Synchronization phases (to speed up transactional memory)

Fabian Landau, Johannes Schneider, Roger Wattenhofer
landauf@ee.ethz.ch,{jschneid, wattenhofer}@tik.ee.ethz.ch
Computer Engineering and Networks Laboratory
ETH Zurich, 8092 Zurich, Switzerland

Abstract

We introduce the *Sync-Phase* technique for software transactional memory. Time is divided into two different alternating phases: computation phases and commit (synchronization) phases. Transactions are only allowed to commit during a commit phase, which allows execution without validation (or the need for cache synchronization) during computation phases. Phases can be determined through time or other means. The technique scales well and it is applicable, whenever a single commit of a transaction triggers certain tasks, e.g. validations of a read set or extending the validity of objects for time-based transactional memory.

For evaluation we implemented the *Sync-Phase* STM technique for DSTM and compared it to other techniques implemented based on DSTM, i.e. the original version and one that uses a commit counter to reduce the number of validations. Benchmarks show that *Sync-Phase* STM allows much faster execution of transactions in most scenarios and scales better than existing approaches relying on a single commit counter.

1 Introduction

Handling concurrency has become an unavoidable burden to leverage the omnipresent multi-core processors. In most cases parallelism is limited and some form of synchronization among cores is needed. Many pitfalls like race conditions and deadlocks that lead to hard to find errors make the life of a programmer difficult. If a software engineer opts for coarse-grained locking which is less troublesome, the code might become very slow. To (partially) address this issue a concept was taken from database systems: *transactions*. They contain an atomic database query. A transaction either completes successfully, or aborts completely, leaving the database in an unmodified state. Transferred to programming languages, a transaction contains arbitrary code which executes atomically. If no conflicts occur, the transaction *commits* its result to the program's memory, otherwise the transaction *aborts* and has no effect on the data accessed by other transactions. An aborted transaction may be restarted until it succeeds.

One of the main issues of transactional memory is performance. A transaction can read and write arbitrary locations in memory, which leads to a problem because the memory may be altered by concurrent transactions at any time. To ensure a transaction executes in a consistent state, the transactional memory system has to check for modifications between two consecutive memory accesses. This *validation* leads to a significant overhead because the list of memory locations that have to be checked, known as *read set*, usually increases as the transaction progresses.

Several solutions have been proposed to reduce the amount of required validations. In this paper we propose a transactional memory system that runs in different phases, called *Sync-Phase* STM. The time is divided into alternating (short) commit and (longer) computation phases. Transactions are only allowed to commit during a *commit phase*, while active transactions compute only during *computation phases* and stay idle during a commit phase. As a result, transactions only need to validate their state and their read set once after each commit phase.

The *Sync-Phase* technique can be implemented in basically all STM systems that validate a read set to ensure consistency. The basic idea of *Sync-Phase* STM, i.e. executing multiple commits at the same time to make them look like a single commit to an outside observer, can also be helpful beyond validations of read sets, e.g. it might be beneficial to use the technique to reduce the overhead for extending the validity of a transaction for time-based transactional memory. Extending the validity of a read set potentially requires traversing all read objects.

For evaluation we build a STM system similar to DSTM and compared a system with phases, to the "original" version, as well as to a variant that only performs validations if another transaction committed. In other words, the system employs a commit counter (as in TL2). The results on a 16 core machine show that DSTM with synchronization phases keeps up well with the commit counter approach and exceeds its performance when access to the counter becomes frequent, i.e. the scalability issues of having a global commit counter manifest.

The comparison shows that *Sync-Phase* STM performs much better than non-*Sync-Phase* STM for transactions with a read set of notable size. For short transactions, the performance gain is not that large. In artificial cases *Sync-Phase* STM might yield no gain or can be a little bit slower. The results further show that *Sync-Phase* STM is scalable for many threads, which is a big advantage considering future systems will probably use lots of cores.

2 Related Work

The idea of using transactions in programming languages to parallelize software was first introduced by Lomet in 1977 [12]. The paper did not describe a practical implementation though and so it did not arouse interest in the scientific community. The idea lay fallow until 1993 Herlihy and Moss [8] proposed transactional memory as a new multiprocessor architecture that allows lock-free synchronization. Lock-free means that no mutual exclusion is required to operate on a data structure, which avoids common problems of the conventional locking techniques like priority inversion, convoying, and deadlocks. They introduced an implementation based on extensions to any multiprocessor cache-coherence protocol and presented simulation results which show that transactional memory matches or outperforms the best known locking techniques for simple benchmarks. In 2003 Herlihy,

Luchangco, Moir, and Scherer [9] presented almost concurrently as Harris and Fraser [6] the first dynamic STM system (DSTM) where transactions could perform random access on data. Since then many systems have been proposed.

In 2003 Harris and Fraser [6] published almost contemporaneously to Herlihy et al.'s DSTM the first practical STM system integrated into a programming language (Java), called WSTM (word-granularity STM). WSTM detects conflicts at word granularity, avoiding the necessity of changing an objects layout to fit the requirements of the STM system. In 2006 Dice and Shavit [3] combined deferred update with blocking synchronization into an STM system called transactional locking (TL). Later in 2006 Dice, Shalev, and Shavit presented transactional locking II (TL2) [2], an improved version of their first algorithm. It is based on a combination of commit-time locking and a global version-clock based validation technique, which efficiently avoids periods of unsafe execution by guaranteeing that transactions operate only on consistent memory states. This counter needs to be incremented atomically before the commit of a transaction. This requires a locking mechanism, which negatively affects performance and, more important, prevents parallel commits of concurrent transactions. Though there are scalable replacements in principal, they do not come for free, i.e. lower performance if contention on the counter is not that large.

With transactional locking, the validation of the read set can be omitted, except when committing, which significantly increases the performance of the STM. According to Dice, Shalev, and Shavit it outperforms all former STM algorithms and is competitive with the best hand-crafted fine-grained concurrent structures. On the downside it uses locking and a global version-clock which is not scalable to modern multi-core multiprocessors with hundreds of threads, as shown for example by Lev et al. in 2009 [10]. In turn Lev et al. presented SkySTM which was the first STM that supports privatizations and scales better than existing implementations. Before in 2008 Avni and Shavit [1] already presented a decentralized approach called TCL, the first thread-local clock mechanism which was scalable to hundreds of threads and works similar to TL2, but without a global clock.

The idea of *Sync-Phase* STM is to divide the time into different phases: computation phases, where transactions are not allowed to commit, and, alternating, commit phases. This reduces the number of required validations of the read set because each transaction needs to validate only once after each commit phase. On the other hand it does not require a global data structure, like, for example, transactional locking II, so it scales well for hundreds of concurrent threads.

Riegel, Fetzer, and Felber presented another time-based method [13] to increase the performance of transactional memory. They introduced a solution comparable to TL2, but with a real timestamp instead of a version-clock, thereby avoiding the bottleneck of a global counter. Though they also mention the possibility to use a global counter (as in TL2) they prefered using an actual clock for scalability issues. Objects are versioned using time-stampes, i.e. all objects have a validity range for a certain transaction. If a transaction accesses a new object and the intersection of all validity ranges of all previously accessed objects plus the new object becomes empty then a transaction can try to extend the validity range of its accessed objects. This potentially requires inspecting each object. In principle an extension might be necessary for every accessed object. With *Sync-Phase* STM we can ensure that for a certain time span, i.e. the duration of a computation phase, only one extension might be necessary. This holds since no commits take place in the computation phase and thus the validity range of the accessed objects cannot change.

