probability $l_r$. We can then write $q_i$ as follows

$$q_i = l_w (P_i \{\text{cond} - 1\} + P_i \{\text{cond} - 2\} + l_r (P_i \{\text{cond} - 3\} + P_i \{\text{cond} - 4\} + P_i \{\text{cond} - 5\}), \quad (9)$$

where $0 \leq i < k$ and subscript $i$ denotes that the tagged transaction is in state $i$.

Obviously, the requested data is in the write set of the considered transaction with probability $\frac{l_w}{l_w + l_r}$. Thus, the identical probabilities for conditions 1 and 4 are
\[ P_i(\text{cond} - 1) = P_i(\text{cond} - 4) = \frac{n^{(i)}_{\text{rw}}}{L}. \]  

The requested data is not in any of the write sets with probability 
\[ P_i(\text{cond} - 2) = \frac{L - n^{(i)}_{\text{rw}} - (N - 1)E[Q_w]}{L}. \] Therefore
\[ P_i(\text{cond} - 1) + P_i(\text{cond} - 2) = \frac{L - n^{(i)}_{\text{rw}} - (N - 1)E[Q_w]}{L} + \frac{L - (N - 1)E[Q_w]}{L}. \]

Recall that \( n^{(i)}_{\text{rw}} \) is the average number of write locks held by 
the tagged transaction in state \( i \). Of the \( L \) data objects, to
which the tagged transaction may issue a write request, 
only those will lead to a successful write request that 
are not owned by any of the \( N - 1 \) other transactions (in numbers 
\( L - (N - 1)E[Q_w] \)).

The probability that the tagged transaction in state \( i \) accesses data that is not locked at all is computed as
\[ P_i(\text{cond} - 3) = \frac{L - n^{(i)}_{\text{rw}} - (N - 1)E[Q_w] - L \left( 1 - \frac{1}{L} \right)^{\left( N-1 \right)E[Q_w]+l_i}}{L}. \]

From all data items \( L \), we subtracted all the numbers of write locks held either by the tagged transaction or by the \( N - 1 \) other transactions. When subtracting the corresponding numbers of read locks, we have to take into account that read locks are not exclusive. The solution of the birthday problem delivers the desired expression for the number of distinct read-locked data objects, where 
\( (N - 1)E[l_i] + l_i \) corresponds to the average current number of all successfully performed read operations (with the tagged transaction being in state \( i \)).

A variation of the birthday problem ("no one with the same birthday as you") becomes relevant for 
\( P_i(\text{cond} - 5) \), which is the probability that the tagged transaction in state \( i \) has the only read lock on the requested data:
\[ P_i(\text{cond} - 5) = \frac{n^{(i)}_{\text{rw}}}{L} \left( 1 - \frac{1}{L} \right)^{\left( N-1 \right)E[Q_w]}. \]

The probability of accessing an object already read (WAR) is multiplied with the probability that none of the 
\( (N - 1)E[l_i] \) (possibly repeated) read requests refers to this element.

Combining (10) through (13), we write (9) as follows with \( 0 \leq i < k \):
\[ q_i = l_i \left( \frac{L - (N - 1)E[Q_w]}{L} \right) + l_w \left( \frac{L - (N - 1)E[Q_w] - L \left( 1 - \frac{1}{L} \right)^{\left( N-1 \right)E[Q_w]+l_i}}{L} + n^{(i)}_{\text{rw}} (1 - \frac{1}{L})^{\left( N-1 \right)E[Q_w]} \right). \]

For pessimistic STM, \( q_k = 1 \), i.e., the commit step is always successful and only consists in releasing read and write locks.

\[ 3.3.2. \text{Optimistic STM} \]

Optimistic STM mandates different conditions under which a transaction \( T \) may move from state \( i \) to the next state \( i + 1 \) in one step. The main differences are that
- the additional validation of the read set (via the data version numbers) after a transactional read operation 
does not fail and
- successful write operations are no longer blocked by read locks of other transactions.

Eq. (9) is thus adapted to
\[ q_i = l_i (P_i(\text{cond} - 1) + P_i(\text{cond} - 2)) \cdot P_i(\text{validation successful}) + l_w (P_i(\text{cond} - 1) + P_i(\text{cond} - 2)). \]

where \( 0 \leq i < k \).

\[ 3.3.3. \text{Optimistic STM with inplace memory updates} \]

Successful validation means that the version numbers of all data objects in the current read set have not been increased by other transactions. On average, the tagged transaction holds \( n^{(i)}_{\text{rw}} \) read locks in state \( i \), to which \( N - 1 \) other transactions must not have written after the corresponding read accesses.

