VU University Amsterdam

BMI Paper

Predictive modeling for software transactional memory

Author:
Tim Stokman

Supervisor:
Sandjai Bhulai

15 October, 2010

Abstract

In this paper a new kind of concurrency type named software transactional memory is examined. Multiple algorithms (namely, the OSTM and the DSTM algorithm) are compared for their performance in a worst case scenario. This situation is modeled by using discrete time Markov chains with a fixed point iteration heuristic to solve a dimensionality problem. This model is verified with simulation and a real world implementation of the algorithms. We conclude that the OSTM algorithm is the safest alternative that remains stable when used in a bad situation (high contention or differently ordered variable access) while DSTM has on average better performance in a normal situation, but handles situations with differently ordered variables badly. We also conclude that fixed point iteration cannot be used to model this situation due to conditional dependencies between the model states, and that simulation is a better alternative to help predict these situations.

Contents

1 Introduction 2
- 1.1 Software transactional memory . . . 4
- 1.2 The problem . . . 5

2 Software Transactional Memory 6
- 2.1 DSTM . . . 7
  - 2.1.1 Contention Management . . . 7
- 2.2 OSTM . . . 8
- 2.3 Current disadvantages of STM systems . . . 9

3 Modeling 9
- 3.1 Tagged transactions . . . 9
- 3.2 OSTM Model . . . 11
- 3.3 DSTM Model . . . 14
  - 3.3.1 Polite contention management . . . 17
- 3.4 Verification of the model . . . 17

4 Results 18
- 4.1 OSTM Results . . . 18
- 4.2 DSTM Results . . . 20
  - 4.2.1 Aggressive contention management . . . 20
  - 4.2.2 Polite contention management . . . 21

5 Conclusion 26
- 5.1 The model . . . 26
- 5.2 The algorithms . . . 26

6 Attachment 1: DSTM implementation 27

7 Attachment 2: OSTM implementation 28

8 Attachment 3: Timing data 29

1 Introduction

The microprocessor industry has been undergoing a small revolution for the last couple of years [8]. The clock speed at which current processors run cannot be significantly increased without overheating and damaging the CPU. So instead of increasing the individual clock speed, manufacturers have resorted to building processors with multiple execution cores (multicore processors) that communicate through shared memory caches. At the time of writing, new consumer PCs include microprocessors with up to six cores.

If you want to build software that takes full advantage of the power of modern multicore processors, you have to build programs where large parts of the program can be executed in parallel. Such a concurrent program has multiple threads of execution communicating through shared memory. These concurrent programs are more difficult to build because the multicore architecture is inherently asynchronous, the multiple threads of execution

can interleave in unexpected and unpredictable ways. The operating system can stop one thread in the middle of modifying a critical piece of data, and then execute another thread to read that same piece of data. These sequence- or time dependent bugs are called *race conditions* (see Figure 1).

In this example (see Figure 1) two threads try to read and write to two different variables. These threads execute the read and write instructions in parallel. This interleaving of the execution causes the results to be undefined. The end results are dependent on the exact timing of the execution.

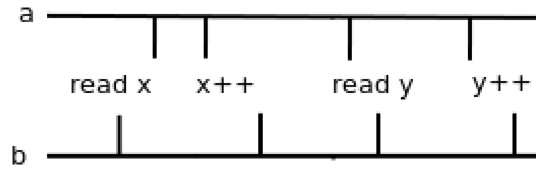


Figure 1: race condition: two shared variables are written by two threads, the interleaving of shared memory access can cause unpredictable end results

The classic method to ensure the correctness of a concurrent program is mutual exclusion [8]. For each *critical section*[1] of the program, access is regulated with a *mutex* or *lock*. This *lock* ensures that only one *thread* can execute that *critical section* at the same moment. If these locks are used correctly, the asynchronous nature of the execution will not influence the result of the program.

While simple and effective, the classical mutex mechanism has a number of issues that make it difficult to use in some contexts. Locks can be vulnerable to bugs that can be very subtle and hard to reproduce since they are triggered by a very specific order of events. Using locks in the wrong order can cause *deadlocks*. In this example (see Figure 2) each thread needs to acquire two locks to access their critical section. If a thread attempts to open a lock that is already in use then it has to wait until the lock is available again. Since both threads are waiting for the other thread to release a lock, they are suspended indefinitely, most likely causing the program to freeze.

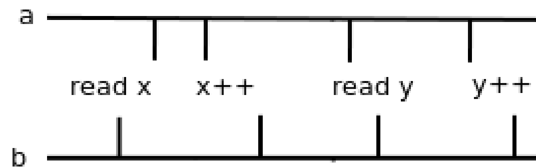


Figure 2: deadlock: two threads blocking each other because of the way they acquired the locks

There can also be problems with priority inversion[2] or conveying[3]. Only through carefully ordering the lock acquisitions and protecting each shared memory access can most of these problems be avoided.

For larger programs, arguably the most important disadvantage of mutual exclusion is that it is not composable. This means that multiple pieces of programming code where critical sections are protected in a correct way using locks often cannot be used together to make a larger correct program. Modern programs often use a lot of external libraries to provide certain functionality such as graphical primitives, database access or network access. This saves time implementing the functionality and makes the final implementation of the program more simple. But because locks are not composable, using external libraries in a correct way concurrently is often impossible. For example, a program uses a library provided to him by his local bank. This library has several routines to manage his account, which are correctly protected by locks on the accounts to prevent concurrent modification of the accounts.

Listing 1: Bank code library

```
def withdraw(account, amount):
  lock(account)
  if(account.money > amount and amount > 0):
    account.money = account.money - amount
  unlock(account)
```

[1] A piece of the program that cannot be executed concurrently due to reading or modifying shared resources

[2] High priority threads are blocked by low priority threads holding locks

[3] A thread is preempted in the middle of a critical section, causing a lot of other threads to wait for the lock to be released

```
def put(account, amount):
  lock(account)
  if(amount > 0)
    account.money = account.money + amount
  unlock(account)
```

Consider a program that wants to use this library (see Listing 1) to transfer money between two accounts. For practical reasons the original library cannot be modified and the program cannot access the locks on the accounts. A naive implementation might use the withdraw and put routines to transfer the money. But this transfer routine would not be *atomic*. At the point between the two method calls, there is a large amount of money missing from both accounts. If the thread executing the transfer is preempted[4], another thread can read an inconsistent state of the system where there is no money in both accounts. For the program to work correctly, it has to modify the original library so it can acquire both locks at the same moment.

1.1 Software transactional memory

There are several alternatives that try to correct the disadvantages of mutual exclusion and offer a new way of synchronizing multiple threads. One of those alternatives is software transactional memory. Software transactional memory offers a concurrent programming model similar to database transactions. A transaction consists of a number of reads and writes to shared variables executed in a single thread. This sequence of steps must appear to take place at once (linearizable) from the perspective of other threads and the transaction has to be executed in an atomic way (all or nothing).

The nice thing about software transactional memory is that theoretically you can reason about concurrent code build with software transactional memory as if it were single-threaded. There is no need to worry about deadlocks and ways threads can interleave, the system provides an abstraction layer that manages these things for you. Software transactional memory system offers the following interface to programmers:

- starting a transaction
- opening an object for reading or writing
- committing a transaction

Using this interface, the user can mark where the transaction starts and ends and which variables are modified or read in this transaction. Then, as the transaction commits, the changes become visible to other threads. Sometimes the transaction has to abort because it encounters a conflict. For example, another thread could try to modify a variable the current thread is modifying or a variable read by a thread is overwritten by another thread before it finishes committing the transaction. The transaction then has to be restarted.

Doing this manually is time-consuming for the user, they would have to identify which variables are read from and written to and would have to restart transactions manually if they are aborted. The ideal STM implementation would be integrated in the programming language providing an *atomic* block [4] in which code would be executed and restarted as a transaction, automatically identifying which shared variables are modified and which are only read. The following example of this integration (see Listing 2) does a number of things implicitly:

- It starts a transaction
- It opens the shared variables x and y (see Listing 2) for writing and adds one to each variable
- It commits
- If it fails at any point during the transaction (due to conflicts), it restarts

Listing 2: software transactional memory integrated in the programming language

```
atomic:
  x++
  y++
```

[4] The operating puts the thread to sleep so it can schedule another thread

Since STM implementations provide support for nested transactions to make the implementations composable, we can use this feature to solve the bank problem without changing the original library (if the library uses software transactional memory).

Listing 3: the bank problem solved with software transactional memory

```
# Library code
def withdraw(account, amount):
  atomic:
    if(account.money > amount and amount > 0):
      account.money = account.money - amount

def put(account, amount):
  atomic:
    if(amount > 0)
      account.money = account.money + amount

# External program using the library
def transfer(from, to, amount):
  atomic:
    withdraw(from, amount)
    put(to, amount)
```

This example (see Listing 3) solves the bank problem because the nested transaction in the transfer routine makes sure that it is executed atomically. The state of the account (the amount of money in it) is never exposed to other threads while the transfer transaction is executed.

1.2 The problem

What we want to study is how the available software transactional memory algorithms scale up from a small number of threads to a larger number of threads. More specifically, we will model how each system will handle a livelock-prone[5] situation where it is likely to get stuck trying to resolve conflicts.

Listing 4: situation being modeled

```
  x = 0
  y = 0

  n threads:
    while(true):
      atomic:
        x++
        y++
  m threads:
    while(true):
      atomic:
        y++
        x++
```

In this situation (see Listing 4) there are two types of threads running. The first of type thread writes the variables in the opposite order the second thread writes them. This situation is very hard to handle for these algorithms for two reasons:

- There is a lot of contention because multiple threads are trying to write to the same set of variables
- Threads are writing in opposite order meaning the algorithm could livelock while trying to deal with the conflicts

We are especially interested how STM algorithms hold up when a lot of threads are trying to write to the same two variables. The model will calculate given the STM algorithm, number of threads for each type and timing for each part of the algorithm what the abort probability and average latency[6] is. Using this model we will compare the following algorithms:

[5] A livelock is a conflict situation similar to a deadlock where multiple threads repeatedly block each other while attempting to resolve a conflicting situation. The difference with the deadlock situation is that when threads are deadlocked they are waiting for the other thread instead of trying to resolve the conflict

[6] time until the next successful transaction

- Dynamic Software Transactional Memory [7] (DSTM) with the aggressive and polite contention manager [9,10]
- Optimistic Software Transactional Memory [2] (OSTM)

These two STM implementations have different strategies and philosophies with regard to handling conflicts and contention. DSTM has a conflict handling mechanism (the contention manager) that can be changed and adapted by the user to his specific situation while OSTM uses a costly algorithm to guarantee a certain minimum throughput. Since DSTM relies on the contention manager strategy to prevent livelock situations, we are curious how some of the basic DSTM contention managers (polite and aggressive) stack up against the OSTM approach in this situation.

The ideal way this situation could be handled can be derived by Amdahl's law [8]. With this law you can derive the maximum speedup of an algorithm that is executed in parallel. Since the situation that is being modeled cannot be executed in parallel, the best scaling factor that can be achieved is: $S = \frac{1}{1-P+\frac{P}{N}} = 1$ (with P being the fraction that can be executed in parallel and N the number of available processor cores). This means that in the best case scenario the latency of the algorithm stays constant.

2 Software Transactional Memory

Most of the software transactional memory implementations belong to the class of nonblocking concurrent algorithms [8]. This means that that they ensure that any thread is not indefinitely blocked by means of mutual exclusion. Nonblocking algorithms (and most software transactional memory algorithms) can offer different kinds of progress guarantees to threads using these algorithms. A nonblocking algorithm is:

- Obstruction-free if any thread executed in isolation (with other threads suspended or not running) will eventually complete in a limited number of steps. This is the weakest guarantee one can have for these classes of algorithms. It does not exclude the possibility of a livelock and can cause problems when certain pieces of data are highly contended. Obstruction-free algorithms allow that all partially completed threads can be aborted and rolled back whereas algorithms with higher progress guarantees rely on helping other threads and ordering operations. The decision when and how to abort conflicting threads is done with an external contention manager, which is responsible for avoiding livelocks.
- Lock-free if there is a system-wide guarantee that at least one thread eventually makes progress in a limited number of steps. Lock-freedom is a stronger progress guarantee meaning that not all partially completed threads can be aborted on a conflict. This guarantee is usually reached by expensive measures like helping conflicting threads and ordering operations.
- Wait-free if there is a system-wide guarantee that all threads will eventually complete their work in a limited number of steps. This is the strongest progress guarantee, but typically has a large overhead in the form of latency and memory requirements. Wait-free algorithms are rarely used in practice because of this large overhead.