In 2007, Lev, Moir, and Nussbaum [11] introduced a system with a similar name, phased transactional memory (PhTM). It is, however, not related to our *Sync-Phase* STM; instead they propose a technique that allows switching between different phases, each implemented by a different form of transactional memory support, thus using the benefits of each implementation according to the current environment and workload.

Another well investigated topic – often cited as performance critical – is contention management (e.g.[14]). A contention manager decides which transaction should be aborted in case of a conflict. Though choosing the right decision can have an extreme effect in a theoretic worst case analysis [5, 15], experimental evaluation [7] hints that load adapation, i.e. reducing the number of concurrently running transactions in case of frequent conflicts,

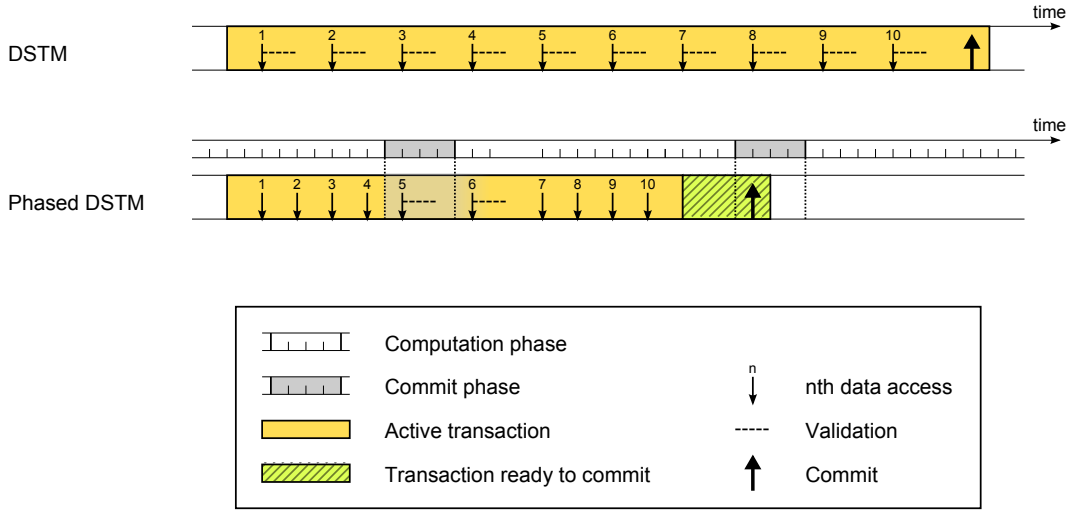


Figure 1: The illustration shows at which occasions a transaction in *Sync-Phase* (D)STM and (D)STM validates its read set. While the transaction validates the read set after each data access in STM, validations are only required during and once after a commit phase in *Sync-Phase* STM. However in (ordinary) STM the transaction can commit immediately, while in *Sync-Phase* STM it has to wait until the next commit phase.

has a bigger influence than the decision which transaction is aborted.

Apart from performance transactional has to deal with many other issues, such as nested transaction and interaction with legacy (non-transactional) code. For a recent overview on transactional memory consult [4].

3 Sync-Phase Technique

The *Sync-Phase* technique alternates *computation phases* and *commit (= synchronization) phases*. During a computation phase, transactions are only allowed to compute and not to commit. Transactions that are finished have to wait for the next commit phase. As soon as a commit phase starts, all transactions that are finished commit their state to the program's memory, while those transactions that are not finished continue execution during the commit phase, but are forced to perform validations. So transactions only need to validate their state once after a computation phase, i.e. directly after the commit phase (on the first access of an yet untouched object) and during the commit phase. Since per computation phase at most one validation is necessary, independent of the number of threads, *Sync-Phase* STM scales well.

The duration of the two phases can be adjusted at will, either based on fixed time intervals or by using an algorithm that sets phase durations at run-time. The behavior of *Sync-Phase* STM in comparison to an STM validating on every read object is illustrated in Figure 1.

3.1 Phase Duration

Intuitively, computation phases should be relatively long compared to commit phases. The reduction of unnecessary validations depends on the relation between the duration of the computation phase and the commit phase. For example, if the commit phase takes 10% of the duration of a whole phase cycle (a phase cycle contains one computation phase plus one commit phase), the validations will also be reduced to roughly 10% of the original amount. In fact, the reduction is somewhat larger, since transactions can execute faster during a computation phase, thus avoiding even more unnecessary validations.

Using only phases of fixed duration is very inflexible. If transactions are short then computations phases should

be short as well, since otherwise transactions must potentially wait a long time before being allowed to commit. To improve the performance of *Sync-Phase* STM we get rid of the explicit use of the phase duration (as a parameter) and define two rules to start and end commit phases. Let N be the total number of active transactions, i.e. non-aborted and running (but potentially temporarily sleeping) transactions, and N_c is defined as the number of transactions that are ready to commit:

1. A commit phase starts if a fraction of all transactions is ready to commit: $N_c/N \geq c_0$ for $c_0 \in [0, 1]$
2. A commit phase ends if a fraction of all transactions has committed: $N_c/N \leq c_1$ for $0 \leq c_1 < c_0$

Rule 1 prevents that time is wasted by waiting for the next commit phase if enough transactions can commit. Ideally, the choice of c_0 is made such that the cost of waiting of the N_c transactions balances the costs of validations of the other $N - N_c$ transactions. We discuss parameters in detail in Section 7.1.3. A commit phase should be short, since transactions must perform validations on every access of an unread object during this phase. In case, only a few transactions want to commit, it might be beneficial to stop commit phases. This is the idea behind the second rule. Generally, it makes sense to let each transaction which is ready to commit, also commit, i.e. stop a commit phase only if no transactions want to commit. In particular, since a commit phase is short. But in case there are a few threads executing many short transactions and many other long running transactions with large read sets, the picture might change. Essentially, these short transactions might cause the sytem to stay in a commit phase permanently as discussed in detail in Section 7.1.3. For illustration of the rules, see Figure 5 in the Appendix.

Unfortunately, Rule 2 might make a lock-free STM system lose its progress property. Though many STM systems do not guarantee lock-freedom (but only obstruction freedom) we still want to briefly address this issue. Assume there are three transactions A, B and C. Say all conflict after having executed for a while, e.g. transactions A and B write to object O_{AB}, A and C write to object O_{AC} and B, C write to object O_{BC}. Now, if we demand that all transactions must be ready to commit to start a commit phase, i.e. $c_0 = 1$, then it might be that there are always two active transactions, but only one is ready to commit. There are several solutions for this problem, e.g. a waiting transaction gains power with time or through aborts of other transactions and is allowed to eventually trigger a commit phase.

Computing the number of active transactions also imposes overhead. However, note that for the evaluation of the rules, looking at a small sample generally provides a sufficient approximation.

4 Implementation

We implemented a transactional memory system which builds upon DSTM with invisible readers. Whenever an object is read for the first time by a transaction, i.e. it is not yet in the read set, a transaction validates its current read set and adds the object to the set. We extended the (core) DSTM system in different ways, i.e. by a commit counter and using synchronization phases, to demonstrate the tradeoffs and effectiveness of our technique. On the one hand one wants to keep computation phases short to minimize the waiting time of a finished but not yet commited transaction, on the other hand one wants to make computation phases long to minimize the number of validations.

