Serializability of Transactions in Software Transactional Memory

Utku Aydonat Tarek S. Abdelrahman

Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto
{uaydonat,tsa}@eecg.toronto.edu

Abstract

The use of two-phase locking (2PL) to enforce serialization in today's Software Transactional Memory (STM) systems leads to poor performance for programs with long-running transactions and considerable data sharing. We propose the use of *Conflict-Serializability* ($\mathcal{CS}$) instead of 2PL. The use of $\mathcal{CS}$ allows transactions to complete when they would have either stalled or aborted with 2PL, potentially leading to better performance. *Multi-Versioning* further increases concurrency by allowing transactions to read older versions of shared data. We implement an obstruction-free STM system that uses $\mathcal{CS}$ with multi-versioning and evaluate its performance on 32 processors using 3 standard benchmarks. Our evaluation shows that the use of $\mathcal{CS}$ significantly reduces abort rates in programs with high abort rates. This results in up to 2.5 times improvement in throughput, in spite of the overhead of testing for the $\mathcal{CS}$ of transactions at run-time. We also propose an adaptive approach that switches between 2PL and $\mathcal{CS}$ to mitigate the overhead in applications that have low abort rates.

1. Introduction

The Transactional Memory (TM) model has gained considerable popularity in recent years, mainly because of its potential to ease the programming of parallel and multicore systems. TM eliminates the need for user-level locks around accesses to shared data, yet guarantees the correct execution of multi-threaded code. Thus, it makes it easier to express parallel programs and leads to finer-grain concurrency.

There have been several systems implemented to realize the TM model in software. These systems are referred to as *Software Transactional Memory* (STM) systems. In these systems, critical sections in a parallel program are declared as *transactions* and all accesses to shared data are redirected to the STM system. The system keeps track of the shared accesses and controls the progress of transactions to ensure their *linearizability*. That is, the system ensures that each transaction appear to execute instantaneously and after transactions are executed concurrently, the state of the shared data and the outcome of the transactions are the same as if these transactions were executed serially in some order.

Existing STM systems ensure linearizability by emulating a technique known as two phase locking (2PL). In this technique, conflicting accesses to shared data are not allowed from the time of first access to the data by a transaction until the transaction commits. Recent blocking STM implementations [1, 2] implement 2PL explicitly by acquiring a lock for each shared data accessed by a transaction and holding this lock until the transaction commits. Similarly, obstruction-free STMs (e.g. RSTM [3] and DSTM [4]) implement 2PL by allowing only one of the conflicting transactions to commit while aborting the others. One exception is SI-STM [5, 6] which uses snapshot isolation to relax linearizability, but requires the programmer to carefully analyze data accesses to ensure correctness. Otherwise, 2PL is used to ensure linearizability.

The use of 2PL is suitable for applications with short-running transactions and a small degree of data sharing. However, 2PL limits its concurrency in applications with long-running transactions and significant data sharing [7]. In these applications, 2PL leads to excessive stalls in blocking systems or high abort rates in obstruction-free ones. Such applications pose a challenge for today's STM systems and require the use of manual techniques such as early release [4, 8] and open nesting [9, 10]. Regrettably, these manual techniques require considerable programming effort to obtain performance and to ensure correctness.

In this paper, we argue that 2PL can be an overly conservative approach for guaranteeing correctness in STM systems. We propose the use of a model more relax than 2PL and linearizability, which is based on *conflict-serializability* ($\mathcal{CS}$) [11]. This relaxed approach allows transactions to complete even when conflicting accesses to shared data are present and it does not impose some of the limitations imposed by linearizability. Hence, the use of $\mathcal{CS}$ allows transactions to complete when they would have stalled or aborted with the use of 2PL, which can lead to more concurrency and to better performance. We also argue that the use of *Multi-Versioning* [11] can improve upon $\mathcal{CS}$ by allowing transactions to access older versions of shared data. This avoids having to reject operations that arrive late to be serializable.

We implement an obstruction-free STM system that we use as a vehicle to experimentally evaluate the use of $\mathcal{CS}$ and multi-versioning in STM systems and its impact on performance. Our evaluation on 32 processors using three benchmarks shows that the use of $\mathcal{CS}$ does indeed result in significantly lower abort rates compared to 2PL. This leads to better performance for applications that have long-running transactions and suffer from high abort rates. For example, the use of $\mathcal{CS}$ with multi-versioning improves the throughput of transactions in a linked list benchmark by a factor of 2.5 on 24 processors compared to 2PL on the same number of processors. In fact, the performance of this linked list application exceeds that of coarse-grain locking without the use of early-release or open nesting.

The reduction in the abort rates is achieved at the expense of increased overhead caused by the run-time testing for $\mathcal{CS}$. This overhead slows down applications that already have low abort rates with 2PL. Therefore, we introduce an adaptive approach that uses 2PL when the abort rate is low and switches to using $\mathcal{CS}$ and multi-versioning when the rate is high. This adaptive technique results in performance that matches the best that can be achieved by using either 2PL or $\mathcal{CS}$, making the adaptive approach superior. We also compare the performance of our adaptive system to that of a blocking STM system (TL2) and to an obstruction-free one (RSTM) and show that the throughput of transactions on our system exceeds that on both.

The remainder of this paper is organized as follows. Section 2 gives background material on STM systems. Section 3 defines $\mathcal{CS}$ and its role in STM systems. Section 4 describes how $\mathcal{CS}$ is tested for and used in our prototype system. Section 5 describes multi-versioning and how it is implemented in our prototype system. Sec-

tion 6 describes our adaptive approach that switches between 2PL and $\mathcal{CS}$. Section 7 presents our experimental evaluation and compares the performance of our prototype to that of other systems. Section 8 presents related work. Finally, Section 9 gives a conclusion.