Some STM algorithms are implemented without any progress guarantee [1], letting transactions run as long as they do not encounter a conflict that cannot be solved without aborting at least one transaction. This gives longer running transactions a greater probability to succeed, but can give problems when some suspended or stuck transactions block the progress of other transactions. Since this type of algorithm is a bit harder to model using the method we use, we will not analyze it.

Most of the software transactional memory implementations (and other nonblocking algorithms) are implemented using read-modify-write atomic primitives [8]. These primitives are usually implemented on the hardware accessible as a processor instruction and do a read and write in an atomic way so no race conditions can occur. One of the more notable atomic primitives is compare-and-swap (see Listing 5) which checks if a value is the same as the expected value before setting the value to a new value. Just like all other read-modify-write atomic primitives, it does this in an atomic way. It cannot be interrupted by other threads. This makes it ideal to check if certain shared data was changed before writing to a shared variable, without this instruction race conditions could occur.

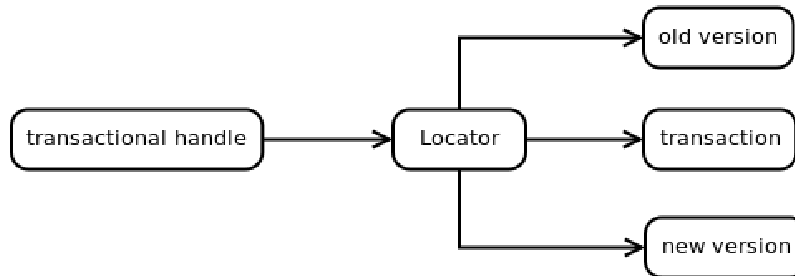
Listing 5: semantics of the compare-and-swap atomic primitive

```
def CAS(data, expected, to):
  if(data == expected):
    data = to
```

2.1 DSTM

Dynamic software transactional memory [7] provides an obstruction-free implementation of software transactional memory. DSTM keeps track of different versions and ownership through a so-called transactional handle. Through this handle the shared variable can be accessed by opening the handle for reading or writing. The handle uses a locator object (see Figure 3) to track different versions. The locator contains the old and the new version and the transaction that currently owns the shared variable meaning only that transaction can write to it. The current status (ACTIVE, ABORTED or COMMITTED) of the transaction determines what version is valid. If the transaction is still active, the current version is unknown. When a transaction wants to change its (or another transaction's) status it uses the CAS operation (see Listing 5) to make the switch. So if a transaction determines it has failed or that it is done, it attempts to execute the CAS operation to signal other threads that it has committed or aborted.

Figure 3: DSTM Transactional handle



When a transaction wants to write to a shared variable, it has to open the handle for writing. It first checks the current locator object. If the transaction that owns the variable is active, then there is a conflict and it will have to be handled by the contention manager (see below). Otherwise, the DSTM implementation can try to acquire the shared variable by executing a CAS operation (see Listing 5) on the handle to switch the locator object to a new one. The new locator contains a reference to himself, a reference to the last valid version as the old object and contains a *copy* of the last version as the new object. This copy is returned to the user so it can be written to.

Reading objects is handled more optimistically. Since reading does not require exclusive access (many threads can read at once while only one can write without causing conflicts), DSTM uses a read list which keeps track of which versions of which variables were read by the transaction. If the variable being opened for reading has an active transaction as owner, then there is a conflict which will have to be handled by the contention manager (see below). After each attempt to open, and before committing, this list is checked to see if any version has been changed since it was read. If this is the case, then the transaction is aborted. After verifying the read list, the transaction itself is verified to check if has been aborted by another transaction.

When the transaction is done, it tries to set its status to COMMITTED by executing a CAS operation to signal other threads that it is no longer writing to the variables it owns. If it succeeds, all the changes the transaction has made are simultaneously published. If another transaction has aborted this transaction, the CAS operation fails and the transaction has to restart.

2.1.1 Contention Management

If there is a conflict between two different transactions (two transactions want to write to the same shared variable or one transaction is reading while the other tries to write) then DSTM uses a contention manager to decide what transaction (if any) should be aborted and which transactions should wait (by suspending the thread a certain amount of time) before trying again. Users can pick a contention manager that is appropriate for their program or even their specific transaction. The DSTM can offer this flexibility because it is only obstruction-free, this gives it the freedom to abort any partially completed transaction.

The contention manager can use heuristics to ensure throughput and fairness independent of the correctness of the implementation. Through these heuristics the contention manager tries to avoid livelocking and ensure progress. The only rule the contention manager has to obey is that it eventually has to abort another transaction when a thread asks for it repeatedly, this rule is necessary to preserve obstruction-freedom. The contention manager is signaled each time the transaction attempts to open a handle for reading or writing, and then the transaction can ask if it can abort through a *should_abort* method. In addition to granting permission to abort another transaction, the *should_abort* method can also suspend the current transaction before denying permission. This allows the other transaction to complete.

The most simple and arguably the worst contention manager is the aggressive contention manager. It gives permission to abort each transaction that conflicts with the current transaction. While this strategy may work for simple situations, it quickly leads to livelocks in the real world where transactions keep aborting each other preventing any transaction from progressing. We expect that this contention manager will not scale much beyond *one* thread.

One of the better performing simple contention managers [9] is the polite contention manager. It uses a strategy of exponential backoff by spinning a randomized (usually normally distributed) amount of time with mean $k \cdot 2^n$ nanoseconds where n is the number of conflicts encountered for this object and k is a constant used for tuning. The polite contention manager backs off a maximum number of N times before it allows the conflicting transaction to be aborted. We use a slightly modified version in our implementation that spins *exactly* 2^{n+k} nanoseconds. While randomization of the suspend time can be beneficial to prevent livelock situations, it is harder to incorporate in our model since it assumes that each part of the algorithm takes an approximately constant amount of time.

There are other more advanced contention managers that use other heuristics to manage conflicts. For more details see [9,10]. Attachment 1 contains a simple DSTM implementation to illustrate the concepts explained above.

2.2 OSTM

Optimistic Software Transactional Memory [2] is a lock-free implementation of software transactional memory. In contrast to DSTM, OSTM separates the writing to shared variables from acquiring them (lazy acquisition). It uses private read and write lists for each transaction to keep track of the version that was written to and the new versions and uses recursive helping to resolve conflicts. The shared variables are stored in a handle object. The handle object either:

- points to the value itself, indicating that it is currently not acquired
- points to a transaction, indicating that the transaction currently owns the variable. The real value is then stored in the transaction's write list

When a transaction gets a request to open an object for writing or reading, it checks the handle if it is currently owned. If it is owned, it is currently not committed and has acquired all its variables (this is the case if its status is READ_CHECK), the transaction tries to get the right version by helping the other transaction. It executes the commit method (see below) on the other transaction and commits the changes for the other transaction. Otherwise, if the conflicting transaction has already committed or not acquired the whole write list, it can get the right version from the private write list of the other transaction. If the transaction has committed but not released it gets the new version, otherwise it gets the old version[7]. After the currently right version has been obtained, it is stored in a private write list together with a copy (for writing to) and the handle. The same thing happens when an object is opened for reading, only in this case no copy is generated since it is not needed.

After a transaction has written and read everything it has to, it will commit in order to "publish" the changes. First it will try to acquire everything in the write list. The acquisition happens in a global ordering (sorted on pointer value or some other unique id) to preserve lock-freedom. To signal other transactions it owns the object, it replaces the object pointed to by the handle with the transaction object through a CAS operation (see Figure 4). There are a number of different situations it needs to deal with:

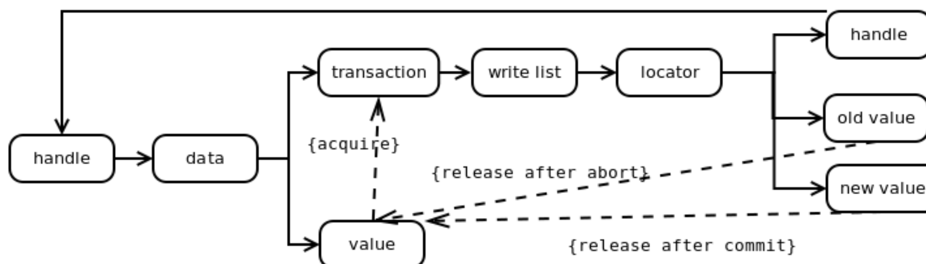
- Another transaction owning the object: in reaction to a conflict the commit procedure will "recursively help" the other transaction. It will apply the commit procedure (which it is currently executing on itself) on the other transaction helping them either complete or fail. This is especially helpful when threads are suspended blocking other transactions, but can be very costly especially considering processor caching[8]. It can also cause problems when the other transaction has been preempted while releasing variables, this means the helping transaction can recurse infinitely when it cannot acquire the released variable. After it has committed the other transaction, it will attempt to acquire the variable again
- The transaction already owns the object: this means another transaction has recursively helped the transaction and has acquired the object for him. There is no conflict so we can continue acquiring the next variable.

[7] The algorithm is not very optimistic in this regard, it assumes the transaction will fail, regardless it makes modeling slightly easier

[8] Each core or processor has its own private cache where it keeps data it frequently accesses. If another core has to access data in that cache, it costs more time to do this because the data has to be synchronized with the other processor

- The object has been overwritten by another transaction: this means that the transaction is invalid and it immediately changes its status to ABORTED and starts to release the owned shared variables. This illustrates one of the biggest disadvantages of the optimistic lazy acquire, instead of allowing a contention manager to manage potential conflicts. It is optimistic about the probability a conflict happening and waits until later to acquire the object.

Figure 4: OSTM release/acquire process



After a transaction has acquired all the objects on the write list, it changes it status to READ_CHECK to signal this other threads. After that it verifies the read list, immediately aborting and releasing all acquired objects if it finds changes. If it encounters a conflict it either aborts the other or helps the other transaction dependending on the ordering between transactions (is its pointer value or other unique id larger than the other transaction). If the transaction did not encounter any problems during the acquisition and read check phase, it can change its status to COMMITTED (or ABORTED if it has failed). After that, the transaction has to release each shared object (again in an ordered way). It does this by executing a CAS operation to change the handle object from the transaction to the new version (or an old version if it was aborted), effectively releasing the variables and publishing the changes (see Figure 4). Attachment 2 contains a simple OSTM implementation to illustrate the concepts explained above.

2.3 Current disadvantages of STM systems

There are some unsolved problems with current implementations of STM systems. Software transactional memory depends on the reversibility of each of the actions in the transactions in case it encounters conflicts. Most IO systems (network, disk etc. . .) are not reversible by nature. This means software transactional memory has to:

- Restrict access to IO actions in transactions
- Use only reversible IO sources in transactions such as transactional database systems
- Buffer the IO actions until the transaction has ended
- Do nothing and assume the programmer will realize the issues and handle them (dangerous and common)

There are also still performance issues with current STM systems (some of which we examine in this paper). Suspended transactions, high contention or different ordering of operations can cause bottlenecks in the systems. Priority inversion can also cause problems when low priority transactions interrupt transactions with a higher priority.

In the end, software transactional memory is not the ultimate solution to synchronization problems. Several programming languages (like haskell or clojure [3] [5]) have successful and mature STM implementations, but Microsoft has recently canceled their own implementation in .NET (virtual machine similar to the JVM) due to several issues (some which are mentioned above).