We used the timestamp contention manager and a simple form of load adpation[7], i.e. if a transaction got aborted, its running thread sleeps for a millisecond.

4.1 DSTM with Synchronization Phases

Whenever a transaction accesses an object not yet in the read set, it checks if a validation is necessary, i.e. if it is in a commit phase or if it has not performed a validation in the current computation phase. We implemented

Sync-Phases in three variations. The first implementation uses phases of fixed length. It is (strictly) clock driven, which means that the start and end points of each phase are exactly defined by the system time. A transaction has to read the clock and then decide which phase is current. Though for fixed phase duration this boils down to a simple test, both the test and commit must be guaranteed to occur within a commit phase. If the time between the test and the commit could be bounded by a small time interval (i.e. by a fracation of the duration of a commit phase, which is in the microseconds range), this would not be much of an issue, but due to preemption of a transaction delays might be arbitrarily long. Currently, there is no operation that allows something like a compare and swap (CAS) operation based on a condition on the system time. Thus, the system does not ensure any correctness properties. Still, we believe that it is insightful to evaluate the performance to determine whether using actual clocks is efficient on current systems. Hardware support which provides a flag that indicates the current phase and changes atomically in time would help. It would solve correctness issues because such a flag could be included in the compare and set operation. The CAS operation thus not only failed if the transaction was aborted, but also if the commit happened in the wrong phase. More precisely, the flag would be 1 for a commit phase and 0 otherwise. A transaction's state would be 1 if it is active, 2 if committed and 3 if it is aborted. A CAS operation to set the state to committed would only succeed if both the commit flag and the state are 1. Our second implementation uses such a flag implemented in software, i.e. through an atomically modified variable that can be updated by a transaction attempting to commit (or by a dedicated coordinating thread that is only active from time to time and updates the flag if necessary). We followed the first approach, i.e. a transaction that tries to commit, reads the clock, potentially updates the flag, commits and waits until the commit phase is over before altering the flag again. Our third implementation uses two rules described in Section 3.1 to start and end commit phases without the need of a clock at all. To deal with the problem that a commit phase might not start due to conflicting transactions (see Section 3.1), we added the number of aborts of still active transactions to the number of committing transactions to obtain the value for variable N_c in Rule 1. For example, if transaction A wants to commit and transaction B has been aborted three times but has still not committed yet, we get $N_c = 4$. Thus, even if all transactions are active but only one of them is ready to commit, a commit phase will start (eventually).

4.2 DSTM with Commit Counter

We implemented DSTM with a global commit counter, i.e. a global counter that is incremented whenever a transaction commits. A transaction only needs to validate its read set, given that the commit counter changed since the last validation (see Figure 6 in the Appendix).[1]

We expect it to perform better than *Sync-Phase* STM for low numbers of concurrent threads and long transactions, but we also expect a drastic decrease in performance as soon as congestion for the counter increases due to many concurrent commits.

5 Evaluation

Four benchmarks were executed on a system with four Quad-Core Opteron 8350 processors running at a speed of 2 GHz.

*Sorted List:*In the sorted list scenario, each transaction inserts an element into a single linked sorted list. The transactions are distributed (roughly) equally over N threads. This scenario is very well suited for *Sync-Phase* STM because there are many validations.

Global Counter: In the counter scenario, all transactions try to increment a global counter. This leads to conflicts if more than one thread is active because only one transaction may modify the counter at a time. Since this task allows no parallelism we expect it to finish equally fast for any number of threads. However more threads

[1]This behavior is not the same as TL2, since TL2 uses a sophisticated versioning approach, which effectively renders validations unnecessary during execution of a transaction. Only before a commit a validation is required.

may introduce some overhead due to conflicts, thus the required time could slightly increase with the number of threads. This scenario is not suited for *Sync-Phase* STM because there are (almost) no validations.

Independent Counters: Each thread creates only transactions that access only one, i.e. their own, counter. As a result, no conflicts can occur, resulting in optimal parallelizability. This allows transactions to commit fast and in parallel – which is not possible in DSTM with commit counter, since the global counter poses a bottleneck. As a consequence we expect *Sync-Phase* DSTM and non-phased DSTM to perform better than DSTM with commit counter, especially for a high number of threads. This scenario represents an application with many objects; the probability for a conflict is small and most transactions access independent data, thus allowing parallel commits.

Array: In the array benchmark, all transactions access elements in a global array of 100 elements. Each transaction accesses randomly a certain amount of different elements in read mode and in write mode. The number of reads and writes is varied in the test.

The array scenario is not inspired by a specific application, but serves as a prototype for many scenarios. For example, the scenario of the sorted list can be modeled as an array scenario with many reads and 1 write. The counter scenario on the other hand can be modeled as an array scenario with 0 reads and 1 write.

5.1 Setting Phase Durations and Parameters for Rules

There is a tradeoff between the duration of the commit phase and the computation phase and overall system performance. The commit phase should be as short as possible to avoid unnecessary validations during the commit phase. If the computation phase is too short many unnessary validations might be done. If it is too long, e.g. significantly longer than the duration of a transaction, a transaction must wait for a long time. Unfortunately, transactions are generally not of homogenous duration and also have unknown read sets. Some take only little time to finish, while others perform expensive computations. In the sorted list scenario, the duration of a transaction grows as more and more elements are inserted and the list gets longer. We thus have to find phase durations which perform well for different lengths of transactions.

We performed benchmarks for each scenario with varying phase settings. In one benchmark a transaction performed an insert into a sorted list with many read accesses. In the other, it wrote only a counter. Only rule-based STM yields good results for both benchmarks given fixed parameter settings. In particular, we stopped a commit phase, if no transaction wants to commit ($c_1 = 0$) and started a phase if at least one transaction wants to commit, e.g. for our 16 core system we used $c_0 = 1/16$. The parameters are quite robust to changes. We present a detailed analysis in the Appendix Section 7.1.

5.2 Comparison of STM Variants

In the following sections, we compare the performance of non-phased DSTM, *Sync-Phase* DSTM, and DSTM with commit counter. *Sync-Phase* DSTM is tested in three different forms: the (strictly) clock driven implementation, the implementation with a phase flag, and the rule-based implementation (see Section 4.1). The rule-based implementation uses the same set of parameters, e.g. parameters for the rules, across all benchmarks. The other implementations of *Sync-Phase* STM use close to optimal phase settings (as discussed in the section before), if not explicitly stated differently.

For each test, a given number of transactions is executed. The faster this task is finished, the higher the performance of the STM for this task. To analyze the behavior of the different STMs, the number of threads and transactions is varied and the required time to execute the test in each variation is presented in a plot. The STMs are tested in four different scenarios: a sorted list, a shared counter, a random access array, and independent counters. All results are shown in Figures 2 and 3. For each benchmark we once kept the number of threads constant and varied the number of operations, i.e. equivalent to the number of cores being 16^2 and we also varied the

[2]More concurrent transactions only reduce overall system throughput, e.g. see [7] and its related work on the benefit of serializing

number of threads for a fixed number of operations.

5.2.1 Sorted List

Since all elements are inserted into the same list, one would expect increasing time with increasing number of threads because more conflicts may happen. However the insert operation allows some parallelism; for example if one transaction inserts an element at the end of the list and at the same time a transaction inserts an element in the middle of the list, both transactions may commit successfully if the first transaction commits shortly before the second transaction.

