Modeling Software Transactional Memory with AnyLogic

Armin Heindl
Department of Computer Science
University of Erlangen-Nuremberg
Erlangen, Germany
Armin.Heindl@informatik.uni-erlangen.de

Gilles Pokam
Microprocessor Technology Lab
Intel Corporation
Santa Clara, CA, USA
gilles.a.pokam@intel.com

ABSTRACT

A flexible simulation model is presented to study different variants of software transactional memory (STM), like pessimistic STM or optimistic STM either with inplace memory updates or write buffering.

The dynamic behavior of transactions is encoded in timed statecharts as provided by the simulation tool AnyLogic in its implementation of real-time UML. Their graphical representation helps to convey the key design issues of the simulation model within this publication. Statistically significant numeric results for varying parameters, like number of threads, number of transactional operations, number of transactional data objects, are obtained efficiently as part of a Parameter Variation Experiment.

Categories and Subject Descriptors

C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Multiprocessors*; C.4 [**Computer Systems Organization**]: Performance of Systems

1. INTRODUCTION

With the increased computational power of multi-core processors, the concept of transactional memory (TM) has received a lot of attention as an alternative to traditional lock-based concurrent programming. Whereas the latter relies on the skills of the application programmer to write correct code that exploits the computational power efficiently, a TM system provides primitives to the programmer to label critical code with external memory accesses, so-called transactions, and resolves resulting conflicts between concurrent transactions at run-time. Thus, drawbacks of lock-based programming may be avoided, namely

- the inefficiency of too coarse-grained locks, which prevent the execution of concurrent threads, when they actually could be executed in parallel, and
- the enhanced programming complexity of too fine-grained locks, which increases the likelihood of deadlocks or other incorrect program behavior.

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 deals with several transactions at the same time in a consistent way. This includes ensuring a consistent view of the transactions on the transactional data and the detection and resolution of conflicting data accesses by the transactions. Incompatible actions of two concurrent transactions will cause one to be stalled [18] or eventually aborted [14] and restarted, which eliminates the problems mentioned above for fine-grained locking. On the other hand, non-conflicting transactions may execute unaffectedly until they successfully finish (i.e., commit), which accounts for the potentially higher efficiency of TM systems.

Many different TM systems have been proposed (and also implemented) with different mechanisms for meta-data organization, conflict detection, contention management and consistency check policy [15]. The primitives needed to support these tasks can be implemented in software [13, 22, 21], in hardware [14, 11, 5, 1, 20, 18] or a combination of both, i.e. hybrid TM [7]. Software TM (STM) can avoid most of the HTM design complexity and may therefore be considered a more attractive alternative. Still, various tradeoffs exist between the mentioned TM tasks to be implemented in software and need to be taken into account in order to provide a reasonable performance. For instance, when should transactional data which is write-accessed actually be updated? Does updating at acquire time (inplace memory updates with potential roll-back) outperform updates at commit time (write buffering)? As the number of design parameters grows, the design decisions become more and more challenging.

In this paper, we present a simulation model to explore this design space. Most of the work on the performance analysis of transactions is dedicated to database systems (see e.g., [10, 23]). Aspects related to the specific state of a transaction, like the amount of data held in shared/exclusive state, are often omitted in database studies, but are determinant for TM system performance. Compared with analytical models [10, 12] for TM, the presented simulation model captures more details and allows to validate assumptions employed in more abstract models. Compared with experimental system implementations, the simulation model may more easily be extended by alternative or additional system features and is usually evaluated more efficiently. Moreover, its presentation in form of real-time UML statecharts (as re-

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5581
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5581

alized in the simulation tool AnyLogic [6]) provides insight into the dynamic behavior of the model. Together with statistically significant numerical results, this establishes trust with the simulation model.

The model distinguishes three fundamental concepts of STM:

- pessimistic STM, where an ongoing write access to transactional data does not only preclude other write and read accesses, but (potentially concurrent) read accesses also preclude write accesses
- optimistic STM with write buffering, where write accesses may preempt ongoing read accesses and locally saved changes due to write operations are only made visible to other transactions at commit time of the writing transaction
- optimistic STM with inplace memory updates, where write accesses may preempt ongoing read accesses and changes due to write operations are visible already at acquire time, but might have to be rolled back in case the writing transaction aborts

Related Work

Many performance studies on TM consider implemented systems with a specific set of TM features (e.g., [11, 18, 1, 2] for HTM and [8, 19, 9, 16] for STM). Different TM implementations are also compared. As one example for HTM systems, we mention [4], which compares the proposed HTM design with a conventional HTM system with eager or lazy conflict detection. Some STM systems also allow the evaluation of different design alternatives within their restricted set of realized TM features. *Happyville* [19] provides an eager conflict detection scheme with either optimistic or pessimistic concurrency control for reads. Optimistic STM with write buffering and inplace memory update is compared in [21, 9], however, – as opposed to this paper – on implemented systems with their intrinsic implementation-dependent effects and based on traces for recorded workloads. Differences also exist in details related to conflict detection, version management and conflict resolution, e.g., regarding a global version lock.