3 Modeling

3.1 Tagged transactions

A software transactional memory algorithm consists of a number of phases where it synchronizes with other threads and makes decisions (branching out) based on what happened. For example, the algorithm may check if a variable

is acquired or if another transaction has already committed, and will decide if it should abort the other transaction. Based on this, we could model the entire system as discrete time Markov chain. However, this has one large problem: dimensionality. We would need s^n states where n is the total number of threads and s is the number of states for each thread. There is another problem, each phase of the algorithm takes a different amount of time, and different threads are not executed in lockstep with respect to each other. This means that the system is impossible to model as a single discrete Markov chain.

The solution is a tagged transaction approach (similar to [6]). To approximate the limiting distribution of the whole system, we model a single transaction and generalize those results to the rest of the system. We model the transaction as a discrete Markov chain whose transition probabilities depend on the limiting distribution and transition matrix of the other transactions. Since we do not have these values, we have to compute them in an iterative way through fixed point iteration. Given that $M_1(m_1, m_2)$ and $M_2(m_1, m_2)$ are functions that generate the transition matrices for transactions of type one and two given the transition matrices of other transactions of those types, the fixed point of these functions is where $M_1(m_1, m_2) = m_1$ and $M_2(m_1, m_2) = m_2$. This point is the limiting distribution of the system. We can calculate certain aspects of the behavior of the system like the long-term average number of restarts or the average latency[9] from the time limiting distribution.

Our starting point as transition matrix for both types of threads is an $n \times n$ identity matrix (where n is the number of states necessary to model the transaction) which is shifted by one to the right giving a matrix that advances one state each time. Given a transition matrix, we can calculate the limiting distribution π_i for $i = 1, 2, \ldots, n$ (the right eigenvector whose eigenvalue is one). Because we know the time t_i spend in each state (through benchmarking, see Attachment 3) we can also calculate the time-limiting distribution $v_i = \frac{t_i \cdot \pi_i}{\sum_{j=1}^{n} t_j \pi_j}$. Given these values, we can calculate better approximations for the transition matrices with T_1 and T_2. We iterate these functions until the maximum relative change of the limiting distribution is less than $\alpha = 0.001$.

The models for the software transactional memory systems have a number of assumptions that are necessary to model the system, but that are often not entirely correct. The models use the following assumptions:

- The model assumes that all threads using STM have full use of one core or processor, no thread is interrupted by the scheduler[10]. This is close to the optimal way of running programs with STM algorithms [1]. Each interruption is very costly because most systems either have to wait or help the stopped transaction. If you assume that some transactions have interruptions, then the model has to incorporate the specific scheduling algorithm used when executing the STM algorithm. This means that the model gives an answer that is only applicable to the systems using that specific scheduling system. The assumption mostly holds on a system with more cores than threads. But even on a system with more processing cores than threads, a program can sometimes be suspended or scheduled to another core (which causes pauses due to caching) due to a number of unpredictable scheduler heuristics.

- The model assumes that at each state in the transaction, the states of the other transactions are approximately distributed according to their time-limiting distribution. If this assumption is incorrect, the model will not work because it assumes the wrong probability distributions for the other threads. Without this assumption, we cannot model the system using known techniques without running into dimensionality problems.

- The model assumes that each part of the algorithm modeled in the Markov chain takes a fixed amount of time. Since all model states were chosen in a way such that the algorithm has to perform a fixed amount of work in each of them, this assumption is mostly correct. There are small hardware related features like branch prediction that could change the timing of some states during the execution of the algorithm, but these small changes should have little influence overall. There is one part of the DSTM algorithm that was slightly modified so it conforms to this assumption, instead of spinning a randomized time waiting for a conflicting transaction to finish its work, the polite contention manager spins a fixed amount of time. It does not change the algorithm significantly, but allows us to check the correctness of the rest of the model without the randomization factor interfering.

Using fixed point iteration has been successfully applied to other systems where it was used to accurately approximate systems with dimensionality problems [11]. We expect that it will provide a reasonable approximation since it worked well for other systems. But in the end, the accuracy of the model depends on how well the model assumptions hold.

[9] average time until the next success

[10] A scheduler is the part of the operating that assigns different threads or processes to the different execution cores that are available

3.2 OSTM Model

The discrete Markov model of an OSTM transaction contains 57 states. The first eight states model the process of adding a variable to the write list. The states up to number twenty-four model the process of committing the changes. At four points in the model, the transaction has to help another transaction. The process of helping another transaction is modeled by eight helping states. Each helping state is replicated four times for each entry point.

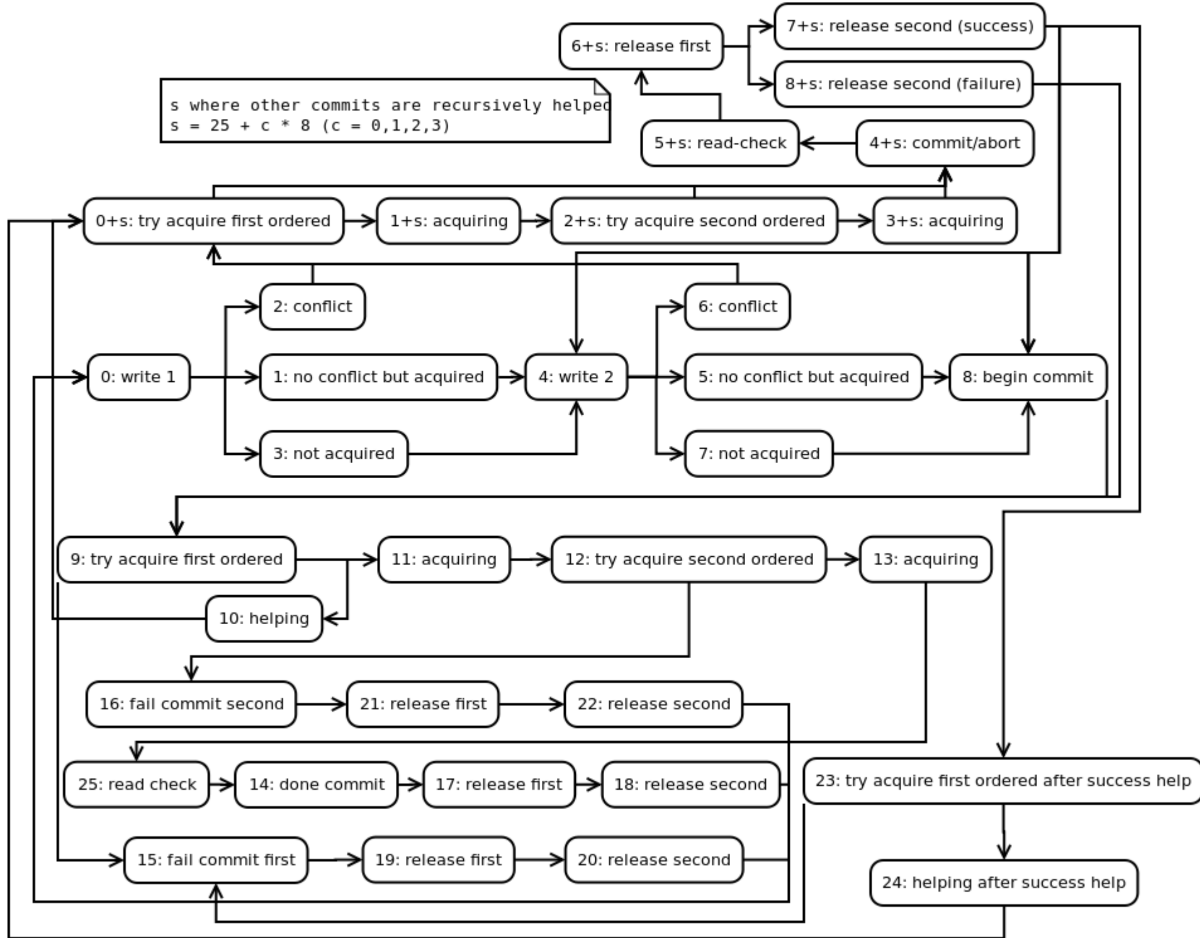


Figure 5: Discrete time Markov model of the OSTM algorithm (model created by myself)

0-7: Adding to the write list A value is added to the write list here. The transaction has to check if another thread has acquired the variable and it checks how it can add a version to the write list that is most likely to be the right version at commit time. There are a number of states in other transactions which conflict with this thread:

- When another thread is in the READ_CHECK phase (acquired all values but not yet committed because it has not checked its read list yet) and it has acquired the value, the transaction helps this thread complete its commit phase with recursive helping. There is one state in which this happens: $c = \{25\}$
- When the thread has committed (or aborted) but has not yet released the variables or it has not yet acquired all variables, there is no conflict and the algorithm takes the version it will be after release (if committed) or was before acquired (if not acquired everything yet). The states in which this happens are (for the first and second variable ordered): $r_1 = \{11, 12, 13, 14, 16, 17, 21\}, r_2 = \{13, 14, 17, 18\}$

Which conflict set matters depends on the type of thread. A thread of type one acquires the variables in order and a thread of type two acquires them in reverse order. Given that:

- k_s (k_o) is the transaction type of your own (the other) transaction
- n (m) is the number of threads that have the same (a different) type as the current thread
- $v_{k,i}$ is the time-limiting probability that a thread of type k is in state i

The probability that another thread has been acquired is the probability that one thread is in one of the acquire states ($r \cup c$):

$$\mathbb{P}(acquired|write) = (n-1) \cdot \sum_{i \in r \cup c} v_{k_s,i} + m \cdot \sum_{i \in r \cup c} v_{k_o,i}. \tag{1}$$

The probability that, given there was a conflict, the conflict came from the same type of transaction is:

$$\mathbb{P}(same|write \cup acquired) = \frac{(n-1) \cdot \sum_{i \in r \cup c} v_{k_s,i}}{\mathbb{P}(acquired|write)}. \tag{2}$$

If the variable is acquired, the probability of a conflict (where the other transaction needs to be helped) is:

$$\begin{aligned}\mathbb{P}(conflict|acquired \cup write) = \mathbb{P}(same|write \cup acquired) \cdot \frac{\sum_{i \in c} v_{k_s,i}}{\sum_{i \in r \cup c} v_{k_s,i}} + \\ (1 - \mathbb{P}(same|write \cup acquired)) \cdot \frac{\sum_{i \in c} v_{k_o,i}}{\sum_{i \in r \cup c} v_{k_o,i}}.\end{aligned} \tag{3}$$

We can use these probabilities to determine the actual transition probabilities:

$$\mathbb{P}(no\ conflict|write) = \mathbb{P}(acquired) \cdot (1 - \mathbb{P}(conflict|acquired)). \tag{4}$$

$$\mathbb{P}(conflict|write) = \mathbb{P}(conflict|acquired) \cdot \mathbb{P}(acquired). \tag{5}$$

$$\mathbb{P}(not\ acquired|write) = (1 - \mathbb{P}(acquired)). \tag{6}$$

After the conflict is handled (through recursive helping or finding the right version), the transaction continues with writing the next variable or starting the commit.

8-25 Committing the transaction In the commit phase, the algorithm has to have exclusive access to the variable so it has to acquire it (by replacing the data with a pointer to the transaction). It acquires the variables in a global ordering (same for each thread). If it is certain the transaction is valid (or not), it commits (or aborts) and releases the variables in the same order. This means that it has to recursively help other threads if they currently own the variable they are trying to acquire. It does not matter if they have already committed (or aborted), if the variable is not released then the current thread cannot acquire. Only the first variable can encounter a conflict because all transactions run in full parallelism (assumption), the other variable would already be released by the time it gets there because releasing variables is less work than acquiring them. So the probability of having to help another transaction for the first variable is:

$$\mathbb{P}(help|try\ first\ acquire) = (n-1) \cdot \sum_{i \in r_1 \cup c} v_{k_s,i} + m \cdot \sum_{i \in r_1 \cup c} v_{k_o,i}. \tag{7}$$

The probability that the transaction is helping another transaction of his own type is:

$$\mathbb{P}(same|help) = \frac{(n-1) \cdot \sum_{i \in r_1 \cup c} v_{k_s,i}}{\mathbb{P}(help|try\ first\ acquire)}. \tag{8}$$

After another transaction has been helped, the other transaction has either been successful (which means that this transaction cannot succeed), or it has not been successful. If the helped transaction was successful, this transaction cannot ever succeed because all the variables in the write list were overwritten. The "after success" states (23-24) are there for this reason, the transaction can either encounter another conflict in which case it helps the other transactions and loops back on the same "after success" state, or fails when it notices its write list is no longer current.