Figure 2 shows that rule-based *Sync-Phase* DSTM and DSTM with commit counter perform equally well. The time they need to execute the tests decreases until 5 threads, then increases again slightly. *Sync-Phase* DSTM with a phase flag performs somewhat worse. Clock driven *Sync-Phase* DSTM needs roughly twice as long to insert the elements. Non-phased DSTM finally performs worst by far, especially with a small number of threads. This is due to the huge amount of validations that is performed with non-phased DSTM.

If the number of threads is fixed to 16 and the number of elements that are inserted into the list is varied from 50 to 1000 then as expected all STMs need more time to finish the task if more elements have to be inserted. More interesting however is the rate at which the time increases. For non-phased DSTM, the complexity of inserting N elements into a sorted list is O(N^3). This is because for each element which is inserted, the transaction has to traverse the list to find the correct location. Traversing the list has complexity O(N), however at each iteration the transaction performs a validation. A validation has also complexity O(N) because there are up to N elements in the read set. Consequently, one insertion has complexity O(N^2) and the whole test thus O(N^3).

The clock driven implementation of *Sync-Phase* DSTM is able to reduce the amount of validations, but it cannot reduce the complexity of the operation. The validations are only reduced to a certain percentage of the original amount. As a result, both non-phased DSTM and clock driven *Sync-Phase* DSTM show the same behavior, with clock driven *Sync-Phase* DSTM being a bit faster. Note that the phase settings for clock driven *Sync-Phase* DSTM are optimized for 300 elements, so that the results for 1000 elements could be somewhat better with an adaptive algorithm.

Sync-Phase DSTM with phase flag, as well as optimized *Sync-Phase* DSTM and DSTM with commit counter perform equally well. More interestingly, their execution time does not increase cubically but rather quadratically. Obviously they are all able to reduce the complexity of the operation from O(N^3) to O(N^2). This happens because the number of performed validations is not a constant fraction of the original amount anymore, but depends on the number of concurrent commits. Concurrent commits happen at an approximately constant rate because it depends roughly on the number of threads, which is constant in this test. This is an important result because it shows that *Sync-Phase* DSTM is able to reduce the additional complexity of an operation which was introduced by the STM's overhead.

5.2.2 Counter

The counter is incremented 10000 times with different STMs and the number of threads varying from 1 to 16. As expected, non-phased DSTM and DSTM with commit counter perform best. Rule-based *Sync-Phase* DSTM is slightly slower, but still competitive. The clock driven implementation of *Sync-Phase* DSTM takes roughly twice as long to execute the test.

The implementation of *Sync-Phase* DSTM with a global phase flag finally performs worst, especially for a low number of threads. This happens because the phase settings are optimized for 16 threads. For 16 threads it performs almost equally well like the clock driven implementation. For lower numbers of threads however, it loses time because the committing transaction has to keep the commit phase active until the time is over. Its thread is

conflicting transactions

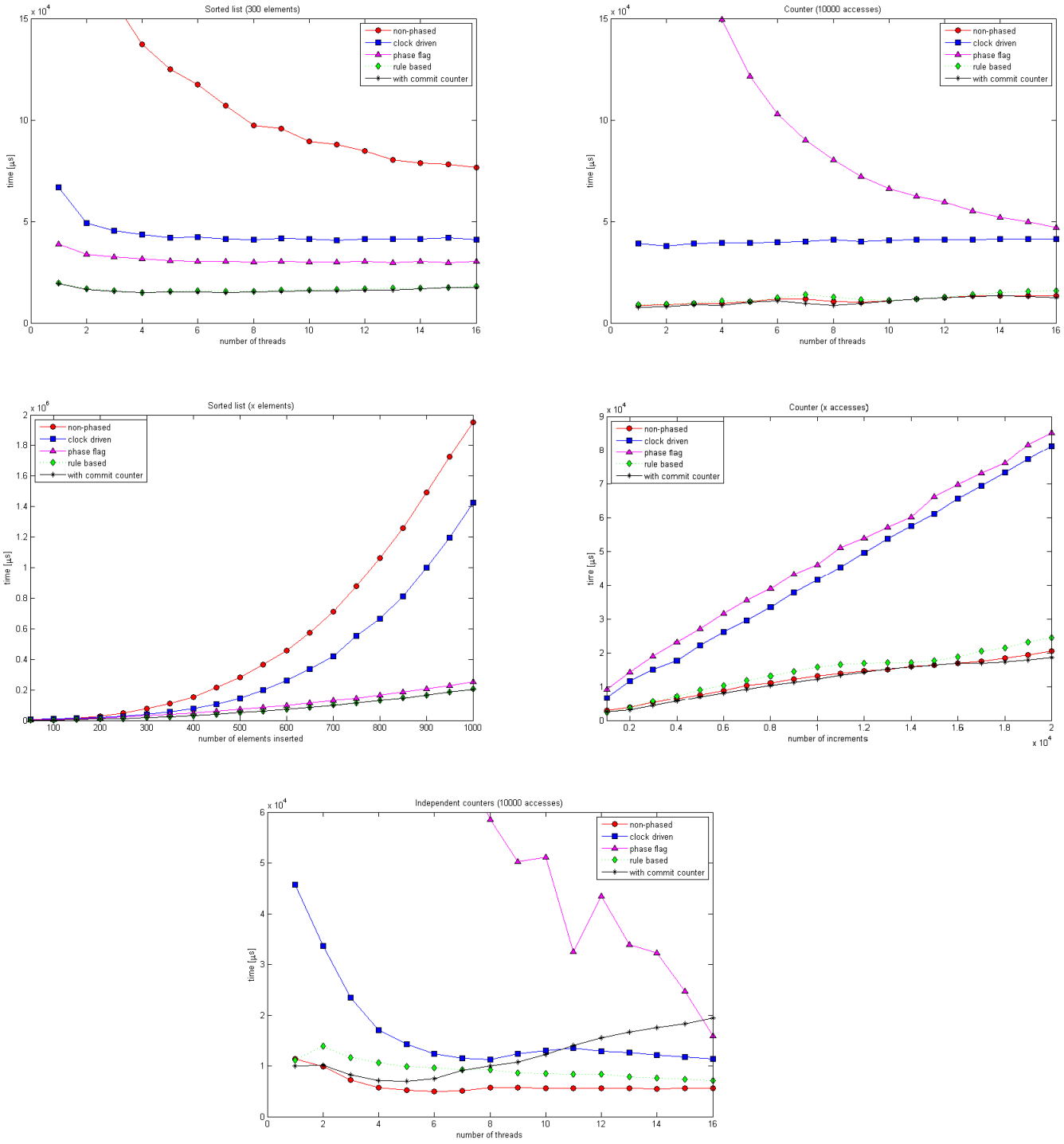


Figure 2: Comparison of performance for three variants of DSTM for the sorted list, the global counter and independent counter benchmarks