In this paper, we pursue a first step to a systematic comparison of the various design decisions in STM (in the simulation tool AnyLogic). For HTM systems, which differ substantially from STM systems (e.g., versioning), such a systematic comparison has been attempted in [3]. A simulation study (tailored to the SPARC architecture) evaluates important HTM tradeoffs for (idealized) base HTM systems and – based on performance results and common "pathological" program execution behaviors – suggests system refinements. Our methodology is different in that we propose to abstract away any implementation detail and focus on the algorithmic performance of the various TM designs. This principle approach is shared with [24]: the execution model therein, however, does not consider transactional memory execution, but rather inter-dependencies between sets of potential parallel tasks. As a result, the execution model, which moreover is limited to optimistic concurrency, cannot provide insight into the dynamics of TM algorithms.

Finally, we point out that in other publications, write buffering is sometimes referred to as write-back or lazy version management, and inplace memory update as write-through, eager version management or undo logging.

In Section 2, we describe the different base STM systems along with the STM primitives that are implemented in the simulation model, which is presented in Section 3. The numerical results of Section 4 compare the three STM variants. Finally, we conclude in Section 5.

2. CONSIDERED STM VARIANTS

Different design decisions, e.g., for meta-data organization, conflict detection, contention management and validation, lead to various STM systems [15]. According to the categorization given in [16], 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.

Generally, for all 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. 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 time, also called eager acquisition. In case a write lock is successfully acquired, the respective transaction is said to *hold a write lock on the data* or equivalently to *own* the data. 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. We will see that for optimistic STM so-called write-after-read (WAR) situations may occur, in which different transactions may hold read and write locks on the same transactional data.

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 a 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., [16]). 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.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5581
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5581

To achieve this, (potentially) conflicting write and read locks on the same data 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 locked via a write lock 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 (possibly also after some stalling), it aborts and restarts. When the transaction aborts or finishes successfully, it releases all collected read and write locks.

The asymmetry in these lock acquisition rules obviously favors read operations over write operations and therefore pessimistic STM is expected to perform better with applications requiring fewer write operations. In these cases, the efficiency is particularly improved by the fact that no validation checks - which are usually associated with read accesses only – are needed. Logically, it does not matter for pessimistic STM, when (e.g., at open time or at commit time) the changes of the write operations are actually performed to the transactional data.

On the down side, excluding the parallel execution of potentially conflicting transactions, which, however, might finish successfully after all due to their timing, ignores a possible performance improvement.

2.2 Optimistic STM

Optimistic STM provides more opportunities for parallel execution than pessimistic at the risk of more aborted/restarted transactions. 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 locked *via a write lock* 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 situation with both transactions being in a potential conflict. Under certain circumstances, typically if the reading transaction finishes before the writing transaction, this potential conflict does not impact the behavior of the two transactions. Both 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. (Naturally, such an *incremental validation* approach may be loosened up to performing the validation check only at commit time.)

Versioning essentially means to associate a global counter (visible to any transaction) with each transactional data. 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.

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: write buffering and inplace memory updates.

- With write buffering, transactions manipulate local copies of the transactional data until the changes are made visible by a successful commit operation.

- With inplace memory updates, the transactions directly manipulate the transactional data at lock acquisition time, but save its original value in an undo log in case the transaction needs to be rolled back due to an abort. At abort, the logged original value is written back to the global memory location of the transactional data.

Transactions must keep track of which read and write locks they have obtained in the course of their lifetime. 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 write buffering*

Only when the transaction can finish successfully (indicated by a successful validation of the read set in the commit operation), it copies the local values of the transactional data in its write set to the global memory locations thus making the changes 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., more precisely at the end of a successful commit operation and thus at the end of the lifetime of the transaction.

Let us have a look at the implications for WAR situations on some transactional data: The reading transactions (i.e., the transactions with read locks on the data) 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. A repeated read request on the same data may be granted despite this write lock by another transaction, since the reading transaction already has the read lock and the data has not yet been modified visibly.

Other 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 from its (re)start until commit. For instance, data that is only read will always have the same value for a transaction in this period; otherwise the transaction will be aborted.

2.2.2 *Optimistic STM with inplace memory updates*

With inplace memory updates for writes, where global transactional memory locations are already modified at write lock acquisition time, a transaction increments the version numbers of transactional data in its write set both at commit and abort time. (Instead, version numbers might be incremented at write lock acquisition time only, but this would essentially mean that writing transactions preempt reading transactions immediately reducing the potential for parallel executions.)

In principle (i.e., for ideal systems), optimistic STM with inplace memory updates might also operate without incrementing version numbers at abort time. However, STM implementations require incrementing version numbers at aborts, since the global memory locations of transactional data have actually been modified with the inplace memory update (irrespective of the roll-back), which should be indicated to other transactions. In effect, the versioning of optimistic STM with inplace memory updates forces speculative readers to be aborted more frequently (than with write buffering), which can be considered as favoring restarted writing transactions.

As a consequence, this behavior implies a lower degree of concurrency in WAR situations. The reading transactions will not abort due to the considered WAR data, as long as they do not attempt to write access *nor to read access* this data and do finish before the transaction with the recently acquired write lock finishes *or aborts*.