If the variable is not owned, there is still a probability that it fails if the variable has been overwritten. If the CAS operation fails because the variable has been written to, the whole transaction has failed (the failed commit and failed release states are there for that purpose). The probability that the variables were overwritten is the probability that after the addition to the write list and before the attempt to acquire, another thread manages to commit their changes. Given:

- $\mathbb{P}_{M,a\to b}$ the set of paths from state a to b in transition matrix M
- $\mathbb{P}_{M,a\to b<t}$ the set of paths from a to one in the set of b that can be reached within t time in transition matrix M (if $a \in b$ then it has to reach b twice)
- t_i the time the algorithm spends in state/path i
- $f(p, t_{max}) = max\{\frac{t_{max}-t_p}{t_{p_1}}, 0\}$ the fraction of time in the first state this path can reach the end state while the time it takes is less than t_{max}
- S the set of all states

The probability that another thread (of type o) does something (meaning it reached states in the set c) between two points (a and b) in the transaction (of type m) is approximately:

$$\mathbb{P}_{event\ between}(m, o, a, b, c) \approx \sum_{p\in\mathbb{P}_{M_m,a\to b}} \sum_{i\in S} \sum_{a\in\mathbb{P}_{M_o,i\to c<t_p}} v_{o,i} \cdot \mathbb{P}(a) \cdot \mathbb{P}(p) \cdot f(a, t_p). \tag{9}$$

The commit phase can fail on both variables because they were overwritten. Since the transaction can acquire the variables in a different order than they were added to the write list, the probability of failure depends on the thread type:

$$\mathbb{P}(no\ failure\ first) = \begin{cases} (1-\mathbb{P}_{event\ between}(k_s, k_s, 4, 9, \{17\}))^{n-1}\cdot & \text{if } k_s = 1 \\ (1-\mathbb{P}_{event\ between}(k_s, k_o, 4, 9, \{17\}))^{m}. & \\ (1-\mathbb{P}_{event\ between}(k_s, k_s, 8, 9, \{17\}))^{n-1}\cdot & \text{if } k_s = 2 \\ (1-\mathbb{P}_{event\ between}(k_s, k_o, 8, 9, \{17\}))^{m}. & \end{cases} \tag{10}$$

$$\mathbb{P}(no\ failure\ second) = \begin{cases} 0. & \text{if } k_s = 1 \\ (1-\mathbb{P}_{event\ between}(k_s, k_s, 4, 9, \{17\}))^{n-1}\cdot & \text{if } k_s = 2 \\ (1-\mathbb{P}_{event\ between}(k_s, k_o, 4, 9, \{17\}))^{m} - \mathbb{P}(failure\ first). & \end{cases} \tag{11}$$

We can use these probabilities to determine the transition probabilities:

$$\mathbb{P}(fail\ on\ first|try\ first\ acquire) = (1-\mathbb{P}(help|try\ first\ acquire)) \cdot (1-\mathbb{P}(no\ failure\ first)). \tag{12}$$

$$\mathbb{P}(acquire|try\ first\ acquire) = (1-\mathbb{P}(help|try\ first\ acquire)) \cdot \mathbb{P}(no\ failure\ first). \tag{13}$$

$$\mathbb{P}(fail\ on\ second|try\ second\ acquire) = 1-\mathbb{P}(no\ failure\ second). \tag{14}$$

$$\mathbb{P}(acquire|try\ second\ acquire) = \mathbb{P}(no\ failure\ second). \tag{15}$$

After the transaction has committed successfully or has failed, it will proceed with releasing all its acquired variables and start another transaction.

26+: Helping other transactions The are four points where another transaction can be helped:

1. adding the first variable to the write list
2. adding the second variable to the write list
3. try to acquire the first ordered variable
4. trying to acquire the first ordered variable after help was successful

The helping states are modeled as a somewhat simplified version of the real commit states. Since the other transaction is helped with the real commit running at full speed, the decision of success or failure of the other commit is not made by the helping commit but by the real commit. Therefore we only decide if we have failed in this model at the end of the help phase. This saves memory and simplifies things.

This probability of success (which determines what comes after the helping phase) derived from the real probability of success given it successfully acquired the first variable:

$$\mathbb{P}(commit\ success|help) = \mathbb{P}(same|help) \cdot \frac{v_{k_s,14}}{v_{k_s,13}} + (1-\mathbb{P}(same|help)) \cdot \frac{v_{k_o,14}}{v_{k_o,13}}. \tag{16}$$

If the helped commit was successful and the transaction was helping another transaction because it encountered a conflict at the first attempt to acquire, then the real transaction cannot succeed anymore. The success state

redirects the transaction to the set of "after success" states (23-24, see above). Otherwise, it continues working on the primary transaction.

The helping commit will always trail the real commit, but there are a few points where the helping commit can skip ahead to the commit/release phase if it detects that the variables were overwritten by the transaction when releasing or by a competing transaction. Given:

- h is the help state before the helping commit
- s is the start state of the help commit
- $\{a \to b, c \to d, \ldots\}$ a set of conditions that can be applied to a transition matrix. It modifies the matrix so each state on the left of the arrow goes to the other state with probability one.
- $\mathbb{P}_{event\ between\ con}(m, o, cm, co, a, b, c)$ is the probability that another thread (of type o) does something (meaning it reached states in the set c) between two points (a and b) in the transaction (of type m) with the condition sets cm and co applied to the matrices of m and o respectively.
- $tcd_{k,i} = \frac{v_{k,i}}{\sum_{i \in r_1 \cup c} v_{k,i}}$ is the time dependent distribution of the committing transaction at the point the transaction is in the helping state. We assume that the helped transaction has this time-limiting distribution when it starts being helped at h instead of v.

The probability that the helping commit skips to the release at the first acquire is:

$$\mathbb{P}(help\ commit|help\ try\ acquire\ first) = \mathbb{P}(same|helping) \cdot \mathbb{P}_{event\ between}(k_s, k_s, h, s, \{17, 21\}) + (1 - \mathbb{P}(same|helping) \cdot \mathbb{P}_{event\ between}(k_s, k_o, h, s, \{17, 21\})). \tag{17}$$

The probability that the helping commit skips to the release at the second acquire is the probability that main commit has released the second variable or that another transaction has overwritten the second variable:

$$\mathbb{P}(release|help\ try\ acquire\ second) = (\mathbb{P}(same|helping) \cdot \mathbb{P}_{event\ between\ con}(k_s, k_s, \{s, s+1\}, \{\}, h, s+2, \{18\}) + (1 - \mathbb{P}(same|helping)) \cdot \mathbb{P}_{event\ between\ con}(k_s, k_o, \{s, s+1\}, \{\}, h, s+2, \{18\})) \tag{18}$$

$$\mathbb{P}(overwritten|help\ try\ acquire\ second) = \mathbb{P}(same|helping) \cdot \frac{v_{k_s,16}}{v_{k_s,12}} + (1 - \mathbb{P}(same|helping)) \cdot \frac{v_{k_o,16}}{v_{k_o,12}}. \tag{19}$$

$$\mathbb{P}(help\ commit|help\ try\ acquire\ second) = 1 - (1 - \mathbb{P}(release|help\ try\ acquire\ second)) \cdot (1 - \mathbb{P}(overwritten|help\ try\ acquire\ second)). \tag{20}$$

3.3 DSTM Model

The DSTM model is quite a bit simpler than the OSTM model on the surface. It has 15 states. There are a number of states where it tries to acquire a variable by switching the locator and there are a number of states where it checks if it was not aborted. These verification states give rise to a lot of complexity because the model has to check each path for the probability it was aborted.

Acquiring The algorithm tries to acquire the variable by executing a CAS operation. First it checks if it can acquire the variable, then it builds a new locator to replace the old one and executes the CAS operation. The acquire can fail at two points. If the variable is owned at the check, the algorithm asks the contention manager if it can abort[11]. Otherwise, it tries to acquire. This can also fail if another transaction acquires the variable in the period it takes to prepare the locator and execute the CAS operation. The probability that a variable is not owned is the probability that none of the other transactions is in a state where it is has acquired the variable[12]. Given that $c_1 = \{5, 6, \ldots, 14\}$ and $c_2 = \{11, 12, 13, 14\}$ are the states in the current transaction where the first and second variable are acquired, the probability of no conflict is:

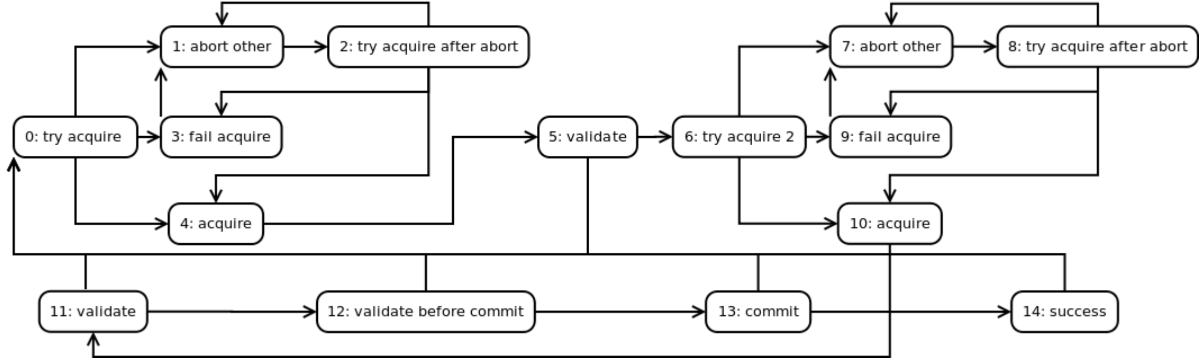
$$\mathbb{P}(no\ conflict|try\ acquire\ first) = (1 - \sum_{i \in c_1} v_{k_s,i})^{n-1} \cdot (1 - \sum_{i \in c_2} v_{k_o,i})^m. \tag{21}$$

$$\mathbb{P}(no\ conflict|try\ acquire\ second) = (1 - \sum_{i \in c_2} v_{k_s,i})^{n-1} \cdot (1 - \sum_{i \in c_1} v_{k_o,i})^m. \tag{22}$$

[11] which is always allowed when using the aggressive contention manager

[12] Multiple transactions can think they own the variable since the algorithm is obstruction-free, but only one truly owns it

Figure 6: Discrete time Markov model of the DSTM algorithm (model created by myself)



The probability that the transaction can acquire is the probability that no other transaction acquires in the time it takes to prepare the locator and execute the CAS operation. Given:

- $c_{free\ 1} = \{0 \to 4, 2 \to 4\}, c_{free\ 2} = \{6 \to 10, 8 \to 10\}$ the set of conditions that indicate the first and second variable are free
- $a_1 = 4, a_2 = 10$ the acquire states for the first and second variable

The probability that the transaction is not interrupted is:

$$\begin{aligned} \mathbb{P}(can\ acquire\ first|no\ conflict) =& (1 - \mathbb{P}_{event\ between\ con}(k_s, k_s, \{\}, c_{free\ 1}, a_1, a_1, \{a_1\}))^{n-1} \cdot \quad (23) \\ & (1 - \mathbb{P}_{event\ between\ con}(k_s, k_o, \{\}, c_{free\ 2}, a_1, a_1, \{a_2\}))^{m}. \\ \mathbb{P}(can\ acquire\ second|no\ conflict) =& (1 - \mathbb{P}_{event\ between\ con}(k_s, k_s, \{\}, c_{free\ 2}, a_2, a_2, \{a_2\}))^{n-1} \cdot \quad (24) \\ & (1 - \mathbb{P}_{event\ between\ con}(k_s, k_o, \{\}, c_{free\ 1}, a_2, a_2, \{a_1\}))^{m}. \end{aligned}$$