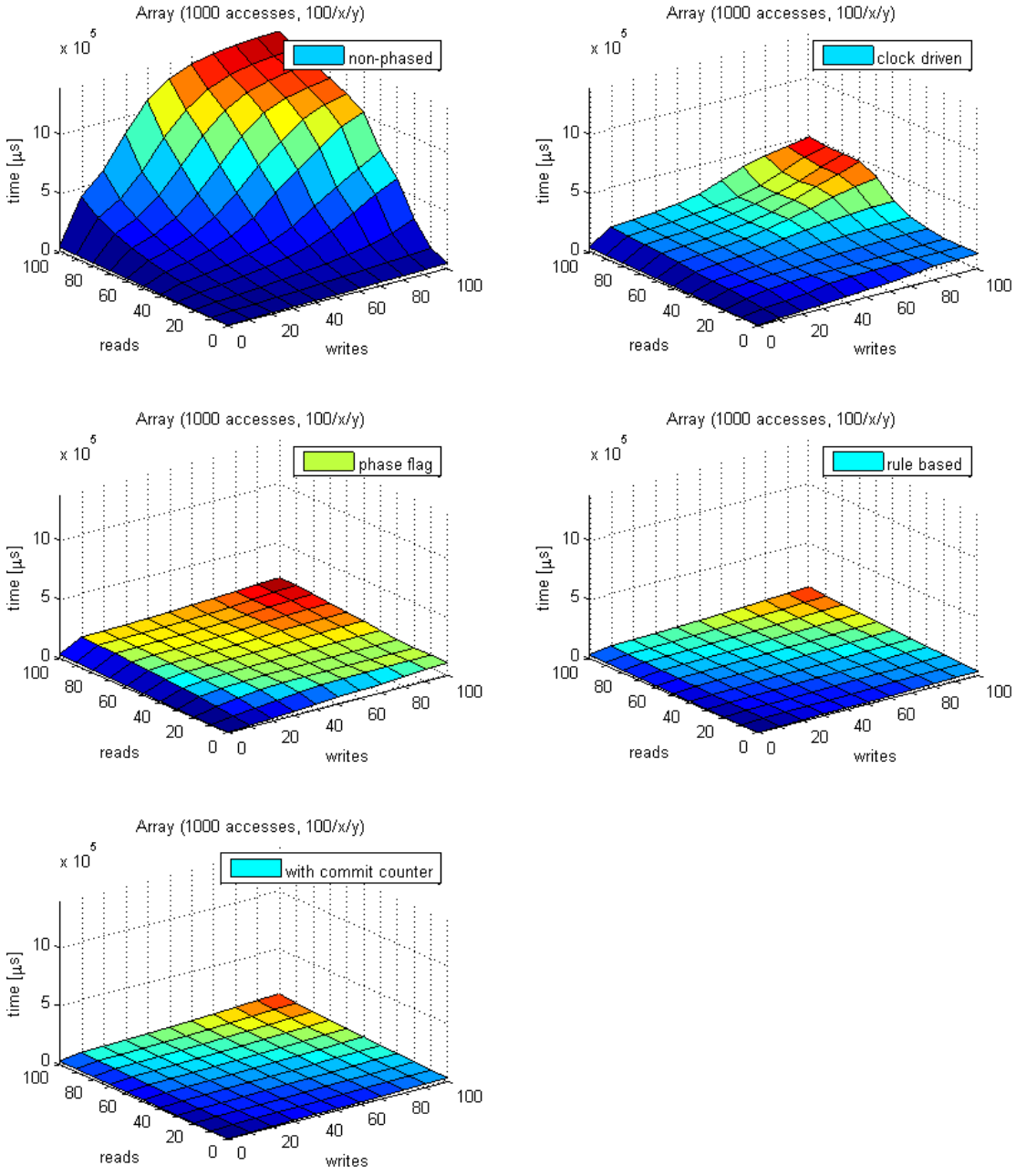


Figure 3: Comparison of different phase settings for three variants of DSTM for the Array benchmark Random elements of an array are accessed in read and in write mode. The array has 100 elements, the numbers of read and written elements are varied from 0 to 100 with 16 threads. All *Sync-Phase* implementations use fixed phase settings of $100\mu s$ phase duration and 10% commit duration.

thus blocked until the next computation phase starts. Consequently, this thread cannot execute a new transaction during the commit phase. This effect has more influence if the test is executed with only a few threads. With more threads, a concurrent thread may step into the breach and start a new transaction.

When the number of threads is constantly 16, but the number of increments is varied from 1000 to 20000, one would expect a linear increase for all STMs. In fact that is almost the case, except for some deviations which are probably due to the internal thread scheduling of Java. The relative performance among all STM implementations remains the same independent of the number of increments.

5.2.3 Array

The plots in Figure 3 are displayed separate for better visibility. As expected, all STM implementations perform best for 0 reads and 0 writes, and worst for 100 reads and 100 writes. Increasing the number of reads or the number of writes has almost the same impact on the time. Non-phased DSTM performs clearly worst. The clock driven implementation of *Sync-Phase* DSTM performs comparably good for up to 40 reads or 40 writes. If both amounts exceed 40, the execution time increases rapidly.

Rule-based *Sync-Phase* DSTM and DSTM with commit counter perform almost equally well. They show roughly a linear increase of time for increasing numbers of reads and writes. *Sync-Phase* DSTM with a phase flag needs roughly twice as long to execute but shows also a pretty flat plot, meaning there is no scenario which is particularly bad. The results of this test show that *Sync-Phase* STM performs well for all combinations of reads and writes and is thus suited for all scenarios. Note that the phase settings are not optimized in this test, hence the results could be even better if the phase settings are chosen adaptively.

5.2.4 Independent Counters

As expected, *Sync-Phase* DSTM and non-phased DSTM perform well (except for *Sync-Phase* DSTM with phase flag which performs badly with only 1 thread; this has the same reason as discussed in the results of the counter scenario). DSTM with commit counter performs good for up to 5 threads, but then its execution time increases roughly linear with increasing number of threads. This happens because transactions cannot commit in parallel, thus introducing congestion and cache misses, which slows down the whole system. This result is important because it shows that *Sync-Phase* STM scales well, in contrast to STM systems that prevent parallel commits like, for example, TL2.

Interestingly the execution time does not decrease notably with increasing number of threads for *Sync-Phase* DSTM, even though the scenario allows full parallelism. One would expect linearly increasing performance, i.e. 16 threads require half the time that is used by 8 threads to execute the test. However the observed effect is much weaker. We are not entirely sure why this happens, but it is obviously not a problem of *Sync-Phase* DSTM because the same happens for non-phased DSTM. It is more likely a limitation of the executing system, for example the memory access or the thread granularity.

6 Conclusions and Future Work

Our benchmark shows that even on a (non-state of the art) 16 core system relying on global counters is not feasible for short transactions. Given the fast growth of the number of cores, we believe that the anarchy of today's transactional memory system must soon give way to a more ordered system. For some software transactional memory systems the *Sync-Phase* technique might be one method to coordinate cores and achieve higher throughput than by unrestricted behaviour of transactions. We are currently working on testing the technique for other STMs.

Acknowledgements
Thanks to Jens Teubner for his helpful comments.

References

[1] H. Avni and N. Shavit. Maintaining consistent transactional states without a global clock. In *SIROCCO*, 2008.

[2] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006.

[3] D. Dice and N. Shavit. What really makes transactions faster? In *Proc. TRANSACT Workshop*, 2006.

[4] H. Grahn. Transactional memory. *J. Parallel Distrib. Comput.*, 70(10), 2010.

[5] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Symp. on Principles of Distributed Computing (PODC)*, 2005.