As for optimistic STM with write buffering, validation checks ensure a locally consistent view of the transactional data, even though inplace memory updates are visible to other transactions in principle. But since the write lock acquired at open time blocks novel access requests from other transactions and since repeated read accesses by speculative readers will lead to their abort, written data becomes effectively visible only after a successful commit.

In summary, increasing version numbers at abort and aborting speculative readers with repeated read accesses to WAR data constitute the major differences of optimistic STM with inplace memory updates as compared with optimistic STM with write buffering.

2.3 Comparison of STM variants

The interaction of different mechanisms makes it very difficult to predict which STM variant will perform better in which situations. The general considerations of this section are intended to motivate the presented STM variants and deepen the understanding of the mechanisms at work. Obviously, TM systems can exploit the available degree of concurrency best, if only few transactions must be aborted. This corresponds to applications with limited potential for conflicts.

For example, if all transactions consist of read operations only, all discussed STM variants should exhibit the same qualitative performance – with quantitative benefits for pessimistic STM due to saved validations. An increasing number of write accesses increases the potential of conflicts. With few write operations (and thus few conflicts), optimistic STM should perform better than pessimistic STM, since in the latter case write lock acquisition is treated more conservatively and leads to more frequent aborts. Speculative readers in optimistic STM increase the opportunity for parallel execution.

With few aborts, the variants of optimistic STM behave very similarly qualitatively. Inplace memory updates should then benefit from the more efficient implementation of write operations. With increasing number of conflicts, the versioning of optimistic STM with inplace memory updates produces more aborts so that write buffering may eventually perform better. Aborted speculative readers, as any restarted transaction, waste computational resources, especially if these transactions comprise a large number of read/write operations.

It should be clear that – with increasing potential of conflicts – it becomes more and more difficult to argue about the involved tradeoffs. Dedicated numerical experiments with the simulation models of the next section help to determine turning points in performance behavior.

3. THE SIMULATION MODEL

We have developed a common simulation model for the three discussed variants of STM. Apart from many other input parameter options, a graphical user interface allows the user to select between pessimistic STM and optimistic STM, and in case of the latter choice, between *inplace memory update* and *write buffering*. Integrating the behavior of the three STM variants into the same statechart lets the model appear more complex than necessary for a single specific variant, but this drawback is more than compensated by the fact that identical code can be reused as often as possible with less maintenance effort.

The model has been built in the high-level simulation tool AnyLogic (version 6.2.2, [6]) using its proprietary implementation of real-time UML. For discrete-event simulations, AnyLogic combines composite structure diagrams for modeling the system architecture and statecharts for modeling the dynamic behavior of the components, called active objects. Active objects may communicate via message passing (of which we make no use in the simulation model presented here) or global variables. In our context, each thread in which transactions may be processed corresponds to a (replicated) active object, while the transactional data represent global variables. State changes in the statecharts are triggered by (timeout/condition/message) events, which may be controlled by guards, and associated with actions, whose code is given in the programming language Java.

3.1 Basic assumptions and data structures

In the current version of the simulation model, we do not consider meaningful read and write operations for the transactions, i.e., the specific values to be read and written are of no interest and, in fact, not necessary for a general performance comparison of the STM variants. Meta-data

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5581
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5581

structures, like transaction records and descriptors, are only included in a minimalistic way, i.e., as needed in order to reflect the behavior of the STM variants.

Thus, the transactional data are viewed as L homogeneous data items only identified by an identification number, `id`. The Java class `DataUnit` for the data items also contains meta-data variables to record the current version number, `verNumber`, the current number of concurrent read accesses, `currAccess` (needed only for pessimistic STM, otherwise helpful for debugging), and a boolean variable to indicate whether the data item is currently write-locked, `exclusive`. The following code snippet shows excerpts of the class definiton for `DataUnit`:

```
public class DataUnit {
    public int id = 0;
    public int verNumber = 0;
    public int currAccess = 0;
    public boolean exclusive = false;

    public DataUnit() /* default constr. */
    ...
    }
```

The L data items may be realized as arrays or for reasons of efficiency, especially for very large values of L, as hash tables, in which the data items reside only as long as they are being accessed. Version numbers are always initialized with 0. In case hash tables are used, the version numbers at the end of the simulation run would then give no clue on how many write operations/accesses have been performed to this data.

Each transaction maintains its own read and write set, realized as hash tables in the simulation model composed of objects of the class `RWSetElement`:

```
public class RWSetElement {
    public int dataUnitId = 0;
    public int verNumber = 0;

    public RWSetElement() /* def. constr. */
    ...
    }
```

For write sets, the variable `verNumber` merely serves to produce debug information, since the global version number of a data unit must not be incremented, while a transaction is holding an exclusive write lock.

The input parameter N specifies the number of threads, i.e., the maximal number of concurrent transactions competing for transactional data. Note that non-transactional code to be executed between transactions may reduce the number of concurrent active transactions. Each transaction has to perform a number of subsequent (static) read and write operations successfully in order to finish. This number of operations may be fixed to k for all transactions or chosen according to some discrete distribution with mean k (and additionally optional lower and upper bounds). 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. The assumptions about transaction sizes and request patterns are quite generic and independent of any specific application. If statistical data is available for an application, the simulation model is easily extended to incorporate this information. Similarly, the simulation model assumes a simplified timing behavior for write and read accesses, validation and commit procedures, etc., as outlined below. This is also easily refined based on available input data for these steps.