For inplace memory updates, since the version number of each data item is incremented once per novel write access - either at commit time or abort time, transactions write to data with the same average rate as they are granted novel write access to them. Validation fails, if any one of the \( N - 1 \) other transactions writes on one of the \( n^{(i)}_{\text{rw}} \) items of data, which happens with probability
\[ 0.5 l_w P_i(\text{cond} - 2) \frac{n^{(i)}_{\text{rw}}}{L - n^{(i)}_{\text{rw}} - (N - 1)E[Q_w]} \times \frac{L - (N - 1)E[Q_w]}{L}. \]

for a single such transaction. After a write access (with probability \( l_w \)), \( P_i(\text{cond} - 2) \) filters out any write accesses to already write-locked data. The factor 0.5 reflects the equiprobabilistic interleaving of the requests issued by the tagged and any other transaction: with probability 0.5, the other transaction does not issue any request before the next request of the tagged transaction. With the frequency of the relevant write accesses being formulated, we have to identify when these write accesses contribute to a validation failure of the tagged transaction. From the perspective of one of the \( N - 1 \) transactions, it cannot write to any data write-locked by others and - since repeated writes do not have an additional impact on the version numbers - the \( E[Q_w] \) write locks of this transaction are also subtracted in the denominator of the first fraction in (16).
The numerator in the last term also subtracts – of all available locks L – the number of all currently held write locks (by both the tagged transaction in state i and the N – 1 other transactions). Thus, the last fraction represents the probability that a requested data item is not covered by a write lock (due to a WAR dependency across transactions) and ensures that this holds for the requested read-set element of the tagged transaction (any of \(n_{qr}^{(i)}\)).

In summary, we approximate \(P_i\{\text{validation successful}\}\) by

\[
P_i\{\text{validation successful}\} = \left(1 - 0.5l_w \frac{L - n_{qw}^{(i)} - (N-1)E[Q_w]}{L}\right) \left(1 - 0.5l_w \frac{n_{qr}^{(i)} - n_{qw}^{(i)} - (N-1)E[Q_w]}{L}\right)^{N-1}.
\]

This expression states that none of the \(N-1\) other transactions has modified a previously unmodified element in the read set of the tagged transaction, since the last request of the tagged transaction.

Inserting (11) and (17), we write (15) as follows with \(0 \leq i < k\):

\[
q_i = l_i \frac{L - (N-1)E[Q_w]}{L} \times \left(1 - 0.5l_w \frac{n_{qr}^{(i)} - n_{qw}^{(i)} - (N-1)E[Q_w]}{L}\right)^{N-1} + l_w \frac{L - (N-1)E[Q_w]}{L}.
\]

With similar arguments, we compute \(q_k\):

\[
q_k = P\{\text{commit validation successful}\} = \left(1 - 0.5l_w \frac{n_{qr}^{(k)} - n_{qw}^{(k)} - (N-1)E[Q_w]}{L}\right)^{N-1}.
\]

3.3.4. Optimistic STM with write buffering and eager locking

Since each transaction actually only writes on the globally visible data with the commit operation, the effective write rate can be approximated with \(1 \frac{1}{E[S]}\). By setting the mean transaction lifetime to \(E[S]\), we assume that delays between requests of a transaction have a unit mean. If we know the probability \(P_{c0}\) that the commit operation of an arbitrary transaction does not affect the read set of the tagged transaction in state \(i\), we can approximate \(P_i\{\text{validation successful}\}\) by

\[
P_i\{\text{validation successful}\} = \left(1 - \frac{1 - P_{c0}}{E[S]}\right)^{N-1}.
\]

Let us now determine the probability \(p_{c0}\): This probability will depend on the size of the write sets of the other transaction (on average \(n_{qw}^{(i)}\)), the size of the read set of the tagged transaction (on average \(n_{qr}^{(i)}\)) and the number of data objects, to which the other transaction may write in its commit step in the first place \((L - (N-2)E[Q_w])\), \(n_{qw}^{(i)} - n_{qr}^{(i)}\) on average. The considered transaction cannot have any data in its write set, which is currently write-locked otherwise. We can also subtract the number of read locks, \(n_{qr}^{(k)}\), held by the considered transaction, since they cover data distinct from those in the write set.