[6] T. Harris and K. Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2003.

[7] D. Hasenfratz, J. Schneider, and R. Wattenhofer. Transactional memory: How to perform load adaption in a simple and distributed manner. In *HPCS*, 2010.

[8] M. Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Symp. on Computer Architecture*, 1993.

[9] M. Herlihy, V. Luchangco, and M. Moir. Software transactional memory for dynamic-sized data structures. In *Symp. on Principles of Distributed Computing(PODC)*, 2003.

[10] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *Proc. TRANSACT Workshop*, 2009.

[11] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Proc. TRANSACT Workshop*, 2007.

[12] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Conf. on Language Design for Reliable Software*, 1977.

[13] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proc. Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2007.

[14] W. Scherer and M. Scott. Advanced contention management for dynamic software transactional memory. In *Symp. on Principles of Distributed Computing(PODC)*, 2005.

[15] J. Schneider and R. Wattenhofer. Bounds On Contention Management Algorithms. In *ISAAC*, 2009.

7 Appendix

7.1 Evaluation of Phase Durations

The times that were needed to execute the benchmarks for different phase and rule settings are shown in the plots in Figures 4 and 7. The duration of a phase cycle, i.e. the duration of a computation and a commit phase together (in the further text and plots called "phase duration") was varied on a logarithmic scale from 10 to 1000 microseconds; the duration of a commit phase (in the further text and plots called "commit duration") was varied from 10% to 90% of the phase duration. In our first benchmark we inserted 300 elements into a sorted list, i.e. the list grows from 0 to 300 elements. The average size of the read set of each transaction when performing a validation is 38 elements, which makes validations expensive.[3] This test is expected to execute a lot faster with *Sync-Phase* STM because the amount of validations can be reduced a lot. In our second experimental transactions try to increment the same counter, which results in a lot of conflicts and also very short transactions, since only one object is accessed. Consequently, the transactions cannot take advantage of *Sync-Phase* STM since the read set is empty.

[3]The list contains 150 elements in average. Since the list is sorted, each transaction has to traverse the list until it finds the correct location to insert the new element. To find this location, it has to traverse half the list in average, which gives an overall average of 75 elements. When validating, the average size is of the read set is thus 37.5 elements.

7.1.1 Strictly Clock Driven Implementation

The plot in Figure 4 shows a clear optimum for this test: the optimal phase duration is approximately $70\mu s$ (mind the logarithmic scale) and a commit duration of 20% (i.e. $14\mu s$). Shorter phase durations perform worse because commit phases happen too frequent, thus resulting in more validations. Longer phase durations perform worse as well because finished transactions have to wait longer until they can commit. Longer commit durations of course also perform worse because this results in more validations. With good phase settings, *Sync-Phase* DSTM is more than twice as fast as non-phased DSTM.

For the counter benchmark *Sync-Phase* STM performs worse than non-phased DSTM. Due to the short execution time of the transactions however, the waiting time for the next commit phase makes up a big fraction of the overall execution time of the test. In fact the test runs faster if transactions execute during a commit phase because in this case they can commit instantly. As a result, the optimal phase settings for this test is a commit duration of 90%, independent of the phase duration.

If we take a closer look at the plot, we see that for a commit duration of 10%, the execution time increases with growing phase duration, until it reaches a maximum at approximately $100\mu s$. This happens because with growing phase duration the transactions have to wait longer until they can commit. For phase durations greater than $100\mu s$ however the execution time decreases again because the commit duration, which is 10% of the phase duration, increases as well. This allows more and more transactions to execute completely during a commit phase, thus allowing more than one transaction to finish during one phase cycle.

7.1.2 Implementation with Phase Flag

Since commit phases during which no transaction commits are completely ignored,they do not trigger any validations. This leads to different results and different optimal phase settings.

The sorted list scenario now performs best with a very short phase duration of only $10\mu s$ and a commit duration of 10% (i.e. $1\mu s$). This is the case because with a short phase duration, commit phases happen very frequently, which reduces the average waiting time if a transaction is ready to commit. Unneeded commit phases are ignored and therefore no additional validations are performed. Overall, the test performs even better with the second implementation of *Sync-Phase* DSTM, than with the clock driven implementation.

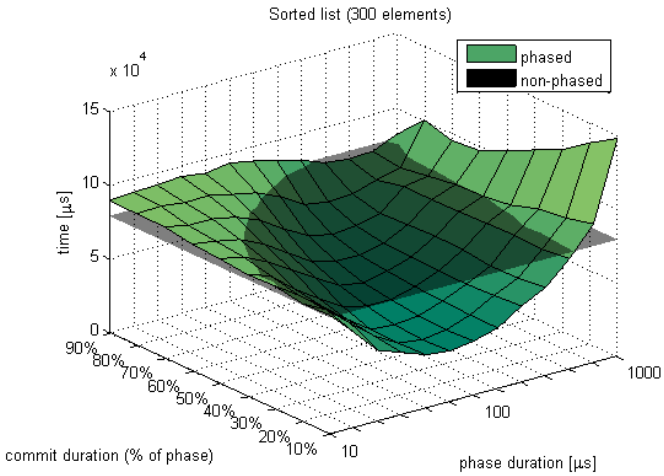
Longer commit phase durations result in even shorter waiting time but on the other hand they also increase the time during which validations have to be performed once a commit phase is started. These effects nearly cancel each other, hence the plot does not indicate many differences for different commit durations. At short phase durations, short commit durations perform slightly better; at long phase durations, short commit durations perform worse.

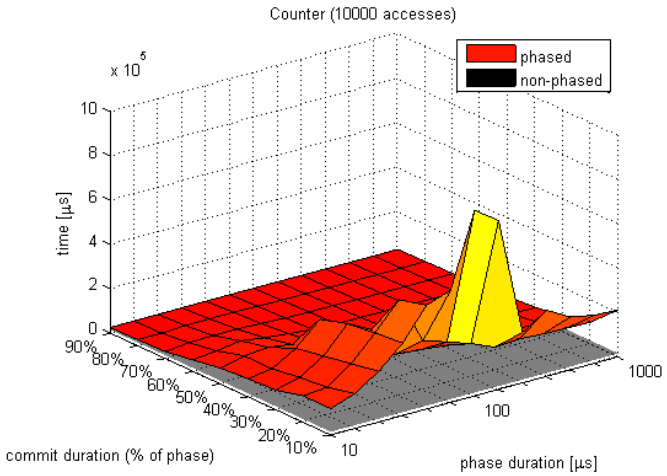
For the counter benchmark the plot looks basically the same like the one for the clock driven implementation. Again we see the worst-case values at approximately $100\mu s$ and 10% commit duration. Overall, the test performs worse than with the first implementation of *Sync-Phase* DSTM. This is the case because the counter test runs faster if transactions execute during commit phases, hence the ignored commit phases actually slow down the system in this scenario instead of speeding it up.

And there is yet another effect which lowers the performance of this test. For a phase duration of $1000\mu s$ and commit durations between 70% and 90% we see a heavy increase of the execution time. This happens because if a transaction aborts, the executing thread waits a short time before restarting it. In this scenario all transactions except one have to abort because they cannot modify the counter in parallel. Consequently the only thread which can execute a new transaction right after a commit is the thread which just committed because it is the only thread that does not have to wait. However in this implementation of *Sync-Phase* DSTM, the transaction which starts the commit phase is also responsible to end it. To do so, it has to wait until the time of the commit phase is over before it can end the commit phase. As a result, this transaction cannot instantly end after it committed; instead it is blocked and has to wait until the commit phase is over. This allows only one transaction to execute per phase cycle in this scenario, which slows down the whole system.