Given these probabilities, we can calculate the transition probabilities of the first attempt to acquire a variable:

$$\mathbb{P}(acquire|try\ acquire) = \mathbb{P}(can\ acquire|no\ conflict) \cdot \mathbb{P}(no\ conflict|try\ acquire). \quad (25)$$

$$\mathbb{P}(fail\ acquire|try\ acquire) = (1 - \mathbb{P}(can\ acquire|no\ conflict)) \cdot \mathbb{P}(no\ conflict|try\ acquire). \quad (26)$$

$$\mathbb{P}(abort\ other|try\ acquire) = 1 - \mathbb{P}(no\ conflict|try\ acquire). \quad (27)$$

The probability that a conflict is with a transaction of your own type is approximately[13]:

$$\mathbb{P}(own\ type|conflict\ first) = \frac{(n-1) \cdot v_{k_s,a_1}}{(n-1) \cdot v_{k_s,a_1} + m \cdot v_{k_o,a_2}}. \quad (28)$$

$$\mathbb{P}(own\ type|conflict\ second) = \frac{(n-1) \cdot v_{k_s,a_2}}{(n-1) \cdot v_{k_s,a_2} + m \cdot v_{k_o,a_1}}. \quad (29)$$

If a transaction has failed to acquire (specifically the CAS operation failed), it will almost always encounter another conflict when it retries. The probability that it will not encounter a conflict after failure is the probability that a third transaction starts acquiring in the time between the end of the interrupting acquire and the end of the failed CAS operation. The probability that this happens is so small[14] that the model does not incorporate the possibility. Instead we assume it will always encounter a conflict after failure and thus will start aborting the conflicting transaction.

[13] Since we cannot be sure when exactly the thread has really acquired the variable, we estimate it by weighing it with the fraction of acquires

[14] It was never encountered during verification except when the transaction was preempted at the point

After a transaction has aborted, it will try again. While it is trying to acquire, another thread could snatch up the variable. We assume that the aborted thread does not try this because it takes quite a while for a transaction to realize it is aborted and after a restart acquire a variable.

Given that:

- s is the state where the acquire process or the check process can be interrupted by the acquisition of a variable by another transaction. The states where this happens are the "try acquire after abort" states (2 and 8) and the actual acquire states (4 and 10)
- The aborted thread cannot reacquire because the time it takes to realize you have been aborted and then acquire the variable is larger than the time to try again after an abort.

The probability that another transaction does not acquire the variable during the attempt to acquire or the check if the variable is owned is:

$$\begin{aligned}\mathbb{P}(not\ acquired|first\ abort) =& (1-\mathbb{P}_{event\ between\ con}(k_s,k_s,\{\},c_{free\ 1},s,s,\{a_1\}))^{\max\{0,n-2\}}\cdot \\ & (1-\mathbb{P}_{event\ between\ con}(k_s,k_o,\{\},c_{free\ 2},s,s,\{a_2\}))^{\max\{0,m-1\}}\cdot \\ & (1-\mathbb{P}(own\ type|conflict\ first)\cdot\mathbb{P}_{event\ between\ con}(k_s,k_o,\{\},c_{free\ 2},s,s,\{a_2\})- \\ & (1-\mathbb{P}(own\ type|conflict\ first))\cdot\mathbb{P}_{event\ between\ con}(k_s,k_s,\{\},c_{free\ 1},s,s,\{a_1\})).\end{aligned} \tag{30}$$

$$\begin{aligned}\mathbb{P}(not\ acquired|second\ abort) =& (1-\mathbb{P}_{event\ between\ con}(k_s,k_s,\{\},c_{free\ 2},s,s,\{a_2\}))^{\max\{0,n-2\}}\cdot \\ & (1-\mathbb{P}_{event\ between\ con}(k_s,k_o,\{\},c_{free\ 1},s,s,\{a_1\}))^{\max\{0,m-1\}}\cdot \\ & (1-\mathbb{P}(own\ type|conflict\ second)\cdot\mathbb{P}_{event\ between\ con}(k_s,k_o,\{\},c_{free\ 1},s,s,\{a_1\})- \\ & (1-\mathbb{P}(own\ type|conflict\ second))\cdot\mathbb{P}_{event\ between\ con}(k_s,k_s,\{\},c_{free\ 2},s,s,\{a_2\})).\end{aligned} \tag{31}$$

Given those probabilities, we can derive the transition probabilities for the situation after an abort:

$$\begin{aligned}\mathbb{P}(acquire|try\ acquire\ after\ abort) =& \mathbb{P}(not\ acquired\ try\ acquire|first\ abort)\cdot \\ & \mathbb{P}(not\ acquired\ acquiring|first\ abort).\end{aligned} \tag{32}$$

$$\begin{aligned}\mathbb{P}(fail\ acquire|try\ acquire\ after\ abort) =& \mathbb{P}(not\ acquired\ try\ acquire|first\ abort)\cdot \\ & (1-\mathbb{P}(not\ acquired\ acquiring|first\ abort)).\end{aligned} \tag{33}$$

$$\mathbb{P}(abort\ other|try\ acquire\ after\ abort) = 1-\mathbb{P}(not\ acquired\ try\ acquire|first\ abort). \tag{34}$$

Validation points At different points the transaction validates if another transaction has not aborted him. Between those points other transactions can abort the transaction because it conflicts with what they want to do. Other transactions will be affected by the acquisition of the variables. Sometimes the transaction acquires a variable between two validation points, changing the number of states that abort the transaction and the conditions imposed on the other transaction midway through the path the transaction takes.

There are four checkpoints in this situation:

- validation after the first variable is acquired
- validation after the second variable is acquired
- validation before committing
- CAS operation to committed

Given:

- $\mathbb{P}_{event\ between\ tdp}(m,o,cm,co_s,co_e,s,a,b,c_s,c_e)$ is the probability that another thread (of type o) does something (meaning it reached states in the set c_s or c_e depending on which terminal set applies at this point) between two points (a and b) in the transaction (of type m) with the condition sets cm and co applied to the matrices of m and o respectively. Before point s is reached in thread o, the condition set co_s (as co) and the terminal set c_s are applied to the thread of type o. After point s has been reached, the condition set co_e (as co) and the terminal set c_e (as c) are applied to the thread of type o

- $c_{ofs} = \{0 \to 1, 2 \to 1, 5 \to 0, 11 \to 0, 12 \to 0, 13 \to 0\}$ are the conditions imposed on a transaction of the same type if the first variable is owned
- $c_{ofo} = \{6 \to 7, 8 \to 7, 11 \to 0, 12 \to 0, 12 \to 0, 13 \to 0\}$ are the conditions imposed on a transaction of another type if the first variable is owned
- $c_{oa} = \{0 \to 1, 2 \to 1, 5 \to 0, 6 \to 7, 8 \to 7, 11 \to 0, 12 \to 0, 13 \to 0\}$ are the conditions imposed if all the variables are owned

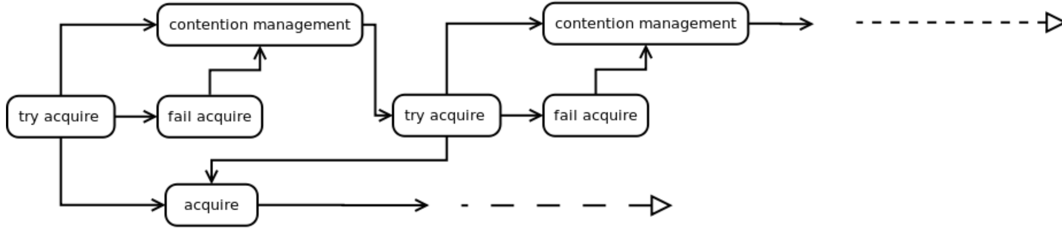
The probability that the transaction is aborted between the checkpoints is:

$$\begin{aligned}
\mathbb{P}(no\ abort|first\ validation\ check) =& (1 - \mathbb{P}_{event\ between\ con}(k_s, k_s, \{\}, c_{ofs}, 5, 5, \{1\}))^{n-1} \cdot \\
& (1 - \mathbb{P}_{event\ between\ con}(k_s, k_o, \{\}, c_{ofo}, 5, 5, \{7\}))^{m}. && (35)\\
\mathbb{P}(no\ abort|second\ validation\ check) =& (1 - \mathbb{P}_{event\ between\ tdp}(k_s, k_s, \{\}, c_{ofs}, c_{oa}, 10, 6, 11, \{1\}, \{1, 7\}))^{n-1} \cdot && (36)\\
& (1 - \mathbb{P}_{event\ between\ tdp}(k_s, k_o, \{\}, c_{ofo}, c_{oa}, 10, 6, 11, \{7\}, \{1, 7\}))^{m}.\\
\mathbb{P}(no\ abort|third\ validation\ check) =& (1 - \mathbb{P}_{event\ between\ con}(k_s, k_s, \{\}, c_{oa}, 12, 12, \{1, 7\}))^{n-1} \cdot && (37)\\
& (1 - \mathbb{P}_{event\ between\ con}(k_s, k_o, \{\}, c_{oa}, 12, 12, \{1, 7\}))^{m}.\\
\mathbb{P}(no\ abort|fourth\ validation\ check) =& (1 - \mathbb{P}_{event\ between\ con}(k_s, k_s, \{\}, c_{oa}, 13, 13, \{1, 7\}))^{n-1} \cdot && (38)\\
& (1 - \mathbb{P}_{event\ between\ con}(k_s, k_o, \{\}, c_{oa}, 13, 13, \{1, 7\}))^{m}.
\end{aligned}$$

3.3.1 Polite contention management

The polite contention management can be "plugged in" by adding n (the parameter of the polite contention manager) sets of states before each try attempt that mirror the real try attempt (with regard to transition probabilities) but instead of aborting the conflicting transaction, they wait a fixed amount of time in the contention management state before retrying. If none of the attempts succeed, they can abort the conflicting transaction at the end of the chain.

Figure 7: DSTM Polite Contention Management States



There are a couple of small changes with regard to the probability calculations:

- the conditions applied to the path-dependent probability take the extra polite contention manager states into account
- the conflict probabilities also include the probability of being in one of the contention manager states

The added states are computationally quite expensive. To check if another transaction was aborted in these states, we have to check each path to the next validation point. Therefore we can only calculate results for a limited number of configurations.

3.4 Verification of the model

To verify the model, we implemented our own STM system and benchmarked it with the situation described above. Using a tracing system to track each phase of the algorithm, we can check our model and the predictions it makes. For each state, we can measure the amount of time it takes to execute the state (see Attachment 3) and the decision

it makes after the state is done. Unfortunately, we do not have the required hardware (with a lot of processing cores) to verify our conclusions with the real world implementation. The only useful data we got from the real world implementation is actual data on the timing of each phase of the algorithm. The low number of cores in the test machine caused too much noise in the data to accurately check the model predictions.

So instead of a real live implementation, we use a discrete event simulation which simulates the $n + m$ threads running the algorithm to check the predictions the model makes. The simulation runs the algorithm as if it had full access to as many cores as it needs. It has the following properties:

- The states the simulation mirror the states of the Markov model, each state is simulated as a discrete event that takes a fixed amount of time to execute. After the state is done, it makes a decision based on the shared state that determines the next phase of the algorithm.
- The simulation starts each thread sequentially, the time between start times is exponentially distributed with $\lambda = \frac{1}{5000}$, the expected interval time is about four times the best case latency for each thread
- The simulation starts measuring latency and success/failure events after the last thread has started, the time between those events is also exponentially distributed with the same parameter λ

4 Results

4.1 OSTM Results

First, we check if the model makes accurate predictions compared to the simulation. As you see in Figure 8, the predictions the model makes about the OSTM algorithm are too optimistic. The assumptions the model made do not hold, so we have to use the results of the simulation to draw conclusions about how the algorithm handles the tested situation. The success rate of the algorithm (see Figure 9) decreases with a rate that is better than linear with the number of threads independent of the type of the threads that are running. This means the algorithm is immune to the effects of the order the variables are acquired, which is exactly what you would expect from an algorithm that orders the acquisition of the variables. The latency of the algorithm (see Figure 10) stays approximately the same independently of the number of variables. It seems that the lock-freedom property of the OSTM algorithm holds and because it has this property, it is able to scale with a constant factor. The latency of this algorithm is somewhat on the high side, but even when the contention rises, the throughput stays approximately constant. This means that the OSTM algorithm has the best scaling factor that can be achieved according to Amdahl's law.