2. Background

A transaction is defined as a finite sequence of instructions that are used to access and modify shared data, and that satisfy the linearizability property [12]. Transactional memory provides the illusion that critical sections, i.e. transactions, execute atomically in isolation. Software Transactional Memory (STM) systems provide this guarantee in software by implementing a layer between the shared-data and concurrent threads. Shavit et al. [13] were the first to propose an STM system. This first implementation was followed by systems such as DSTM [4], WSTM [14], FSTM [15], RSTM [3], McRT-STM [1] and TL2 [2]. These systems differ in their approach for the implementation of STM functionality. The remainder of this section describes the space of implementations based on five aspects: liveness, write-target, granularity, acquiring for writes and visibility of reads.

Liveness: obstruction-free vs. blocking. In obstruction-free STM systems [3, 4], a transaction runs without acquiring locks, optimistically assuming no other transaction operates concurrently on the same data. If this is still true at the end of its execution, the transaction commits its modifications to shared memory. If not, the transaction aborts, its modifications are rolled back and it is re-executed. In other and more recent, blocking STM systems [1, 2], shared-data is associated with locks, transactions automatically acquire these locks at the time of access to protect shared-data, and ensure that deadlocks do not occur. *In this paper, we test the use of $\mathcal{CS}$ in an obstruction-free STM implementation*.

Write Target: write-buffering vs. undo-logging. With write-buffering, the transaction buffers the modifications in a temporary storage and applies them to the memory only when the transaction commits. In contrast, with undo-logging, a transaction directly updates the memory and keeps a log to rollback the memory state if the transaction aborts. *In this paper, we focus on an STM implementation that uses write-buffering*.

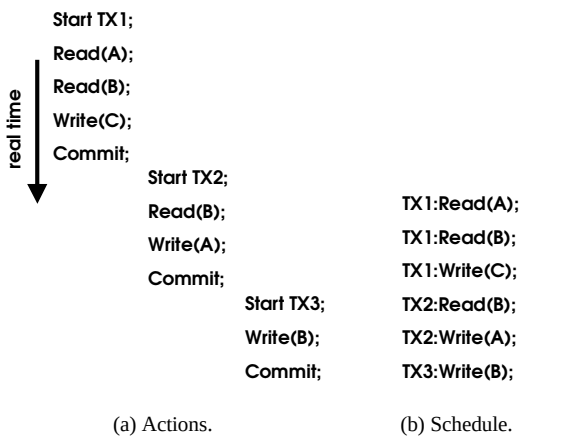
Granularity: word-level vs. object-level. With object level conflict detection, an STM system guarantees that two transactions do not access the same object in a conflicting way. With word level conflict detection, on the other hand, an STM system gives the same guarantee for two transactions accessing the same memory location. *In this paper, we focus on object-level conflict-detection*.

Acquiring for Writes: eager acquire vs. lazy acquire. In eager acquire, a transaction gets the ownership of the object during execution just before writing to the object, announcing to the other transactions that if they try access the object a conflict will arise. On the other hand, in lazy acquire, transactions write to concurrent objects independently, without claiming any kind of ownership. Potential conflicts are detected at commit time when transactions acquire the ownerships of objects until they are updated. *The STM implementation described in this paper uses lazy acquire*.

Visibility of Reads: invisible readers vs. visible readers. With visible readers, transactions are aware of any object read by other transactions. Conflicts are immediately detected if a transaction tries to write to an object that was already read by another object. In contrast, with invisible readers, objects are read transparently to other transactions. The current version of the object is stored in a list and this version should match the version of the object in memory at commit time. *In this paper, we focus on visible-readers*.

3. Conflict-Serializability and STM Systems

In this section, we describe the concept of *conflict-serializability* ($\mathcal{CS}$), which originates from the database domain [11]. We formally define $\mathcal{CS}$ and describe how it guarantees serializability in concurrent executions. We then discuss how existing STM systems use a more strict approach to serialization than $\mathcal{CS}$, called *Two-Phase Locking* (2PL). The use of 2PL leads to unnecessary stalls in blocking STM systems or unnecessary aborts in obstruction-free ones.



(a) Actions. (b) Schedule.

Figure 1. Serial transactions.

We propose that the use of $\mathcal{CS}$ can increase concurrency and result in better performance.

3.1 Conflict-Serializability $\mathcal{CS}$

Transactions are represented by the set of actions they perform in time. The actions are: read, write, abort, and commit. Read actions represent when objects are read. Write actions represent when objects are written. However, since writes are buffered, the effects of these write actions are not visible to other transactions until the time of commit, when shared memory is updated. Therefore, it is possible to view all write actions of a transaction to happen at the end of the transaction, even though write actions occur any where in the transaction. Further, write actions represent when an object is committed while commit operations represent when a transaction commit completes.

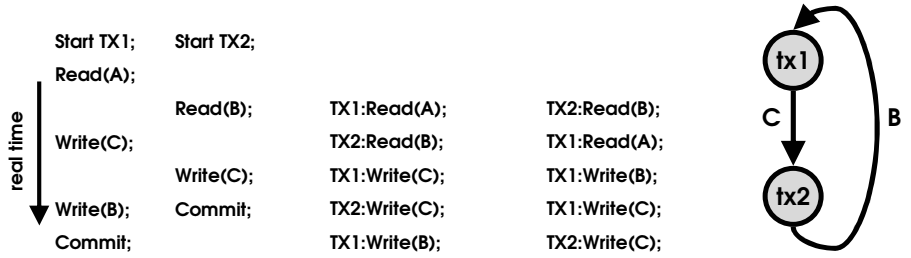
The *schedule* of a set of transactions is a list the actions performed by the transactions in the order they happen in time. Figure 1(a) shows three transactions, TX1, TX2 and TX3. Transaction TX1 reads objects A and B and writes to object C; TX2 reads B and writes to A; TX3 writes to object B. The schedule for these transactions is shown in Figure 1(b).