3.2 Dynamic behavior of a single transaction

The dynamic behavior of each thread process, and thus of each transaction, is defined by a single AnyLogic statechart of which N instances execute in parallel during a simulation run. Its graphical representation with simple states (rounded rectangles), branch states (diamonds) and transitions (arcs) is shown in Figure 1.

Within a thread,

- a transaction may be executed (see activities between transitions *TxnStart* and *TxnEnd*),
- a set of operations on non-transactional data of a possibly random duration may be performed before or after a transaction (in states *PreInternOps* and *PostInternOps*, respectively),
- or the thread may be idle for some period (in state *IdleThread*), which indicates that too few concurrent processes are active to exploit the multiple cores.

In our simulation, the threads – started at time 0 in state *PreInternOps* (see initial pointer *ThreadStart*) – continue to run until the stopping criteria are reached.

The action of transition *TxnStart* generates the sequence of read and write operations on items of the L transactional data. First, the number of static operations to be executed within the transaction is determined – either fixed according to input parameter k or randomly (e.g., according to a beta distribution with lower and upper bound). Then, the type for each operation, i.e., read or write, is selected according to the write access probabilities l_w. At the same time, the transactional data item to be accessed is picked from the L items according to a discrete-uniform distribution.

The sequence of operations is maintained throughout the lifetime of a transaction, i.e., it survives possible restarts until the final commit (see state *Commit*)[1].

Except after the final operation, the (unkilled) transaction returns to state *ProcessRequest* in order to process the next operation in the sequence (if the previous operation has been successful) or to restart with the first operation (otherwise). In the former case, *ProcessRequest* is entered via transition *MoreOperations*, in the latter via state *Abort* and transition *TxnRestart*.

Depending on whether the next operation is a write request or a read request, the transaction branches at the topmost branch state following transitions *WriteRequest* or *ReadRequest*, respectively.

In what has been described so far, the statechart behaves identically for all considered STM variants. Different lock management, validation and versioning within the transactions are partly reflected in graphical elements of the statechart, which are applicable only for specific STM variants,

[1]If livelocks cannot be resolved by the delay in state *Abort*, a transaction may not be restarted and an independent transaction will be started instead (see transaction *TxnKill*).

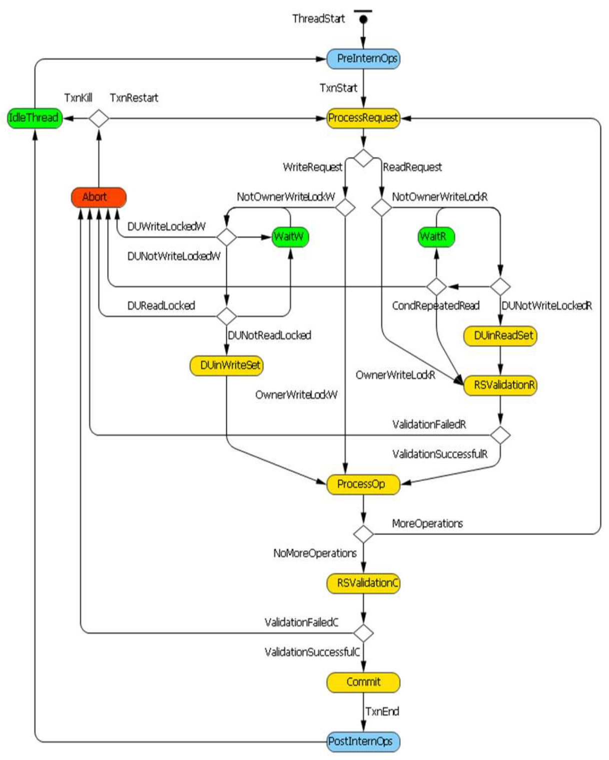

Figure 1: Statechart of thread and transaction behavior

but mostly within the Java action codes associated with transitions (especially with respect to versioning). Before we discuss the differences on our model walk-through along the read and write branches, we link three of them directly with the statechart in Figure 1:

1. Since pessimistic STM does not perform any read set validation, the transitions *ValidationFailedR* and *ValidationFailedC* will never be taken and no time is spent in states *RSValidationR* and *RSValidationC*.

2. In optimistic STM, existing read locks on transactional data do not block a write access to this data and therefore transition *DUNotReadLocked* in the write branch is always chosen over the other alternative transitions of the branch state before simple state *DUinWriteSet*.

3. The transition *CondRepeatedRead* only applies to optimistic STM with write buffering, for which speculative readers may successfully read-access WAR data repeatedly.

Let us now consider the read branch starting with transition *ReadRequest*. The first branch state checks if the transition already owns the data unit (i.e., holds a write lock on the data), which would imply the right to read the data.