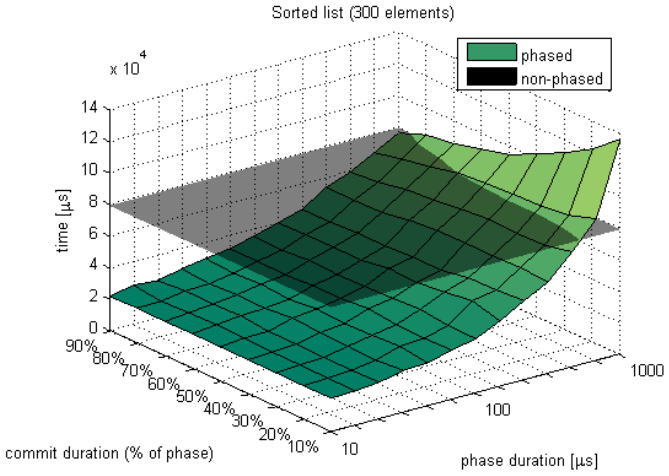
7.1.3 Rule-based Implementation

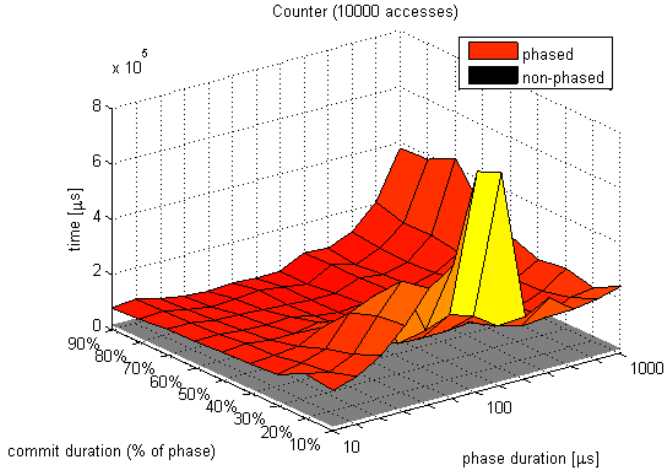
The figure 7 shows that obviously waiting for many transactions to commit (almost) concurrently only makes sense if this is possible at all, i.e. conflicts are rare. Note that the influence of the parameters is mainly small, i.e. it is only larger than 20% for the independent counters benchmark. For the (global) counter benchmark only one transaction can commit though many more can be active. Since we also count active transactiosns that aborted and restarted once as committing, a commit phase will start eventually, but clearly it should start immediately if just one transaction wants to commit. For the sorted list most of the time at most 2 or 3 transactions can commit (almost concurrently), thus the results are similar as for the counter benchmark. For the independent counters benchmark the read set is empty, thus, ideally, a commit phase is started whenever





((a)) Clock driven implementation





((c)) Implementation with a phase flag

Figure 4: Comparison of different phase settings for two variants of DSTM with synchronization phases for two benchmarks using 16 threads: 300 elements inserted into a sorted list, 10000 increments of a counter

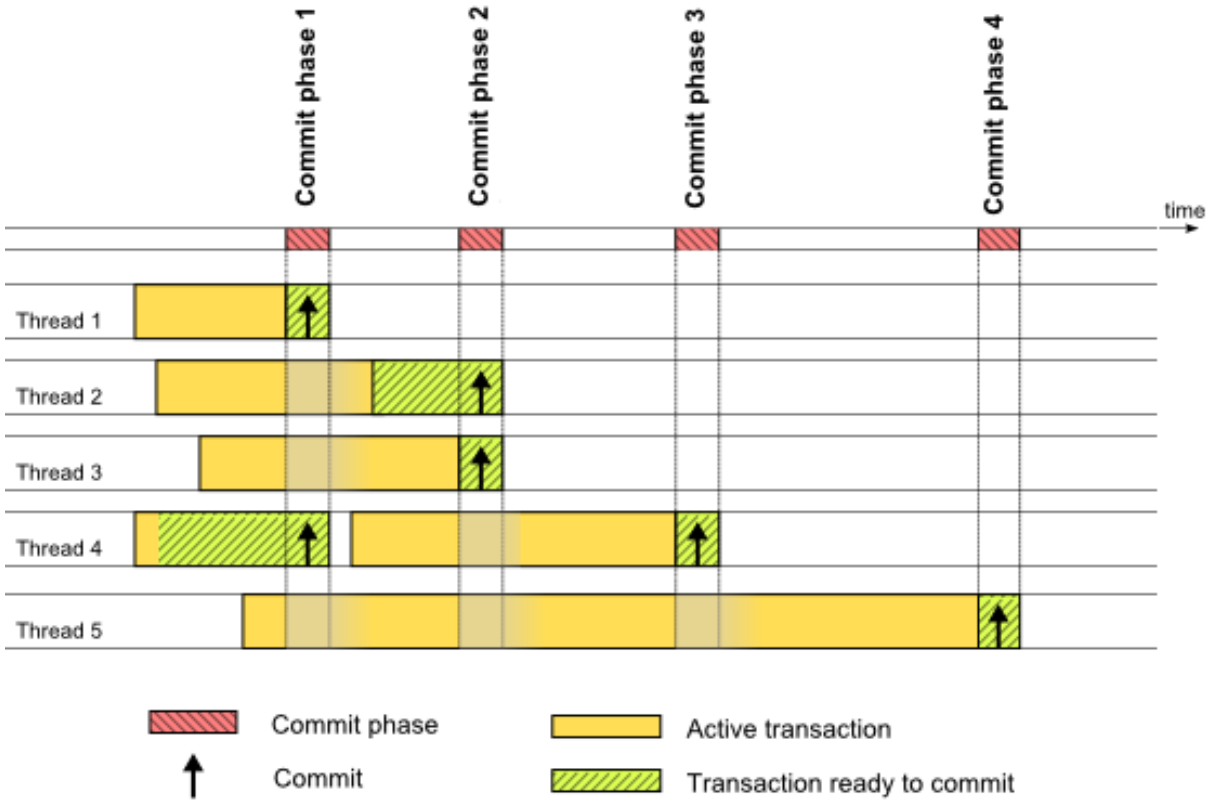


Figure 5: The illustration shows at which occasions commit phases exist in *Sync-Phase* STM with activated optimizations during an execution of transactions in five concurrent threads. Commit phases are started if at least $1/3$ of all active transactions are ready to commit (Rule 1). Commit phases are stopped if no transaction wants to commit (Rule 2).