A *Serial Schedule* is a schedule in which the actions of one transaction do not interleave with the actions of another in time. For example, the schedule in Figure 1(b) is serial because all the actions of TX1 are before the actions of TX2 and all the actions of TX2 are before the actions of TX3. In this case, we can think that each of TX1, TX2 and TX3 executed atomically in this order. In contrast, the schedules in Figure 2(b) and Figure 2(c) are not serial because the actions of the transactions interleave in time. Serial schedules are always correct concurrent executions.

Linearizability guarantees that the state of the concurrent system after the execution of transactions is equivalent to a state had the transactions been executed one after the other in some serial order. However, this serial order should match the actual execution order of transactions at run-time. That is, if a transaction `TX1` completes before another transaction `TX2` starts executing, the final state of the execution should be equivalent to a serial execution of transactions in which TX1 comes before TX2.

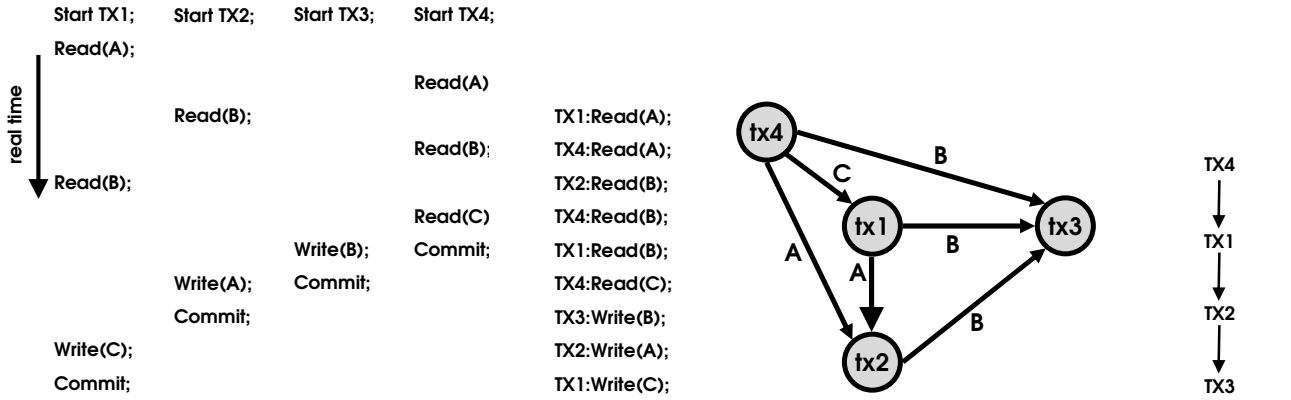
Serializability provides the same guarantees as linearizability, however, it does not impose any restrictions based the execution order of transactions. That is, the final state of the execution can be equivalent to any serial execution of transactions, no matter which transaction completes earlier. Serializability guarantees the correctness of concurrent transactions [11]. If a schedule is not serializable, an erroneous outcome of execution may result.

A *conflict* is said to exist between two or more actions if (1) the actions belong to different transactions, (2) the actions access the same object, (3) at least one of the actions is a write. For instance, in Figure 2(a), the pair of actions TX1:Write(B) and TX2:Read(B) are conflicting, and so are TX1:Write(C) and TX2:Write(C). Two



(a) Actions. (b) Schedule 1. (c) Schedule 2. (d) Ser. graph.

Figure 2. Conflict equivalent schedules.



(a) Actions. (b) Schedule. (c) Serializability graph. (d) Serialization.

Figure 3. Conflict-serializability of transactions.

transactions are said to be *conflicting* if they contain actions that conflict with one another.

Two schedules are said to be *conflict-equivalent* if (1) they contain exactly the same actions and (2) the order of the conflicting actions is the same. The two schedules in Figure 2(b) and Figure 2(c) contain the same set of actions. They are conflict-equivalent because in both schedules TX2 reads B before TX1 writes to it and TX1 writes to C before TX2 does so.

A schedule is *conflict-serializable* if it is conflict-equivalent to one or more serial schedules. The conflict-serializability ($\mathcal{CS}$) of a schedule can be tested using the *serializability graph*. In this graph, each node represents a transaction and each edge represents a conflicting action between two transactions. The edges are directed and are incident from the transaction that performs its action earlier in time. A schedule is conflict-serializable if its serializability graph is acyclic [11].

Consider the schedules in Figure 2(b) and Figure 2(c). Their serializability graph is shown in Figure 2(d)[1]. This graph has a cycle and is thus not conflict-serializable. This is because the conflicting accesses to object B require that, in any conflict-equivalent serial schedule, TX2 executes before TX1. However, the conflicting accesses to object C require that TX1 executes before TX2. This makes it impossible to obtain a serial schedule. In contrast, consider the example transactions in Figure 3(a) and their schedule in Figure 3(b). Transactions TX2 and TX3 are conflicting because of their accesses to object B. Similarly, TX4 is conflicting with all the other transactions because of accesses to objects A, B and C. The serializability graph for this schedule is shown in Figure 3(c). This serializability graph in acyclic and thus the schedule is conflict-serializable. Indeed, the transactions can be executed serially in the order shown in Figure 3(d).