If so (i.e., an element with the respective `dataUnitId` is in the write set), read and write sets do not have to be modified and the transaction may directly validate the current read set in optimistic STM or directly process the read operation in pessimistic STM. In the former case, the read set is validated in simple state *RSValidationR* (timing) and the following branch state (decision), before the operation is processed in state *ProcessOp*, if the validation is successful. Recall (see comment 1. above) that the option for pessimistic STM makes the validation procedure immediate and always successful, i.e., it is eliminated.

More commonly, the transaction with the read request will not have the write lock for the data and will be directed along transition *NotOwnerWriteLockR*. By looking up the respective `DataUnit` object's member variable `exclusive`, the subsequent check finds out, if the transactional data is write-locked or not. In the former case, the transaction may

- try to reaccess the data unit after some (possibly random) delay (path via state *WaitR*)
- or may have to abort (path via state *Abort*).
- Only for optimistic STM with write buffering (see comment 3. above), a repeated read access might initiate a validation procedure via transition *CondRepeatedRead*.

If the data unit is not write-locked (transition *DUNotWriteExclusiveR*), it is added to the read set (at the entry to state *DUinReadSet*). The read set is then validated or skipped as outlined above.

In the read branch, the transaction may be aborted, if a write lock by another transaction is encountered or if the validation procedure fails (only for optimistic STM). In the action code of the transition emanating from state *Abort*, all locks that the transaction holds are released, i.e., the write and read sets are deleted and the variables in the respective `DataUnit` objects updated. More precisely, a released read lock decrements the counter `currAccess` (which is always incremented when a respective novel element is entered into any read set) and a released write lock switches the boolean variable `exclusive` to false. In case of optimistic STM with inplace memory updates, the version number `verNumber` of any transactional data indicated by the write set must be incremented additionally.

Up to the branch state before the simple state *DUinWriteSet*, the write branch is essentially identical to the corresponding part of the read branch, including the state *WaitW*. Related specifiers differ only in the last letter (*W* instead of *R*). Note, however, that transition *OwnerWriteLockW* leads directly to state *ProcessOp*, since our STM variants never validate the read set after a write request, and that the additional branch state to check for repeated read accesses becomes obsolete in the write branch. Also, a different backoff mechanism may be employed for write accesses in state *WaitW*.

Let us assume that the write branch has certified that no write lock has been issued on the requested transactional data, i.e., the statechart control is in the branch state before state *DUinWriteSet*. According to comment 2. above, only pessimistic STM additionally checks that neither a read lock is held by another transaction[2]. Only then may the transaction add the respective write lock to the write set (at entry of state *DUinWriteSet*) and continue with processing the operation. Otherwise, the transaction aborts (i.e., enters state *Abort*) or goes into backoff (i.e., enters state *WaitW*), as already known from other failures in the lock acquisition phase.

[2]If the considered transaction holds a unique read lock on the data unit, the read lock will be replaced by a write lock.

Like for reads, the transaction is simply delayed in state *ProcessOp* by a period, which reflects the complexity of the transactional memory operation, e.g., the actual manipulation of data values, as well as other types of operations, e.g., integer operations, which may be included in the transactional memory region before the next transactional memory operation.

If the final operation of the transaction has been processed successfully, the statechart leaves the loop (via transition *NoMoreOperations*) and performs the final read set validation by comparing the version numbers. The logic of the statechart in simple state *RSValidationC* and the following branch state is in complete analogy to the validation in the read branch - only applied to a larger read set. A successful validation initiates the commit operation (see state *Commit*) with different timing for the three STM variants. The consequences of the action code of transition *TxnEnd*, however, will be very similar in all cases:

- The version numbers of the transactional data, for which write locks are held, are incremented (not necessary for pessimistic STM).
- All read and write locks are released, i.e., read and write sets of the transaction are dissolved.
- All transaction-related information (counters, data structures, etc.) is reset.

Finally, we address a technical issue pertinent to the implementation of statechart behavior in AnyLogic, but elementary for the functionality of our model. Generally, a transaction must not be interrupted by other transactions between checking the conditions for lock acquisition and actually entering the corresponding element into the read or write set. This is guaranteed in the statechart by calling the methods `DUinReadSet()` and `DUinWriteSet()`, which perform the latter set operations, in the entry action code of the states *DUinReadSet* and *DUinWriteSet*, respectively. The sequence of branch states leading to these simple states is executed without consuming time and cannot be interleaved with activities of other transactions.

3.3 Performance measures

Naturally, a discrete-event simulation model allows to specify all kinds of performance measures. For example, one could easily find out, whether the last operation of an arbitrary transaction is more likely to produce a restart, if it is a read or a write operation. In order to compare the three STM variants, we investigate rather global characteristics in this paper, like

- the mean number of restarts per transaction, denoted by $E[R]$
- the mean number of steps per transaction, $E[S]$, including all read/write requests and the commit operations also in repeated attempts after restarts,
- the mean number of locks held by a transaction (at request epochs), $E[Q]$

We are primarily interested in the first two characteristics: $E[R]$ tells us how often (on average) an arbitrary transaction had to abort and restart, while – for identical values of $E[R]$ – a smaller value of $E[S]$ indicates that these aborts occurred earlier (on average) in the static sequence of read/write operations. In ideal conflict-free situations with optimal performance, $E[R]$ and $E[S]$ should assume their minimal values 0 and $k+1$, respectively. While k denotes the length of a transaction, parameter $E[S]$ may be called its lifetime (in terms of numbers of steps). Finally, the mean number of locks, $E[Q]$, or more precisely the product $E[Q]E[S]$, serves as a rough measure for lock-related overhead, including the overhead due to meta-data organization, allocation of local copies, read/write set management, etc.