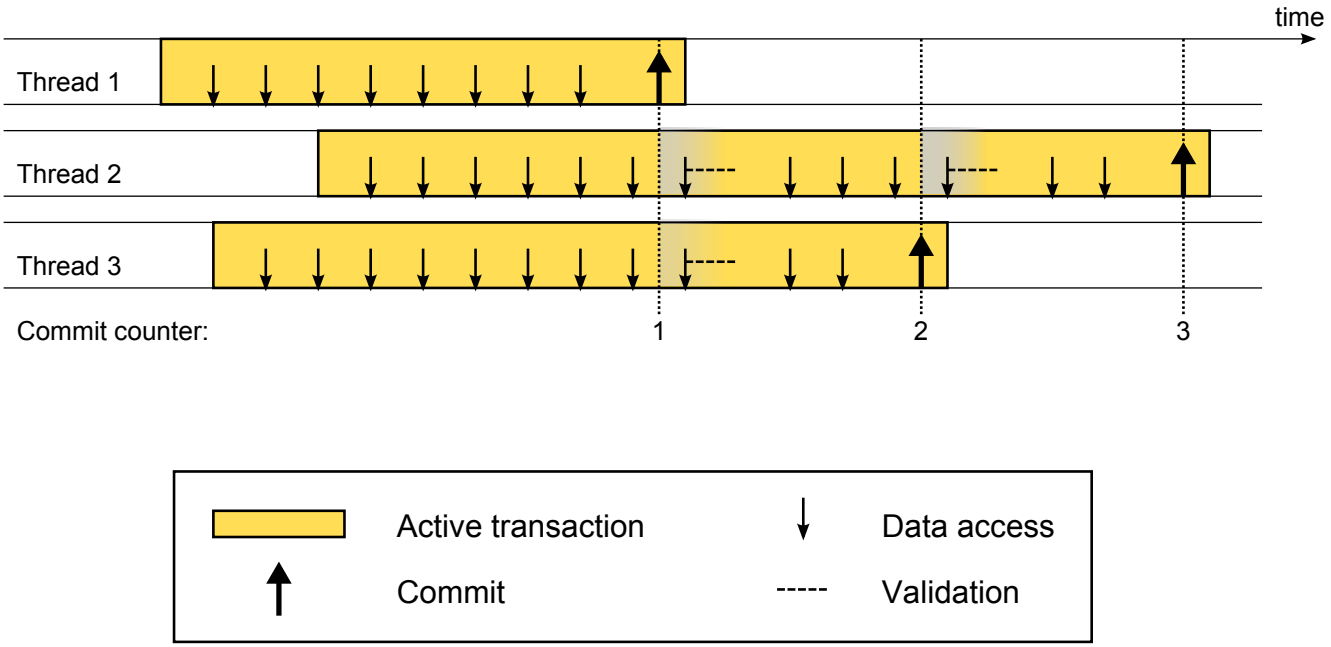


Figure 6: The illustration explains the behavior of DSTM with a global commit counter: whenever a transaction commits, the commit counter is increased which triggers a validation during the next data access in all concurrent transactions.

(at least) one transaction wants to commit. Waiting only slows things down. If we start a commit phase already for few transactions wanting to commit and stop it only if almost all of them have committed, a commit phase might last very long. This is due to the fact that computing the number of transactions ready to commit takes about the same time as a transaction needs to start and become ready to commit. Thus, for a low threshold to stop a commit phase the number of transactions ready to commit might remain above the threshold for quite along time. Increasing the threshold makes commit phases shorter, thus slowing things down. We also performed a read only benchmark, i.e. we accessed 20000 elements randomly out of 100000. For this extreme case, we see clearly that waiting for transactions makes sense, i.e. optimally a commit phase starts if there are at least 5 transactions ready to commit. The parameter to end a commit phase has basically no influence, since all transactions commit once the commit phase is started, i.e., before the system is able to evaluate Rule 2.

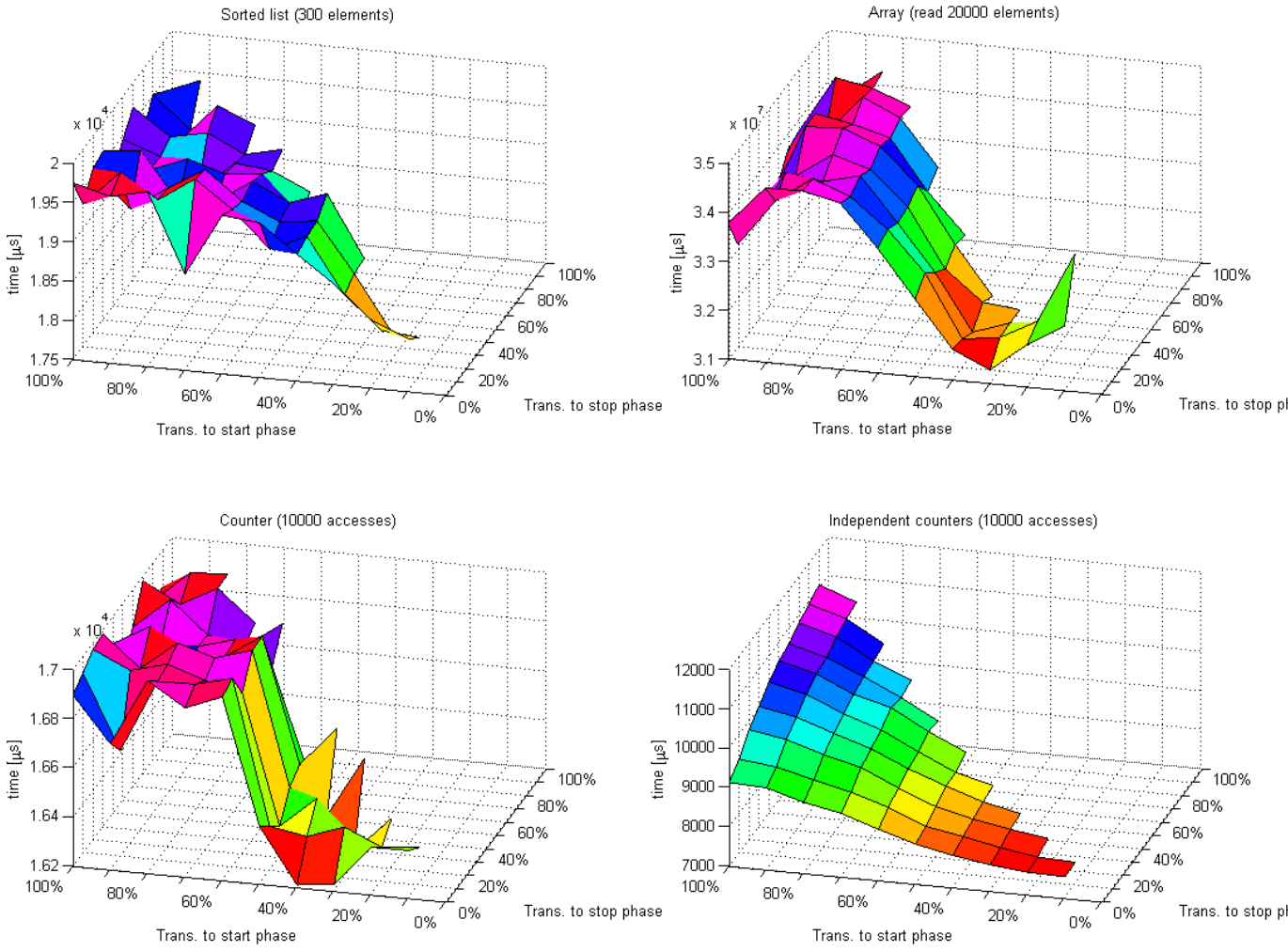


Figure 7: The plot shows the execution time for different parameters for the rule-based execution. The percentage of active transactions needed to start a commit phase respectively to stop a phase are varied.