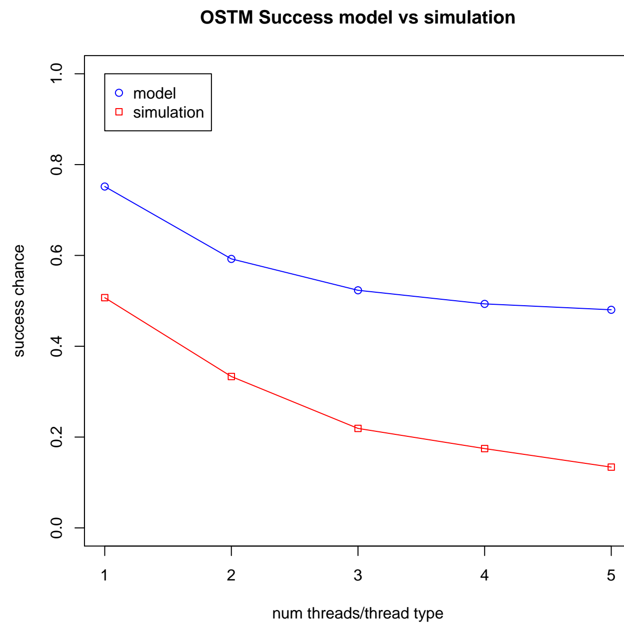


Figure 8: Comparison between the predicted performance and the performance of the simulation for the OSTM algorithm

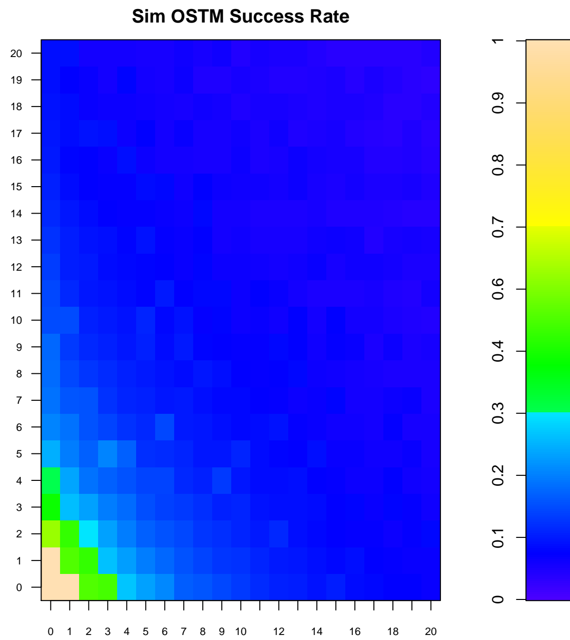


Figure 9: Success rate of the OSTM algorithm

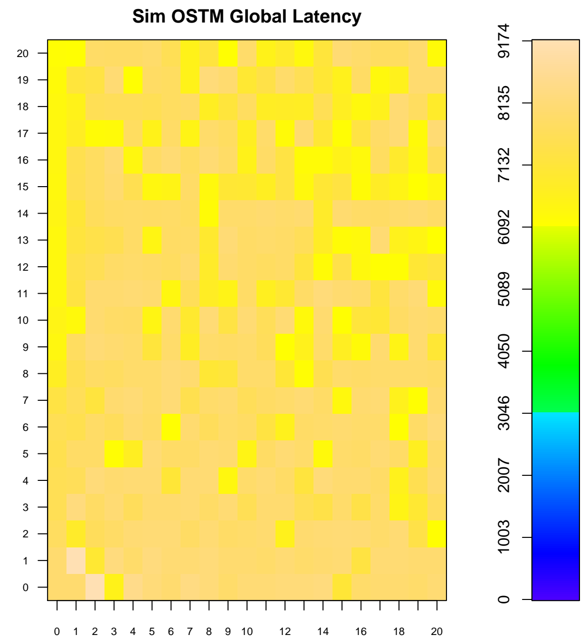


Figure 10: Global latency of the OSTM algorithm

4.2 DSTM Results

4.2.1 Aggressive contention management

The model again fails to predict the algorithm accurately (see Figure 11). The aggressive contention manager is so bad at managing this situation that both the simulation and the model will eventually predict a success rate of zero. The predicted and simulated performance do not converge not because the model predicted the situation correctly but because it cannot predict anything other than zero performance.

As expected, the DSTM algorithm with the aggressive contention management performs badly (see Figure 12) in all situations where more than one thread is running. If there is any sort of contention or conflict between threads, the success rate drops to almost zero. The latency (see Figure 13) of the algorithm is just as bad as the success rate. While there are some small peaks in the performance (all due to a very small number of successful transactions), the latency is extremely high in all situations with more than one thread running. This pretty much shows that the aggressive contention manager should not be used in any situation, it collapses when there is any sort of conflict encountered.

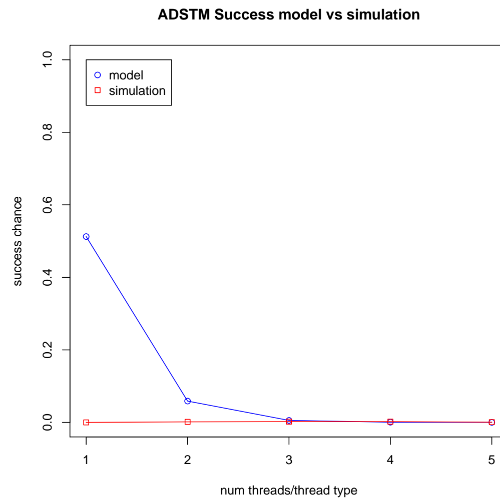


Figure 11: Comparison between the predicted performance and the performance of the simulation for the ADSTM algorithm

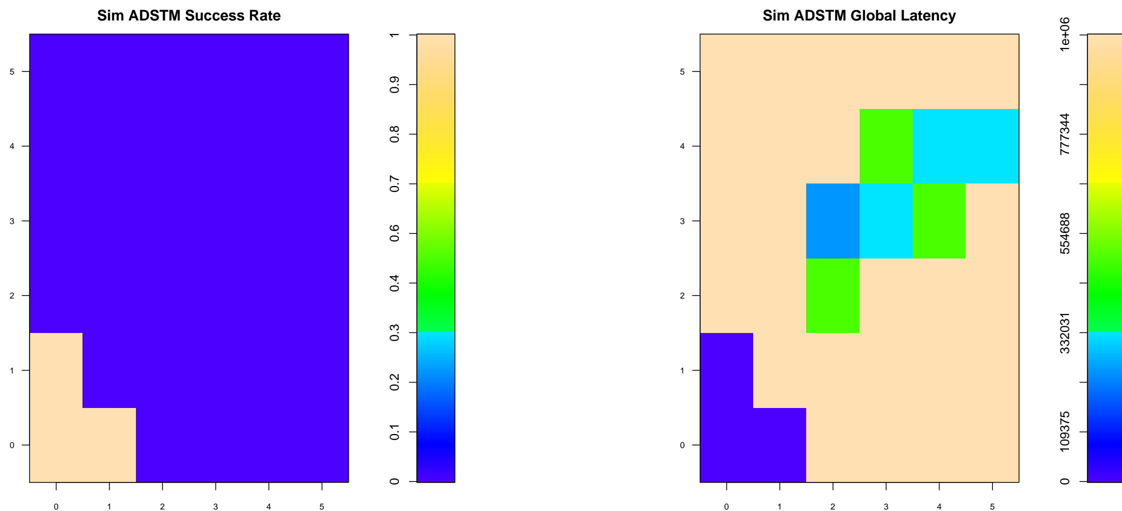


Figure 12: Success rate of the ADSTM algorithm

Figure 13: Global latency of the ADSTM algorithm

4.2.2 Polite contention management

The model fails to predict the system accurately again (see Figure 14), it is too optimistic in its prediction. Now we look at how different sets of parameters for the polite contention management system influence the results of the system. There are two parameters that matter:

- The "politeness" parameter that influences how many times the algorithm has to wait before it can abort, we simulated the system with a politeness parameter of one to nine with a step size of two
- The "delay" parameter that influences the amount of the time the algorithm has to wait, we simulated the system with a delay of 125 and 250 nanoseconds (the time duration of some of the shorter states of the algorithm, it seems reasonable to choose the parameter as something that is proportional to the rest of the algorithm)

As you can see in Figure 15, the "politeness" parameter has a lot of influence on situation. When the algorithm can wait longer before aborting conflicting threads, it can handle more complicated scenarios. The polite contention manager is clearly better at handling the simple situations where there is only one type of thread running (look at the edges of the graph for those situations). But if you look at the latency (see Figure 16) then it becomes clear how bad the performance becomes when the polite contention manager cannot handle the number of conflicts or the ordering issues caused by the different types of threads.

Timing can also be a significant factor. If you look at Figure 17 and 18, you see the higher timing factor ensures that the algorithm can handle the situation better in combination with a high politeness parameter. But, with a politeness parameter value of eleven the algorithm becomes somewhat unstable. It seems that the combination of high parameter values enable the algorithm to wait much longer than usual before aborting conflicting transactions. This can lead to large instability when the algorithm encounters a lot of situations with high contention or ordering issues. This mean parameter values that are too high can cause instability.

There are a number peaks in the data where the polite contention manager seems to perform very badly. This is related to the inherent instability of the situation. Because the algorithm is fundamentally not equipped to handle this particular situation, it can happen that multiple threads get in a loop where they constantly block and abort each other without either succeeding (livelock). While this livelock situation is not stable, it does mean that the system is stuck for a long period of time.

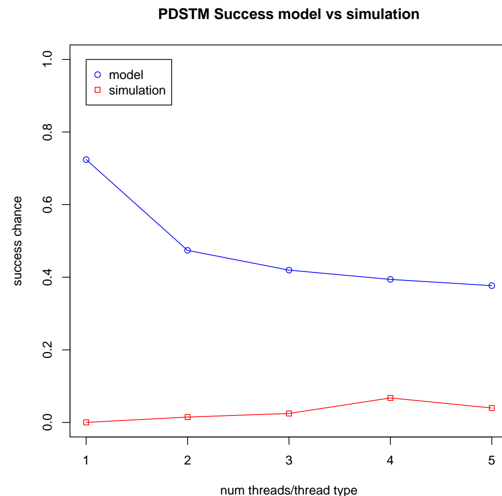


Figure 14: Comparison between the predicted performance and the performance of the simulation for the Polite DSTM algorithm