We mainly consider discrete-time statistics to reduce the impact of timing of the involved activities in order to highlight the performance differences primarily due to the contention mechanism and lock management. Naturally, the timing, like the duration of read and write operations, inherently influences the behavior of the STM variants. To understand this issue better, we will also look at the throughput (i.e., the number of processed transactions per time unit) in the next section.

In the simulation model, statistics are collected in two phases. In the active object *Thread*, which contains the statechart of Figure 1, transaction-specific information is collected in counters or `Statistics` objects provided by AnyLogic, as appropriate. For instance, an integer variable is sufficient to count the number of restarts a transaction undergoes (i.e., whenever state *Abort* is reached in the statechart), while recording the current number of locks that the transaction holds (i.e., when leaving state *ProcessRequest*) in a discrete-time `Statistics` object is most convenient in order to compute the average value. In the first phase, counters need to be incremented and `Statistics` objects updated at the appropriate (and quite intuitive) locations in the statechart. In fact, the simulation model does this for many more and much more detailed statistics than mentioned above. In the second phase, the action code of transition *TxnEnd* records all statistics of this transaction in model-global `Statistics` objects in active object `Main`, from which the final performance measures are computed at the end of the simulation.

In order to obtain statistically significant simulation results, we perform as many replications as required to achieve a relative error of 5% for confidence intervals of level 95 %. This kind of simulation control is applied only to the performance measures listed above (except for the throughput) and partly supported by the tool AnyLogic in the *Parameter Variation* experiment. In addition, *Parameter Variation* experiments facilitate to produce simulation results for varying input parameters, like increasing write request probabilities l_w.

4. NUMERICAL RESULTS

Applying simulation control as described in the previous section, we conduct an experiment series to investigate the performance differences between the three STM variants. For all cases, the input parameters L, N and k are set to values of a magnitude as they might occur in typical benchmark applications (see STAMP [17] for comparable values): $L = 1.000.000, N = 16, K = 100$. The write access probability l_w is varied between 0 and 1. Each transaction must perform a sequence of k static operations, in which only repeated reads may occur, but no repeated writes. Furthermore, Table 1 summarizes the timing associated with the transitions emanating from simple states for pessimistic

STM. While most timeout values are 0 or constant, but possibly dependent on the current size of the read and write sets (i.e., $\|RS\|$ and $\|WS\|$, respectively), the time spent in states *ProcessOp* and *Abort* is distributed according to a (shifted) exponential distribution.

Table 1: Sojourn times in simple states for pessimistic STM

state	delay distr.	mean
ProcessRequest	constant	0
DUinWriteSet	constant	$0.1\frac{\|WS\|}{k}$
DUinReadSet	constant	$0.1\frac{\|RS\|}{k}$
RSValidationR	constant	0
ProcessOp	exponential	1.0
RSValidationC	constant	0
Commit	constant	$0.1\frac{\|RS\|+\|WS\|}{k}$
PostInternOps	constant	0
IdleThread	constant	0
PreInternOps	constant	0
Abort	constant and exp.	$0.1\frac{\|RS\|+\|WS\|}{k}$ 1.0

In order to reflect the operational overhead for optimistic STM, e.g., due to validation, some timeouts are increased by some constant as compared with pessimistic STM:

- Validation time in each of the states *RSValidationR* and *RSValidationC* depends on the size of the read set and takes $0.01\frac{\|RS\|}{k}$ time units.
- Since optimistic STM with write buffering requires a commit operation to copy the local write buffers into global memory locations, the value $0.1\frac{\|WS\|}{k}$ is added to the sojourn time in the state *Commit* in this case.
- Since optimistic STM with inplace memory updates has to roll back the original values in its undo log during an abort, the value $0.1\frac{\|WS\|}{k}$ is added to the sojourn time in the state *Abort* in this case.

Figures 2 to 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. (We omit the confidence intervals in the figures, since they are rather small and would impair the readability of the graphs.) All performance measures for a single variant with the statistical significance mentioned above were obtained in less than two hours on a Windows PC with Intel Pentium 4 CPU with 3.4 GHz and 1 GB RAM. Each replication, of which between 4 and 11 had to be performed for a fixed value of l_w, processed around 100,000 transactions.