3.2 Serializability in Transactional Memory

Today's STM systems do not allow conflicting accesses to an object from the first time the object is accessed until commit time[2]. Recent blocking STM implementations guarantee this explicitly by acquiring locks for each object accessed by a transaction and by holding these locks until the transaction commits. Similarly, obstruction-free systems allow only one of the conflicting transactions to commit and abort all others, irrespective of the write policy used. More specifically, systems that employ eager-acquire writes do not allow an object to be accessed in write mode by other transactions until the transaction that owns it commits. Similarly, systems that use lazy-acquire write policy allow write-write sharing, but they abort conflicting transactions at commit-time. STM systems with visible readers guarantee that if an object is accessed in read-only mode no other transaction can access it in write mode. Similarly, systems with invisible readers apply validation to detect conflicts and abort transactions.

This policy of handling conflicting accesses employed by today's STM systems effectively emulates *two-phase locking (2PL)* in which a transaction must request an exclusive lock before reading or writing an object and releases this lock only when it commits [11]. However, 2PL has been shown to be a *strict* model of

[1] These schedules have the same serializability graph because they are conflict-equivalent.

[2] This is also true for hardware TM systems, but we focus in this paper on software systems.

serializability for database transactions [16]. That is, for a fixed number of actions performed by a fixed number of transactions, the set of serializable schedules is a superset of the set of schedules that obey 2PL. Therefore, STM systems guarantee serializability by using 2PL, but indeed 2PL is not necessary for serializability.

The strictness of 2PL is illustrated using the example shown in Figure 3(a). Due to the conflicts, only one of the four transactions will commit when 2PL is used. The other transactions will either abort or stall. However, the actions in the figure are indeed serializable. This can be easily seen in the serializability graph shown in Figure 3(c). The schedule of is conflict-equivalent to the serial schedule shown in Figure 3(d), and is thus conflict serializable. All four transactions can commit safely without stalls or aborts.

The use of 2PL in STM systems results in lack of concurrency in blocking systems and in unnecessary aborts in obstruction-free ones. This is because some schedules that do not obey 2PL may very well be serializable and thus produce correct outcome. For applications that have abundant parallelism and are without excessive data sharing, the use of 2PL is not likely to be a problem. However, for applications that involve excessive data sharing among transactions, 2PL is likely to limit concurrency and degrade performance. It is not surprising that these types of applications do not perform well on today's STMs and do require the use of techniques such as early release to reduce the number of aborts and obtain good performance.

Therefore, we propose the use of $\mathcal{CS}$ instead of 2PL to determine if transactions should progress or stall/abort in STM systems. We expect that the use of $\mathcal{CS}$ will be beneficial to performance, particularly in the case of applications that have a large degree of data sharing making them difficult to execute efficiently on today's STM systems. In the following section we describe how $\mathcal{CS}$ can be realized within an obstruction-free STM.

4. Implementation

We implemented an obstruction-free STM, which we refer to as the *Toronto Software Transactional Memory (TSTM)*. The system is intended as a vehicle for exploring the benefits of using $\mathcal{CS}$ to enforce serialization. Thus, we elect to adapt a programming model that is similar to that of comparable systems such as DSTM [4] and RSTM [3], to detect conflicts at the object level, and to use write-buffering. Further, TSTM does not support nesting of transactions, and therefore each thread executes a single transaction at a time. These decisions allowed us to focus on the implementation and evaluation of the use of $\mathcal{CS}$. This section describes, the programming model of TSTM, how $\mathcal{CS}$ is tested for in the system and discusses some performance considerations.

4.1 Programming Model

A transaction is started by using the BEGIN_TRANSACTION keyword, which returns an identifier for the current transaction. This transactional identifier is used in transactional operations. The END_TRANSACTION keyword completes a transaction either by committing or aborting it. Objects shared among transactions are wrapped with special transactional wrappers. To access a shared object, the transactional wrapper must be "opened". Four transactional "open" operations can be performed. They are:

- *Open For Read*. This operation is used to access a shared object in read-only mode.
- *Open For Write*. This operation is used to access the shared object in write-only mode. That is, the data members of the shared object can not be read before being written to first.
- *Open For Read-Write*. This operation is used to access the shared object in both read and write mode. That is, the data members of the shared object can be read and can also be modified.
- *Open For Delete*. This operation is used to tell to the transactional memory system that, if the transaction commits successfully, the wrapper and the shared object contained in it can be destroyed.

When a transactional operation is called, the transactional memory system: (1) makes sure that the shared object can be accessed in the indicated mode, (2) records the access, and (3) returns the shared object, or in the case of a write operation, a copy of the shared object.

4.2 Testing for $\mathcal{CS}$

A naive approach to test for $\mathcal{CS}$ is to incrementally build the serializability graph during the execution of transactions and to check after each action that the graph remains acyclic. This approach is likely to incur significant overheads due to the cost of the required graph traversals. Instead, we use a more efficient approach that attempts to incrementally construct a conflict-equivalent serial schedule based on the actions performed by the transactions. If such a schedule can be constructed then the transactions are conflict-serializable. This approach extends the timestamp intervals approach for testing serializability that is employed in real-time databases [17] (see the related work section).

The conflict-equivalent serial schedule is constructed by determining the *serializability order number (SON)* for each transaction. The SON is an integer that indicates the relative order of a transaction among all transactions in the conflict-equivalent serial schedule that is being constructed. During execution, the SONs of the transactions are determined based on the relative order of their conflicting actions. That is, the transaction that performs its access first will have a smaller SON because, in any conflict-equivalent schedule, the relative order of the conflicting actions must be the same, as described in Section 3. If a unique SON can be determined for each transaction a conflict-equivalent serial schedule exists and it can be assumed that the transactions execute in the same order as their SONs. Further, in this serial schedule, it can be assumed that all read/write/commit actions made by a transaction atomically happen according to their SONs.