Understood as an urn problem without putting back any of the items in two colors (red/black), we are interested in the probability that in \(n_{qr}^{(i)}\) draws, we pick none of the \(n_{qw}^{(i)}\) red items in a total of \((L - (N-2)E[Q_w])\), \(n_{qw}^{(i)} - n_{qr}^{(i)}\) red and black items. The corresponding probability of the hypergeometric distribution can be computed as

\[
p_{c0} = \frac{\left[ L - (N-2)E[Q_w] - n_{qw}^{(i)}\right]}{[L - (N-2)E[Q_w] - n_{qw}^{(i)} - n_{qr}^{(i)}]}.
\]

where we rounded the entries to reasonable integer values to enable evaluation of the binomial expressions. Instead of subtracting \(n_{qr}^{(i)} + n_{qr}^{(k)}\) in the topmost expression of (21), we subtracted \((1 - (1 - \frac{1}{E[S]})^{k-1})\) in order to eliminate non-distinct read locks of the tagged and the other transaction. Once again, the solution of the general birthday problem delivers this expression. Note that the tagged transaction is in state \(i\), while the other is committing in state \(k\) (as indicated by the superscripts).

Inserting (11) and (20), we write (15) as follows:

\[
q_i = l_i \frac{L - (N-1)E[Q_w]}{L} \left(1 - \frac{1 - P_{c0}}{E[S]}\right)^{N-1} + l_w \frac{L - (N-1)E[Q_w]}{L}.
\]

Analogously, we compute \(q_k\) from

\[
q_k = P\{\text{commit validation successful}\} = \left(1 - \frac{1 - P_{c0}}{E[S]}\right)^{N-1}.
\]
4. Algorithms and performance measures

For each STM variant, a different set of equations has to be solved by fixed-point iteration until a sufficient precision is reached:

<table>
<thead>
<tr>
<th>STM Variant</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pessimistic STM</td>
<td>(14), (1), (7) and (6)</td>
</tr>
<tr>
<td>Optimistic STM (inplace memory update)</td>
<td>(18), (1), (7) (and implicitly (6))</td>
</tr>
<tr>
<td>Optimistic STM (write buffering)</td>
<td>(22), (1), (7), (4), (21) and (23)</td>
</tr>
</tbody>
</table>

In all our experiments with reasonable parameters settings, we have always encountered convergence of this scheme within a few iterations. Regarding optimistic STM with write buffering and eager locking, one needs to iterate over more equations (including $q_k$), since $E[S]$ appears in the formulas of the non-conflict probabilities $q_i$.

Several STM performance measures may be obtained from the DTMC-based models. For instance, the values of the non-conflict probabilities or – as a global characteristic – the mean number of write-locked data objects at an arbitrary time $(NE(Q_{lw}))$, may be of interest. In the next section, we focus on the following key STM performance measures of an arbitrary transaction:

- mean number of restarts: $E[R] = E[S_0] - 1$ (see (5) for $E[S_0]$);
- mean number of steps of a transaction (including all lock requests and the final commit step): $E[S] = \sum_{i=0}^{\infty} \frac{1}{P_i}$;
- mean number of locks held by a transaction: $E[Q] = E[Q_r] + E[Q_{lw}]$.

The parameters $E[R]$ and $E[S]$ should be interpreted together to assess the quality of the conflict management scheme. Obviously, the smaller the mean number of restarts, the better the performance of the STM system. A scheme. Obviously, the smaller the mean number of restarts, the better the performance of the STM system. A

5. Numerical results

In this section, we conduct experiments to investigate the performance differences between the three STM variants. In a first set of experiments, we validate the DTMC-based model with data from a discrete-event simulation. In the second set of experiments, we highlight the use of the analytical model to predict performance behavior for projected input parameters. In all experiments, the input parameters $L$ and $k$ are set to values of a magnitude as they might occur in typical benchmark applications [30]: $L = 1.000.000$ and $k = 100$. Each transaction must perform a sequence of $k$ static operations.

5.1. Model validation for varying write access probability $l_w$

Here, we vary the write access probability $l_w$ between 0 and 1, while the number of parallel threads $N$ is set to $N = 16$. The comparison of analytical and simulation results will show that the DTMC-based models capture very well the trends observed in STM. Especially for larger numbers of $N$ or $k$ and/or often small values for performance measures (like the mean number of restarts), a discrete-event simulation requires relatively long run-times for statistically significant results, whereas the numeric evaluation of the analytic iteration schemes of Section 4 produces results in Maple [34] quasi-immediately after a few iterations (usually much fewer than 10 for convergence of the non-conflict probabilities $q_i$ within an absolute deviation of $10^{-6}$).