With more and more write operations, the mean number of restarts and the mean number of steps per transaction generally increase due to more conflicts. However, the mean number of locks held per transaction, $E[Q]$, slightly decreases, since the restarts cause the transaction to spend relatively more time in states where it holds fewer locks. As already indicated in Section 2.3, the three variants exhibit similar performance in the extremal points, i.e., for $l_w = 0$ and $l_w = 1$. With either only read or only write operations, the qualitative behavior of pessimistic STM and optimistic STM (with eager lock acquisition) is actually identical. For $l_w = 0$, 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

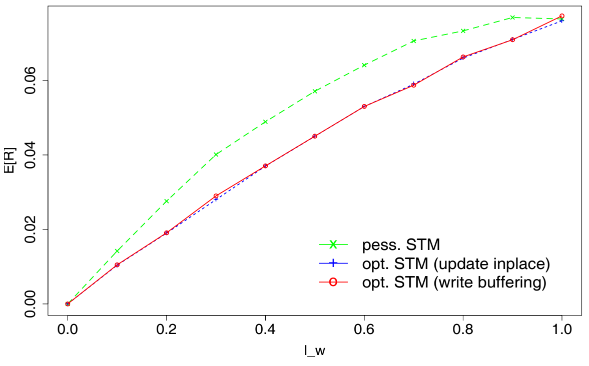

Figure 2: Mean number of restarts $E[R]$ per transaction vs. write probability l_w

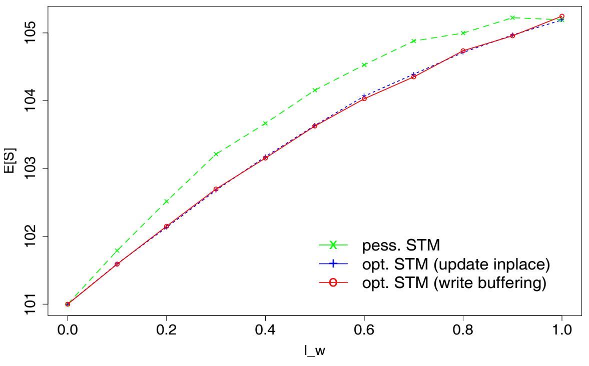

Figure 3: Mean number of requests $E[S]$ per transaction vs. write probability l_w

number of restarts, $E[R]$, grows to a value of around 0.076, which causes the transactions to finish in slightly more than 105 steps (on average) and hold around 49.57 locks (on average).

Between the extremal points, optimistic STM shows a significantly better performance than pessimistic STM (with respect to $E[R]$ and $E[S]$), especially in the range $0.4 < l_w < 0.6$, where the mean number of restarts per transaction is decreased between 20% and 25%. For larger values of l_w, the curves for pessimistic and optimistic STM approximate each other less smoothly than for smaller values of l_w.

For eager lock acquisition, the two optimistic STM variants do not differ significantly in performance. In Figures 2 to 4, the two optimistic curves can hardly be distinguished. With the rather few conflicts observed in our experiments (as desired for realistic applications), the differences in the versioning rules have only limited impact on the TM performance. Especially Figure 2 would support the claim that in our setting optimistic STM with inplace memory updates is to be favored over optimistic STM with write buffering: with the almost identical performance in terms of the low number of restarts per transaction, the implementation by means of

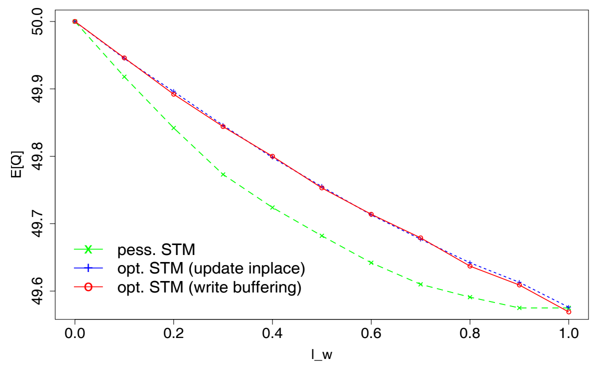

Figure 4: Mean number of held locks $E[Q]$ per transaction vs. write probability l_w

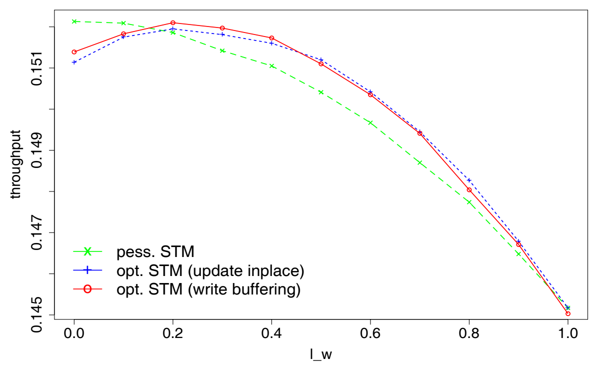

Figure 5: Throughput (in number of transactions per time unit) vs. write probability l_w

inplace memory updates should be more efficient due to the lower lock overhead. Recall that then these implementations reduce the need to copy memory locations to a minimum. Since an average transaction encounters much fewer abort operations than the one successful commit operation (with the set write WS usually being smaller at abort time than at commit time), the different timing addressed above should lead to slightly shorter transaction processing times for inplace memory updates - and thus to slightly larger throughputs. However, Figure 5 suggests that this is only the case for $l_w \geq 0.5$, i.e., when there are more write than read operations. For $l_w \leq 0.4$, the throughput of optimistic STM with write buffering appears slightly larger. Theoretically, both throughputs should be identical at $l_w = 0$ (when the write set WS is always empty). So, we can only attribute the deviation of the two optimistic curves in this point of Figure 5 to the variability of the timing behavior.