The SONs of transactions are determined using the following two basic rules:

1. If a transaction TX1 accesses (reads or writes) an object that has already been committed by another transaction TX2, then TX1's SON must be be higher than that of TX2, i.e. SON(TX1) $>$ SON(TX2). This is because the access of TX1 comes later than the access of TX2.
2. If a transaction TX1 reads an object that is later committed by another transaction TX2, then TX1's SON must be lower than that of that of TX2, i.e. SON(TX1) $<$ SON(TX2). This is because TX1's read action happens earlier than the commit action of TX2.

The first rule imposes a lower bound on the SON of a transaction. Similarly, the second rule imposes an upper bound on the SON of a transaction. Thus, we can associate with each transaction a pair of integer values that reflect the lower and upper bounds on its SON. During execution, the lower bound of a transaction is initialized to 0, and the first rule is repeatedly used to increase this lower bound. The upper bound is initialized to ∞ and the second rule is used to lower its value. If at any moment during execution, the lower bound on the SON for a transaction becomes higher than its upper bound, this transaction cannot be placed in a conflict-equivalent serial schedule. In this case, the transaction aborts.

If a transaction performs all its accesses without aborting, it starts its commit phase during which it examines its SON range and determines a specific SON. If the upper bound of the range is not ∞, then it reflects the SON of some conflicting transaction. Therefore, the SON of the transaction is selected as the upper bound minus one. If the upper bound is ∞, then the SON of the transaction is set to the lower bound plus n, where n is the number of threads[3]. In both cases, the choice of the SON reflects the need to assign a sufficiently high SON to each transaction so that it is possible to assign unique smaller SON values to the remaining $n-1$ active transactions.

Figure 4 uses an example to illustrate the above method for testing the $\mathcal{CS}$ of transactions. Transactional operations are shown

[3] In TSTM, each thread executes a single transaction.

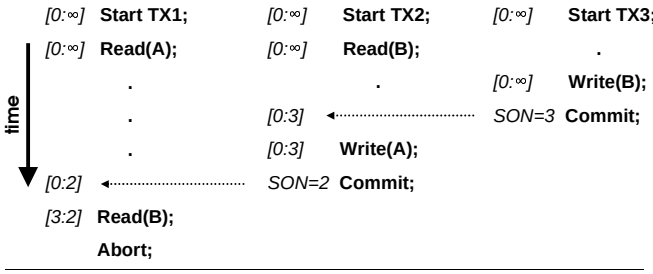


Figure 4. An example schedule for $\mathcal{CS}$.

in bold, whereas SON ranges and values are shown in italics to the left. SON ranges are shown as [L:U], where L is the lower bound and U is the upper bound. There are three transactions that all start at the same time with initial SON ranges of $[0 : \infty]$. It is assumed that the shared objects, A and B were not accessed by any transaction before the three transactions started. Hence, until TX3 commits, the transactional operations TX1:Read(A), TX2:Read(B) and TX3:Write(B) do not affect SON ranges. When TX3 attempts to commit, its range is still $[0 : \infty]$ and, hence, it is assigned a SON value of 3, which is its lower bound plus 3, the number of threads. Further, when TX3 commits, the upper bound of the SON of all other active transactions that access object B must be updated according to the second rule. Thus, the upper bound of the SON of TX2 is updated to 3. When TX2 commits, its SON has a lower bound of 0 and an upper bound of 3. Thus, it is assigned a SON value of 2, which is its upper bound minus 1. Next, TX2 finds that TX1 has already read the object A and is still active. Thus, TX1's SON upper bound is updated to 2. The next action is TX1:Read(B). In accordance with the first rule, this action causes TX1 to update its SON lower bound to 3, which is the SON of TX3 that committed object B. Consequently, TX1's SON range becomes [3:2], which is invalid because its SON upper bound is smaller than its SON lower bound. Thus, TX1 aborts without completing the read action.

Therefore, in the execution of the 3 transactions in Figure 4, TX2 and TX3 successfully committed. On the other hand, TX1's accesses were not conflict-serializable with the remaining operations and thus, TX1 is aborted. That is, the actions of TX1 generated a cycle in the serializability graph.

In order to prove that the described technique produces serializable schedules, we need to show that a cycle in the serializability graph of a schedule causes invalid SON ranges. First, let's assume two transactions TX1 and TX2 and an edge TX1 $\rightarrow$ TX2 between them in the serializability graph of the schedule. This will result in TX1's SON value to be lower than TX2's SON value, i.e. SON(TX1) $< SON$(TX2). This is forced by the first and second rules of the technique described in the previous section. More specifically, the first rule is used if this edge is caused by a write-write or a write-read conflict, and the second rule is used if the edge is caused by a read-write conflict. Now, let's assume a cycle in the serializability graph which consists of transactions TX1, TX2, TX3, ..., TXN and TX1. Consequently, we have SON(TX1) $< SON$(TX2), SON(TX3) $< ... < SON$(TXN) $< SON$(TX1). This gives us SON(TX1) $< SON$(TX1) which is a contradiction and will result in one of the transactions in the cycle to have an invalid SON range. Thus, the technique will always produce serializable schedules.