As opposed to the DTMC-based model, the simulation model implements $N$ transactions explicitly as concurrent processes, of which each one initially selects a sequence of $k$ read and write accesses according to the probability $l_w$. This sequence is maintained throughout the lifetime of the (simulated) transaction, i.e., after an abort, the transaction retries the identical sequence of operations (unlike the analytical model). Moreover, the simulation selects exponentially distributed delays for read and write operations. By this choice in comparison with the DTMC-based model, we demonstrate that fundamental trends in STM performance do not depend on the specific distribution of durations between requests. Note that – apart from a similar probabilistic behavior – the analytical model makes no assumption about the inter-request distribution. Regarding contention and conflict management, the simulation model implements the logic described in Section 2. More details about the settings for the simulation model can be found in [22]. The simulation experiments have been performed in the tool AnyLogic [35] with a confidence level of 95% and a relative error of 5% (over all shown performance results).

Figs. 2–4 show the behavior of the performance measures $E[R]$, $E[S]$ and $E[Q]$ for the three STM variants as the probability $l_w$ increases. The solid and dashed lines correspond to the analytical results, the dotted ones to simulation results. We omit the confidence intervals in the figures, since they would impair the readability of the graphs. Generally, with an increasing share of write operations, the mean number of restarts and the mean number of requests per transactions increase due to more conflicts. However, the mean number of locks held per transaction, $E[Q_{lw}]$, slightly decreases (note the scaling of the y-axis1), since the restarts cause the transaction to spend relatively more time in states where it holds fewer locks.

With either only read or only write operations ($l_w = 0$ and $l_w = 1$, respectively), the qualitative behavior of the considered STM variants is actually identical. This fact is more accurately captured by the analytical models for $l_w = 1$. For $l_w = 0$ (for both analysis and simulation), no restarts take place and each transaction finishes in exactly 101 steps ($k$ operations plus the commit) leading to a mean
number of held locks of 50. For \( l_w = 1 \), the mean number of restarts, \( E[R] \), grows to a value of around 0.077. As the most conspicuous difference between the analytical and simulation model, we observe that at \( l_w = 1 \) “analytical” transactions finish in slightly less than 105 steps (104.83) and hold around 49.37 locks (on average), while “simulated” transactions finish in slightly more than 105 steps (\( \approx 105.2 \)) and hold around 49.57 locks (on average). In accordance with the interpretation of the performance measures in Section 4, the analytical model tends to cause restarts slightly earlier than the simulation model. However, we emphasize that the differences between analytical and simulation results amount to relative errors of less than 1% and are thus much smaller than the confidence intervals.

The uncertainty due to confidence intervals also poses a problem when differences in simulation results between STM variants are considered. With simulation, the two optimistic STM variants cannot be reliably distinguished (for eager lock acquisition\(^3\)). The analytical curves suggest that STM with inplace memory update performs slightly better especially for smaller and moderate values of \( l_w \). In typical applications, read operations usually occur more frequently than write operations. Though the performance benefit in terms of \( E[R] \) (and the other measures) may be deemed marginal, it becomes relevant in the light of the fact that implementations of inplace memory update reduce the need to copy memory locations to a minimum, when few conflicts arise. Then, the technique of inplace memory updates may clearly outperform write buffering, which copies transactional data to and from local buffers.

Both simulation and analysis show that optimistic STM improves significantly over pessimistic STM between the extremal points, where around \( l_w = 0.6 \) the mean number of restarts per transaction is decreased by around 25%. For larger values of \( l_w \), the curves for pessimistic and optimistic STM converge less smoothly than for smaller values of \( l_w \). When the considered performance measures of optimistic and pessimistic STM are equal (i.e., for \( l_w = 0 \) and \( l_w = 1 \)), pessimistic STM has advantages, since no validation checks are needed.

Generally – and especially considering the fundamental difference between the DTMC-based approach and the simulation model, the analytic model captures the trends very well for all three measures and produces results quasi-immediately. To produce a single curve with the simulation tool AnyLogic took us around two hours on a standard PC (Intel Pentium 4 CPU with 3.4 GHz and 1 GB RAM). In particular, the DTMC-based model helps in arguing about performance differences between STM variants. As compared with usually overparameterized simulation models (see timing behavior, etc.), the analytic model which is based on few input parameters provides a better understanding of the STM algorithms. The observations of this section extend to other experiments with different values for \( L, N, k \). As indicated in [16] (for optimistic STM with inplace memory update), the probabilistic conflict resolution of the analytic models works better for smaller ratios \( \frac{kN}{L} \) (at least below 1). In practical applications, this is usually the case.