The theoretic throughput relation of pessimistic STM at $l_w = 0$ is captured correctly by Figure 5: as pessimistic STM does not perform validation checks, transactions finish faster, which results in higher throughput. With increasing l_w, the relationship with respect to optimistic STM is already inverted at $l_w = 0.2$, when the lower number of restarts and the higher degree of concurreny begins to pay off for optimistic STM. However, we point out that this turning point as well as the quantity of the throughput differences may be very sensitive to specific model parameters. For example, we observed that increasing the validation times associated with states *RSValidationR* and *RSValidationC* by a factor of 10 pulls down the throughputs for optimistic STM dramatically at $l_w = 0$. As a consequence, pessimistic STM then maintains a higher throughput in the range $l_w \leq 0.5$. This highlights the importance of proper and efficient validation schemes.

Finally, we mention that optimistic STM with write buffering may be (and is often) combined with lazy lock acquisition, which may further improve the performance of this variant. We plan to investigate the impact of this and other modifications of the basic STM variants in future studies.

5. CONCLUSIONS

Optimizing transactional memory systems is a very challenging task, as it involves dealing with very complex parameter interactions. This complexity does not facilitate the understanding of TM systems. Currently implemented TM systems realize various design decisions and measured performance differences across systems are very hard to relate to specific aspects. Therefore, we propose in this paper a simulation model that integrates three basic STM variants into a single model for the purpose of a detailed performance comparison. In its current version, the model allows the user to assess baseline algorithms for pessimistic STM and optimistic STM with either write buffering or inplace memory updates. Thus, the performance impact also of minor qualitative differences may be assessed, specifically in this paper between the two optimistic STM variants with eager lock acquisition. For parameter settings common for STM applications, we found that optimistic STM may improve performance in terms of mean number of restarts per transaction by around 25%. In future work, this simulation model will be extended to study different validation strategies for optimistic STM, different backoff mechanisms before and after abort and deadlock avoidance mechanisms.

6. REFERENCES

[1] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proc. 11th Int. Symposium on High-Performance Computer Architecture (HPCA'05)*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Jayaram Bobba, N. Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *Proc. 35th Int. Symposium on Computer Architecture (ISCA'08)*, Bejing, China, June 2008.

[3] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proc. 34th Int. Symposium on Computer Architecture (ISCA'07)*, San Diego, CA, USA, June 2007.

[4] Luis Ceze, James Tuck, Josep Torrellas, and Calin

Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. 33th Int. Symposium on Computer Architecture (ISCA'06)*, Boston, MA, USA, June 2006.

[5] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, O. Colavin, and B. Calder. Unbounded page-based transactional memory. In *Proc. 12th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, Boston, MA, USA, Oct 2006.

[6] XJ Technologies Company. Anylogic 6.2.2. Multi-method simulation software, Petersburg, Russian Federation, http://www.xjtek.com/, 2009.

[7] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proc. 12th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, pages 336–346, Boston, MA, USA, Oct 2006.

[8] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proc. 20th Int. Symposium on Distributed Computing (DISC'06)*, pages 194–208, Stockholm, Sweden, February 2006.

[9] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proc. 13th ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming (PPoPP'08)*, Salt Lake City, USA, February 2008.

[10] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[11] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (TCC). In *Proc. 11th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[12] Armin Heindl and Gilles Pokam. An analytical performance model of software transactional memory. In *Proc. IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, Boston, MA, USA, April 2009.

[13] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. 22nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC'03)*, pages 92–101, Boston, MA, USA, July 2003.

[14] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th Annual Int. Symposium on Computer Architecture (ISCA'93)*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[15] James Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[16] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Proc. 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, Ottawa, Canada, June 2006.

[17] Chi Cao Minh, Jae Woong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. IEEE Int. Symposium on Workload Characterization (IISWC'08)*, pages 35–46, Seattle, WA, USA, September 2008.

[18] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proc. 12th Int. Symposium on High-Performance Computer Architecture (HPCA'06)*, pages 254–265, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Y. Ni, A. Welc, A. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'08)*, Nashville, TN, USA, October 2008.

[20] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. *SIGARCH Comput. Archit. News*, 33(2):494–505, 2005.

[21] B. Saha, A.-R. Adl-Tabatabai, R. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high-performance software transactional memory system for a multi-core runtime. In *Proc. 11th ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming (PPoPP'06)*, New York, NY, USA, 2006.

[22] Nir Shavit and Dan Touitou. Software transactional memory. In *Proc. 14th ACM/SIGACT-SIGOPTS Symposium on Principles of Distributed Computing (PODC'95)*, pages 204–213, Ottawa, Canada, 1995.

[23] A. Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Computing Surveys*, 30(1):70–119, March 1998.

[24] Christoph von Braun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proc. 13th ACM SIGPLAN Symposium of Principles and Practice of Parallel Programming (PPoPP'08)*, Salt Lake City, USA, February 2008.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2009.5581
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2009.5581