However, the technique is not exact. That is, due to the assignment of timestamps, it may cause transactions to abort even without a cycle in the serializability graph. In order to elaborate on this, let's assume an ordering of transactions TXO1$\rightarrow$TXO2$\rightarrow$...$\rightarrow$TXON caused by conflicting accesses to an object O. TSTM will assign to these transactions SON values that increase monotonically starting from 0. Now, let's assume another ordering of distinct transactions TXP1$\rightarrow$TXP2$\rightarrow$...$\rightarrow$TXPZ caused by conflicting accesses to another object P. Consequently, the SON values of these transactions will also monotonically increase starting from 0, and hence, they will receive similar SON values. Even though the transactions accessing object O and the transactions accessing the object P do not conflict over these two objects, the assignment of SON values imposes an ordering among these transactions, i.e. SON(TX01)=SON(TXP1) $<$ SON(TX02)=SON(TXP2). Thus, the assignment of SON values may impose unnecessary constraints. These constraints may restrict the set of equivalent serial schedules, causing unnecessary aborts. Consequently, we can say that CS as implemented by TSTM is more relaxed than linearizability, but it is more strict than CS.

4.3 Transactional Operations

In this section we describe the steps that must be taken for each transactional operation in order to implement the SON-based method used to test for $\mathcal{CS}$ described above.

- *Open For Read*. In this operation, the transaction: (1) updates its lower bound based on the SON of the last transaction that committed the object, (2) verifies its SON range and aborts if the range is not valid, (3) records the read, and (4) gets the shared object stored in the wrapper.
- *Open For Write*. This operation performs 3 main steps: (1) it makes a copy of the shared object stored in the wrapper, (2) it records that this particular object is opened for write, and (3) it returns the copy. No serializability checks are made at this time; they are made at commit time.
- *Open For Read-Write*. This operation is the combination of "open for read" and "open for write" operations performed in that order.
- *Open For Delete*. This operation is similar to "open for write". However, in addition, the object is also inserted to an *object deletion list* which will be cleaned when all the transactions active when the object is opened for delete are completed.
- *Committing*. A transaction commits in five steps:
  1. The objects written by the transaction are acquired in accordance with the lazy-acquire technique.
  2. The transaction updates its SON lower bound based on the SONs of transactions that last committed the objects it opened in write mode. If the SON range is valid, the transaction prepares for committing, otherwise it aborts. Note that while a transaction T is committing, other transactions may try to change the SON upper bound of T. This creates a race condition, which is handled conservatively by aborting transactions when they try to update the SON upper bound of a committing transaction.
  3. The transaction determines its SON within its SON range, as described earlier.
  4. The transaction iterates over the active readers of the shared objects it opened in write mode and updates their SON upper bound with its own SON.
  5. The transaction commits all the objects it opened in write mode by replacing the shared objects in the wrappers with their newer versions. Replacing the shared objects with their newer versions also releases them, completing the commit.

4.4 Performance Considerations

There are three sources of overhead that arise because of the use of $\mathcal{CS}$ to detect for conflicts among transactions. The first is the algorithmic overhead and it arises because of the higher complexity of the commit operations compared to 2PL. Given n threads and w writes in each transaction, the complexity of the commit operation is $O(n \times w)$ [4]. This is higher than $O(w)$ complexity of the commit operation under 2PL and visible readers.

The second source of overheads arises from the need to maintain information on reader transactions even after they commit. For this purpose, TSTM maintains a global readers table where each

[4] The details of the complexity analysis are omitted due to space considerations.

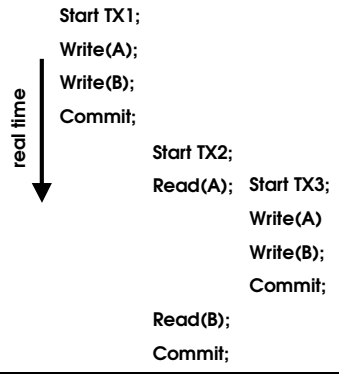


Figure 5. Multi-versioning.

shared object is mapped to an entry of the table. Each entry refers to the last reader transaction for each thread. Transactions insert themselves to the table when they read an object. However, they are not removed from the table until these read actions cannot potentially cause any aborts for future transactions. At this point, the transaction is said to be *retired*. Hence, the use of this global readers table introduces delays as transactions are determined to retire or not. The third source of overhead arises from the need to maintain SON ranges for each transaction. The need to access and update these values during execution results in additional cache misses, degrading performance.

5. Multi-Versioning

In its simplest form, $\mathcal{CS}$ can be verified by only keeping track of the latest committed values of objects. That is, once an object is committed by a transaction T, the old value of the object becomes redundant and discarded. Any transaction that later accesses this object conflicts with T.

We propose to further reduce abort rates and improve upon the use of $\mathcal{CS}$ by using *multi-versioning*, a concept that is used in the domain of databases [11]. In multi-versioning, each committed value of a shared object is called a *version*. Every time an object is committed, a new version of the object is created. The STM system keeps a number of past versions for each object, in addition to the latest/current versions. A transaction can read either the latest version of an object as well as one of its older versions kept. Reading an older version effectively creates the illusion that the read action took place earlier in time. Thus, a transaction can search for an appropriate version to read such that the read action will be serializable. That is, a transaction always attempts to read the latest version of an object, however, if this is not serializable, it iterates over the older versions. This avoids having to reject operations that arrive late to be serializable.