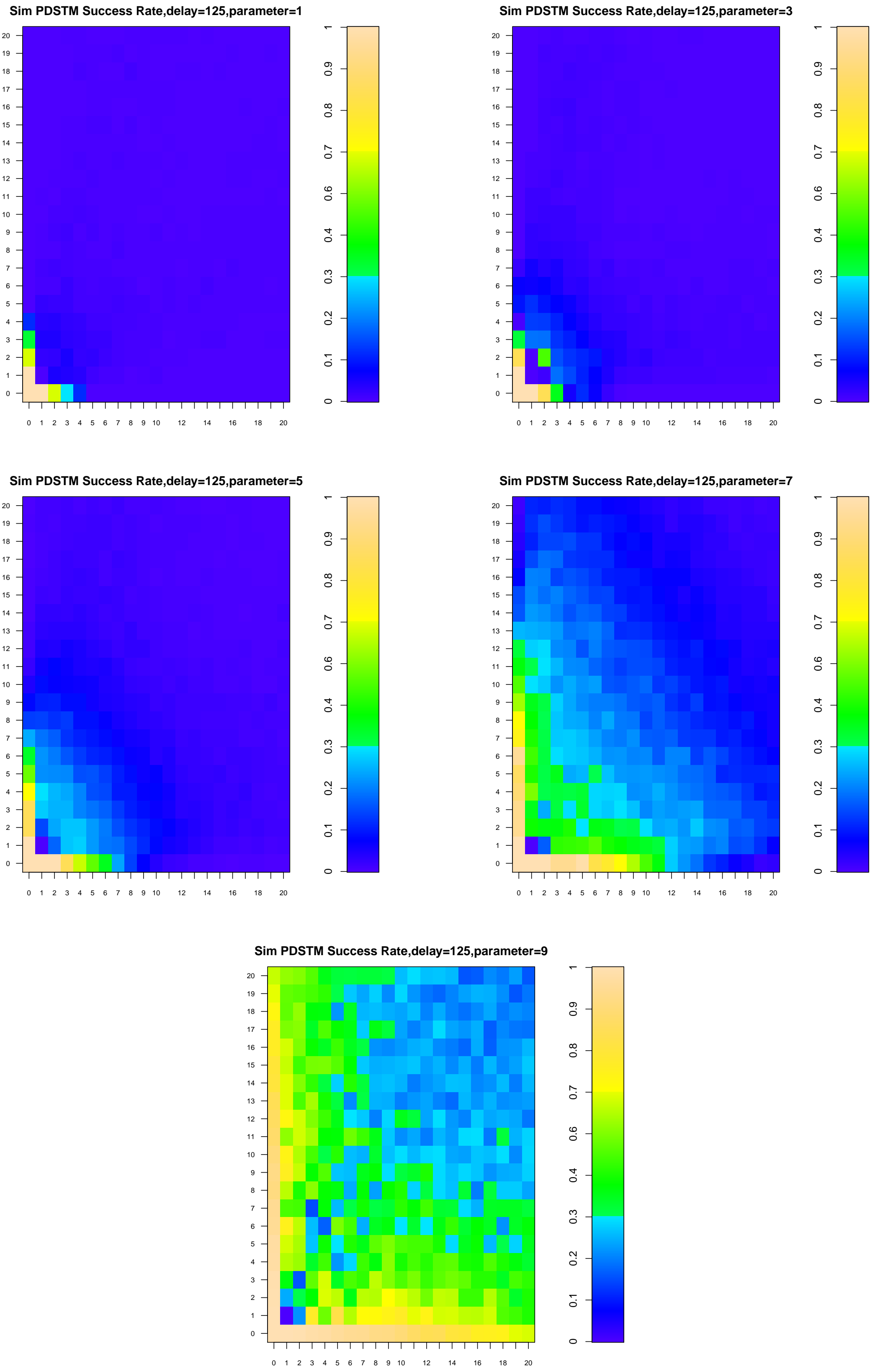


Figure 15: Success rate for Polite DSTM with time parameter 125

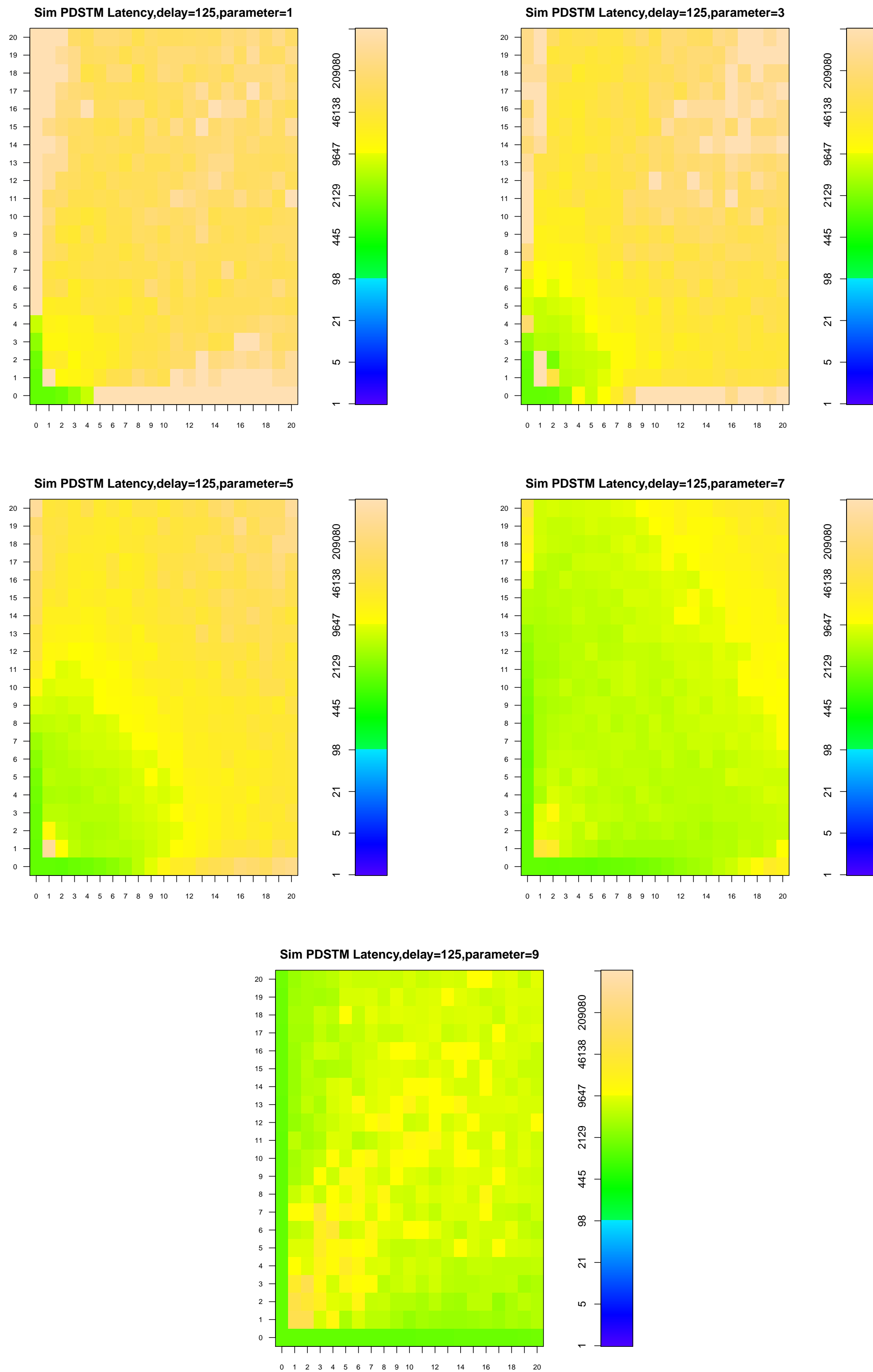


Figure 16: Global latency for Polite DSTM with time parameter 125

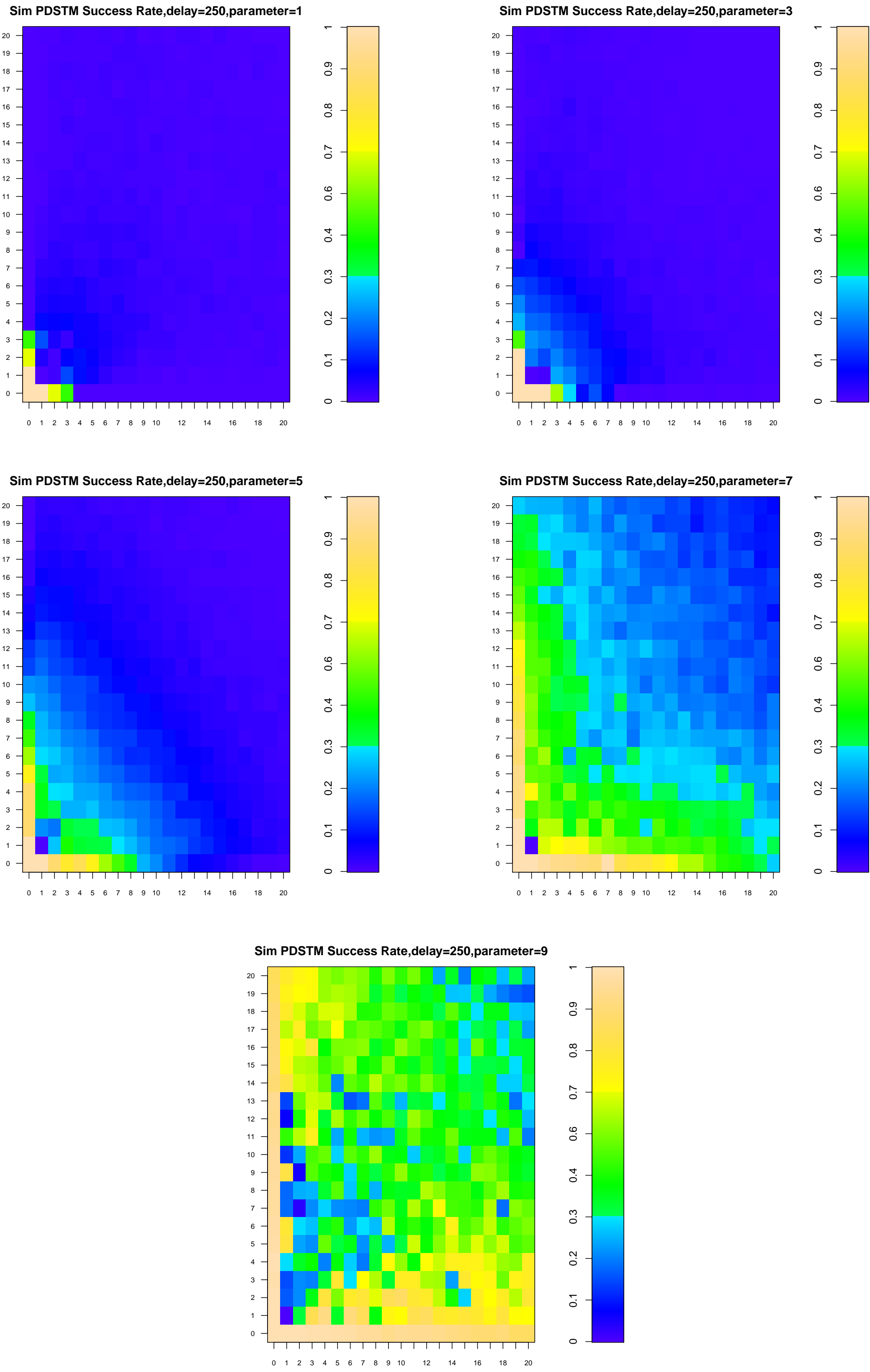


Figure 17: Success rate for Polite DSTM with time parameter 250

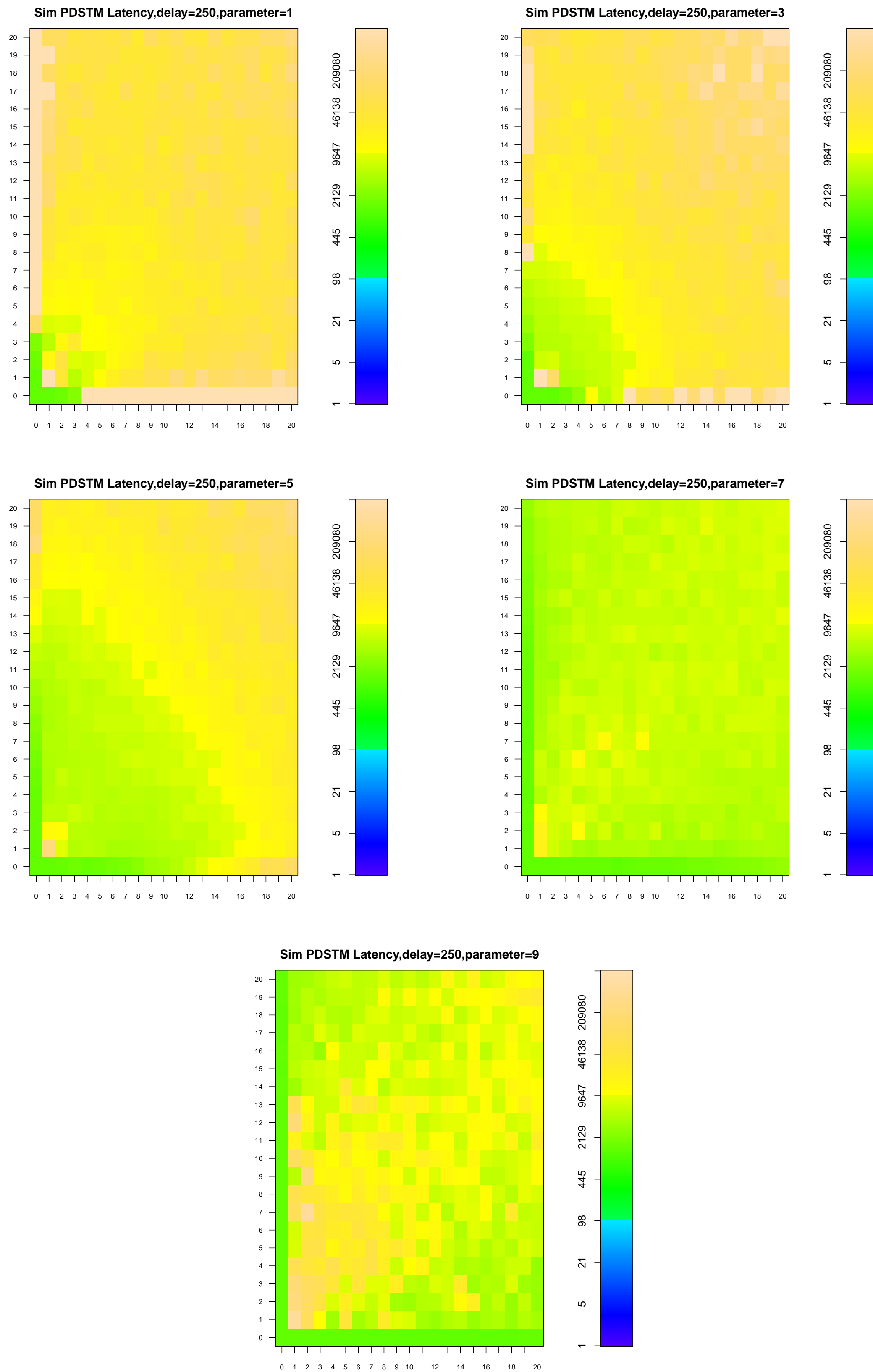


Figure 18: Global latency for Polite DSTM with time parameter 250

5 Conclusion

For this paper we compared the DSTM and OSTM algorithms by modeling them with Markov chains and we used simulation to verify the conclusions.

5.1 The model

The model was unable to predict the performance of the situation accurately. Fortunately we had the results of the simulation to analyze. The main cause of the inaccuracy in the model is that the assumption that the time-limited distribution of other threads is independent of the current state is false. Each algorithm has multiple points of failure, and these points are not independent. If the algorithm has survived the first "checkpoint" (each algorithm has parts where it checks if it can still continue), it is more likely to fail the next test. The model was unable to take these dependencies into account, which caused it to be too optimistic. There was no other obvious way to model the system, and since the same technique was applied successfully in other studies [11] it seemed like a good choice. It is nevertheless good to see in which cases this technique does not work. Simulation or real world benchmarking is a better alternative to predict performance in these systems.

5.2 The algorithms

It seems that the OSTM algorithm is the "safer" alternative that can handle the most difficult situations without anything affecting the latency of the algorithm. Because OSTM orders the acquisition operations it executes, it can handle differently ordered variable writes and high contention without trouble. But the lock-freedom property of OSTM can be expensive. In "normal" situations (without ordering issues) the Polite DSTM algorithm with reasonable parameter values has a much lower latency on average. Because OSTM uses a late acquire strategy, it has a much higher probability that the transaction has to be restarted than when Polite DSTM tries to acquire variables in a situation with one type of thread.

The DSTM algorithm has quite a bit more difficulty dealing with differently ordered variable acquisition and contention. With the right parameter values, it can exceed the performance of the OSTM algorithm in the normal situations. But because it is only obstruction-free, the situations where it encounters transactions that write in a different order can cause instability in some situations. If these parameter values are too low or the situation is unstable, then the latency can become extremely high. This means that in some specific situations, the DSTM algorithm can easily cause speed issues in your program. Other contention managers might provide a somewhat better performance, but none of these heuristics can overcome the weakness of DSTM with regards to the ordering of variable acquisitions. Any contention management algorithm that would negotiate among competing threads would have a larger overhead than the OSTM algorithm itself and thus would be pointless. The DSTM algorithm can offer great performance, but it does require quite a bit of tuning to find the right combination of parameters and contention management heuristics. It will never be completely safe, but in most situations it is a better choice than the OSTM algorithm. If the heuristics and parameters are tuned correctly and livelock situations can be avoided, then the DSTM algorithm performs better than the OSTM algorithm. If this is not the case, then the OSTM algorithm is a good and safe choice.

References

[1] R. Ennals. Software transactional memory should not be obstruction-free. 2006.

[2] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25, 2007.

[3] S. Halloway. *Programming Clojure*. Pragmatic Bookshelf, 2009.

[4] T. Harris and K. Fraser. Language support for lightweight transactions. *OOPSLA*, 2003.

[5] T. Harris, S. Marlow, S. Jones, and M. Herlihy. Composable memory transactions. 2006.

[6] A. Heindl and G. Pokam. An analytic framework for performance modeling of software transactional memory. *Computer Networks*, 53.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, 2003.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, 2008.

[9] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. 2004.

[10] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. *PODC*, 2005.

[11] L. A. Tomek and K. S. Trivedi. Fixed point iteration in availability modeling.

6 Attachment 1: DSTM implementation

Here is a simple implementation of DSTM [7]:

Listing 6: DSTM pseudo code

```
class Handle:
  def Handle():
    self.start = null

class Locator:
  def Locator(transaction, old, new):
    self.transaction = transaction
    self.old = old
    self.new == new

class Transaction:
  def Transaction():
    self.read_list = {}
    self.status = ACTIVE

def get_data(handle):
  switch(handle.start.transaction.status):
    case ABORTED:
      return handle.start.old
    case ACTIVE:
    case COMMITTED:
      return handle.start.new

def open_read(handle, transaction, contention_manager):
  # tell the contention manager an attempt to open a handle has started
  contention_manager.signal_read()
  while(true):
    locator = handle.start
    if(locator.transaction == transaction):
      data = get_data(handle)
      break
    else if(locator.transaction.status == ACTIVE):