\(^3\) Note that optimistic STM with write buffering may be (and is often) combined with lazy lock acquisition, which improves the performance of this variant.
5.2. Performance trends for increasing parallelism

After years of increasing clock frequency of CPUs, hardware engineers (continue to) enhance computational power by raising the number of cores in CMPs. With respect to STM performance, it is important to identify which variants cope best with the growing parallelism that will soon reach hundreds of cores. For such scenarios, implemented STM systems do not exist yet and simulation is inefficient. However, our analytical models give clues on these performance trends at low computational cost.

In the experiments of this section, we fix $l_w$ to $l_w = 0.3$ (besides $L = 1.000.000$ and $k = 100$) and double the number of cores $N$ starting from $N = 16$ until $N = 512$. Figs. 5 and 6 show the analytical trends for the mean number of restarts $E[R]$ and the mean number of steps $E[S]$ per transaction, respectively, for the projected values of $N$. For $N = 512$ and under the considered workload, all STM variants would cause an arbitrary transaction to restart at least once (on average). All curves suggest an exponential increase. While the two optimistic STM variants show similar performance, the gap between optimistic STM and pessimistic STM widens. The relative error of $E[R]$ for pessimistic STM with respect to optimistic STM with inplace memory update reaches a maximum at $N = 512$ (45.06% as compared to 36.35% for $N = 16$). Optimistic STM with inplace memory update appears to perform consistently better than optimistic STM with write buffering, whose relative deviation actually decreases from 17.45% (for $N = 16$) to 3.2% (for $N = 512$). This converging performance can also be observed for $E[S]$ in Fig. 6. Generally, for $E[S]$ the performance differences are less pronounced and amount to a relative deviation of "only" 10.6% for pessimistic STM for $N = 512$. Still, it should be clear that these relative deviations in terms of $E[S]$ directly translate to system throughputs in transactions per second, where a 10% gain is considered very significant.

6. Conclusions

We proposed an analytical framework for modeling diverse STM variants based on DTMCs. In particular, this paper discussed pessimistic STM and optimistic STM with eager locking, in the latter case either with write buffering or inplace memory updates. A comparison of the results with simulation models demonstrated both the accuracy and the capability of the analytical models to reveal the trend behavior of the considered STM variants as well as their relative performance to each other.

Optimizing transactional memory systems is a very challenging task, as the complex parameter interactions do not facilitate the understanding of STM systems. The DTMC-based models allow to quickly analyze the impact of the most relevant input parameters on system performance and to compare results consistently across various fundamental design decisions. Thus, the analytical models contribute invaluable insight in addition to more detailed simulation models and prototype STM implementations. Both simulation and implementation may suffer from long run-times and related issues of statistical significance. More importantly, the large number of involved parameters (e.g., related to timing, benchmarks, etc.) may block the view on the algorithmic behavior of STM. In this respect, a generic artificial workload, as applied in this paper, seems most appropriate for a systematic analysis of STM variants.

The tagged-transaction modeling approach may be extended in different ways: waiting times before and after an abort may be introduced at the expense of additional states in the DTMC. Different behavior of a tagged transaction class and other transactions classes may be incorporated by means of class-dependent write access probabilities $l_w$. Generally, the framework may be adapted to other STM schemes, like optimistic STM with write buffering and lazy locking.

References

Gilles Pokam is a senior research scientist in the Microprocessor Technology Lab at Intel Corporation, working on diverse topics related to multi-core platforms such as architecture support and software for concurrent programming models, performance evaluation and simulation, and architecture support and software for programmer productivity. He received a MS in computer engineering from the Technical University of Berlin, Germany, and a PhD in computer science from the University of Rennes I, France, in 2004. Before joining Intel in 2007, he was a research fellow at the IBM T.J. Watson Research Center in NY where he worked on multi-threading technology and transactional memory for multi-core architectures. Prior to that, he was a postdoctoral research scientist in the Computer Architecture Lab at the University of California, San Diego. He is the recipient of the IEEE MICRO Top Picks Award 2006 recognizing the most significant papers in computer architecture. He is member of IEEE and ACM.