Consider the transactional actions shown in Figure 5. Normally, the operations of TX2 can not serialized because (1) object A is committed by TX3 after TX2 reads it and (2) TX2 reads object B committed by TX3. These two conflicts create a cycle in the serializability graph. However, if we assume that the action TX2:Read(B) happened earlier than the commit of TX3, TX2 would read *the version* of object B committed by TX1. This eliminates the second conflict that causes the schedule to be non-serializable. In this case, the equivalent serial schedule would be TX1, TX2, TX3. With multi-versioning, the old version of object B is kept even after a new version is created by TX3. This allows TX2 to read this old version committed by TX1, but not the latest one committed by TX3, serializing the read action with the remaining ones.

TSTM implements multi-versioning by linking each shared-object to its immediate older version. Wrappers point to the current versions of objects. Each version stores the SON value of the transaction that created it. When the wrapper is opened for read by a transaction, the SON values stored in versions are compared against the SON upper bound of the transaction, starting from the latest version. If the stored SON value is lower than the upper bound for a version, then this version can be accessed by the transaction without violating serializability. Thus, this version is returned by the read action and the lower bound of the transaction is updated based on the SON value stored in the version. Transactions abort only when all the available versions are iterated without finding an appropriate version to return.

Because versions are stored as linked-list, unlimited number of versions can be kept for each object. However, in order not to increase the memory footprint, we delete versions when all the transactions that were active when this version was created are completed (committed or aborted). Thus, an active transaction can always access any version of any object created since it started, but possibly not any versions created before.

6. Adaptive TSTM

The use of $\mathcal{CS}$ to enforce serializability introduces the overheads described above. Its main performance benefit stems from lowering the abort rate of transactions. Thus, it is likely for the use of $\mathcal{CS}$ to benefit performance in applications that have long-running transactions and extensive data sharing leading to high abort rates and high costs of re-running transactions. However, in applications where transactions are short-running and have infrequent data sharing, it is likely that the overhead of the use of $\mathcal{CS}$ dominates any benefit of using it. Indeed, in these cases, the use of 2PL is more appropriate.

Therefore, we propose a simple approach that uses 2PL when the abort rate is low and switches to use $\mathcal{CS}$ with multi-versioning when the rate is high. Our implementation starts with using 2PL. When the abort exceeds a certain threshold T_h, the implementation switches to using $\mathcal{CS}$ and multi-versioning. If the abort rate drop below a threshold T_l, the implementation switches back to 2PL. We opt to use two different thresholds $T_l < T_h$, in order to prevent thrashing, i.e. frequent switches between two approaches.

Our system not only switches between using 2PL and $\mathcal{CS}$, but also switches the visibility of reads. When using $\mathcal{CS}$ it uses visible-reads; whereas when using 2PL it uses invisible reads. This is because applications with short-running transactions and low abort rates tend to benefit from the use of invisible reads [18].

7. Experimental Evaluation

In this section, we provide an experimental evaluation of TSTM and the potential benefit of using $\mathcal{CS}$ to enforce serializability. We measure abort rates, timing breakdowns and the throughput of transactions. We compare the throughput of transactions in TSTM against their throughput on RSTM, a state-of-the-art obstruction-free STM system and on TL2, a state-of-the-art blocking system. Further, we compare throughput against coarse-grain and fine-grain locking. The results show that the overheads of checking for $\mathcal{CS}$ in TSTM do increase the execution time of transactions. However, the benefit of the reduction in the abort rates outweighs these overheads and significantly improves throughput for benchmarks that have high abort rates. Further, the results show that our adaptive technique is effective in avoiding the overheads for benchmarks with low abort rates. The performance of TSTM is superior to that of RSTM and TL2.

7.1 Experimental Framework

Our experiments are conducted on a Sun-Fire 15K cache-coherent multiprocessor with 72 UltraSPARC-III 1.2 GHz processors running Solaris 10. The system has 288 GB of RAM, in addition to 64KB of on-chip L1 and 8 MB external L2 data caches. We compile all benchmarks with GCC-3.3.1 at the -O3 optimization level. We run our experiments over a period of 5 seconds and average them over a set of 3 runs. Although, the machine has 72 processors, our experiments use only up to 32 processors. This is done to avoid the potential impact of other jobs that may be running on the system. Thus, in our experiments, the results of the 3 runs are consistent with one another over time and over different system loads.

Table 1. Benchmark characteristics.

	Duration	Abort rate
LL	197.17 microsec.	43.3%
RBT	10.97 microsec.	0.1%
GR	255.15 microsec.	49.1%

7.2 Benchmarks

- Linked-List (`LL`). This is a straight-forward implementation of an ordered linked-list, where each node contains a single key and the nodes are singly-linked to each other. Transactions perform a 1:1:1 mix of insert, delete and lookup operations. The keys range between 0 and 16383. There is extensive data sharing in this benchmark.
- Red-black Tree (`RBT`). This is the RBT implementation included in the `rstm_v2` package [19]. Each node in the tree contains a single key and transactions perform a 1:1:1 mix of insert, delete and lookup operations. The randomly generated keys range between 0 and 65535, which limits the depth of the tree to 16 when the tree is fully balanced. The tree structure limits the data sharing among transactions in this benchmark.
- Graph (`GR`). This benchmark implements an undirected graph, where each node contains a single key and linked to four randomly selected nodes. Nodes are arranged in a global ordered list which is used to locate nodes for linking and unlinking. Each node contains an ordered list of its neighbors. The key values ranging between 0 and 4095 and transactions perform a 1:1:1 mix of insert, delete and lookup operations. The access patterns in this benchmark are complex which causes extensive data sharing among transactions.

Table 1 shows the time it takes to execute a transaction on a single thread and the abort rates at 8 threads for each benchmark when using 2PL. The table shows that transactions in LL and GR take significantly longer to execute. This is due to the long chain of nodes that must be traversed in these benchmarks. In contrast, RBT has shorter transactions, mainly due to the tree structure. Furthermore, LL and GR suffer from frequent aborts but RBT has a low abort rate.