      # conflict, let the contention manager decide
      if(contention_manager.should_abort(transaction, locator.transaction)):
        CAS(locator.transaction.status, ACTIVE, ABORTED)
    else:
      data = get_data(handle)
      if(transaction.read_list[handle]):
        break
      else:
          transaction.read_list[handle] = data
          break

  verify(transaction)
  return data

def verify(transaction):
  for(key in transaction.read_list):
    if(get_data(key) != transaction.read_list[key]):
```

```
      # read list invalid, abort!
      CAS(transaction.status, ACTIVE, ABORTED)
      throw AbortedException
  if(transaction.status == ABORTED):
    throw AbortedException

def open_write(handle, transaction, contention_manager):
   # signal the contention manager that an attempt to open a handle has started
   contention_manager.signal_write()
   while(true):
     locator = handle.start
     if(locator.transaction == transaction)
       verify(transaction)
       return get_data(handle)
     else if(locator.transaction.status == ACTIVE):
       # conflict, let the contention manager decide
       if(contention_manager.should_abort(transaction, locator.transaction)):
         CAS(locator.transaction.status, ACTIVE, ABORTED)
     else:
       data = get_data(handle)
       new_locator = Locator(transaction, data, data.clone)
       if(CAS(handle.start, locator, new_locator) == new_locator):
         verify(transaction)
         return new_locator.new

def commit(handle, transaction):
  verify(transaction)
  return CAS(transaction.status, ACTIVE, COMMITTED) == COMMITTED
```

7 Attachment 2: OSTM implementation

Here is a simple implementation of OSTM [2]:

Listing 7: OSTM pseudo code

```
# wrapper around object
class Handle:
  def Handle(object):
    self.object = object

# represent entry in transaction read/write list, multiple versions possible
class Locator:
  def Locator():
    self.old = null # old version (before successful commit)
    self.new = null # new version (after commit)
    self.object = null # reference to the handle or transaction if acquired

# Represents transaction
class Transaction:
  def Transaction():
    self.write_list = [] # always sorted
    self.read_list = []

def get_data(transaction, handle):
  data = handle.object
  if(is_transaction(data)): # if this is a transaction, then it owns the data
    other_transaction = data
    other_entry = other_transaction.write_list.search(handle)
    if(other.status == READ_CHECK):
      # transactions should have a fixed order, for example by pointer value,
      # to prevent two transactions both helping each other
      if(transaction.status != READ_CHECK or transaction > other):
        other_transaction.commit # help other
      else:
        CAS(other.status, READ_CHECK, FAILED) # help self by aborting the other
    if(other.status == SUCCESSFULL):
      data = handle.new
    else:
      data = handle.old
```

```
  return data

def open_write(transaction, handle):
  entry = transaction.write_list.search(handle) # check the current write list
  if(exists entry):
    return entry.new # already opened for writing, return the new version

  entry = transaction.read_list.search(handle) # check the current read list
  if(entry):
    transaction.read_list.remove(handle) # we are now writing
  else:
    entry = Entry()
    entry.object = handle
    entry.old = obj_read(handle)

  entry.new = entry.old.clone
  transaction.write_list.insert(entry)

def open_read(transaction, handle):
  entry = transaction.read_list.search(handle) || transaction.write_list.search(handle)
  if(exists handle):
    return entry.new # already opened for reading/writing
  entry = Entry()
  entry.object = handle
  entry.old = transaction.get_data(handle)
  entry.new = entry.old
  transaction.read_list.insert(entry)
  return entry.new

def commit(transaction):
  desired_status = FAILED
  # acquire in order (write list must be sorted on a global ordering)
  for(entry in transaction.write_list)
    while((data = CAS(handle.object.data, handle.old, transaction)) != handle.old): # try acquiring
      if(data == transaction):
        break; # we own this object, because an other transaction 'helped' us or the CAS succeeded
      else if(not is_transaction(data)): # someone else wrote to this data
        goto decisision_point
      else # conflict with other transaction who is committing
        data.commit # help the conflicting transaction
  CAS(transaction.status, UNDECIDED, READ_CHECK) # go to the read phase
  for(entry in transaction.read_list)
    if(transaction.get_data(entry.object) != entry.old):
      goto decision_point
  desired_status = SUCCESSFULL # we have successfully passed both read and write checks
decision_point:
  # try to set to desired status
  while( ((status = transaction.status) != FAILED) and (status != SUCCESSFULL)):
    CAS(transaction.status, status, desired_status)
  for(entry in transaction.write_list): #release all acquired resources
    CAS(entry.object.data, transaction, (status == SUCCESSFULL) ? entry.new : entry.old)
  return (status == SUCCESSFULL)
```

8 Attachment 3: Timing data

State	Time (nanoseconds)
Try Acquire	248
Abort Conflict	286
Abort Try Acquire	151
Fail Acquire	1065
Acquiring	184
Validate	101
Commit	120

Table 1: DSTM Timing Data (from benchmarks real world implementation)

State	Time (nanoseconds)
Write	1200
No conflict owned	602
Conflict resolve	511
No conflict	110
Try Acquire Write	602
Helping	981
Acquiring	365
Done Commit	365
Release	602
Read Check	365

Table 2: OSTM Timing Data (from benchmarks real world implementation)