7.3 Evaluated STM Systems

The performance of the benchmarks is measured using several implementations of STM systems in addition to fine- and coarse-grain locking. The implementations are:

- TSTM. This is the version of our STM system described Section 4. It checks for $\mathcal{CS}$ and uses visible readers.
- TSTM-base. This is a version of TSTM that uses 2PL instead of $\mathcal{CS}$ to enforce serializability. Otherwise, it is identical to TSTM. We use it to measure the impact of testing for $\mathcal{CS}$.
- TSTM-ver. This is a version of TSTM that uses $\mathcal{CS}$ and that has multi-versioning enabled.
- TSTM-adaptive. This is the version of our STM system that uses the adaptive approach described in Section 6. The threshold values used for abort rates are $T_l = n \times 0.005 - 0.02$ and $T_h = n \times 0.005 + 0.02$, where n is the number of threads.
- RSTM. This is version-2 of the RSTM implementation provided with the `rstm_v2` package [19]. It is an obstruction-free system with an implementation and a programming model similar to that of TSTM. That is, it uses write-buffering and object-level conflict detection, and it requires objects to be opened in read or read-write modes before accesses to them. We use the Polite conflict manager and all four combinations of RSTM's options: visible vs. invisible readers and eager acquire vs. lazy acquire with/without the validation heuristics [20]. In our results, we only report the best performing option for each of the benchmarks.

Table 2. Abort rates of TSTM and RSTM at 8/24 threads.

	TSTM		TSTM-ver		RSTM	
	8	24	8	24	8	24
LL	12.5%	36.7%	5.3%	12.7%	43.3%	65.0%
RBT	0.3%	1.0%	0.2%	0.6%	0.1%	0.2%
GR	13.3%	54.7%	7.4%	50.1%	49.1%	71.2%

- Coarse-Grain Locking (CGL). In this implementation, a global spin lock is acquired at the start of each insert, delete and lookup operation, and released at the end. Thus, no concurrency is allowed and execution is completely serialized.
- Fine-Grain Locking (FGL). In this implementation each node is acquired with a spin lock separately only when it is accessed, thus allowing concurrency among threads at the expense of increased locking overhead. For RBT, we use Hanke's algorithm with concurrent writers and no-locking for reads, which is one of fastest implementations available [21].
- TL2. We use the original TL2 implementation that runs on 64-bit SPARC machines. The package includes includes a RBT implementation. We modified this RBT implementation to match our own and ported LL and GR benchmarks following TL2's programming model. We also increased the size of TL2's read table to enable the execution of our benchmarks without overflows.

7.4 Experimental Results

7.4.1 Abort Rates

Table 2 shows the ratio of aborted transactions with respect to the total number of transactions executed for LL, RBT and GR benchmarks at 8 and 24 threads. The table shows that TSTM and TSTM-ver suffer from significantly less abort rates compared to RSTM. For instance, RSTM aborts almost half the transactions executed in LL and GR with 8 threads, whereas TSTM's abort rate is at about 13% and TSTM-ver's abort rate is even lower at $5-7\%$. Furthermore, as the number of threads increases, the abort rates also increase for both systems due to higher conflicts caused by concurrent transactions. However, RSTM's abort rate is 65% at 24 threads for LL, whereas TSTM aborts 36.7% and TSTM-ver only aborts 12.7% of the transactions.

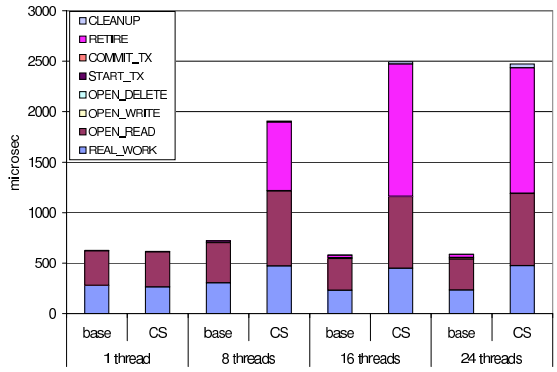
In the case of RBT, the abort rates are generally very low for both RSTM and TSTM. However, we note that the rates are slightly higher for TSTM and TSTM-ver when we expect them to be lower. This is due to the conservative handling of the race condition in TSTM, which was described earlier in Section 4.3 in the context of the commit operation. This handling causes a small amount of unnecessary aborts with short-running transactions and large number of threads.

7.4.2 Timing Breakdowns

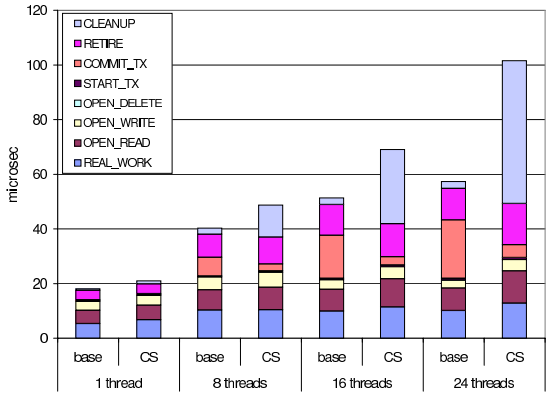
Figure 6 shows the breakdown of time spent in LL and RBT transactions at 1, 8, 16 and 24 threads for TSTM-base and TSTM. Due to space limitations, we only comment on the overall time it takes a transaction to execute and do not present breakdowns for the GR benchmark, which are very similar to those for LL. The figures clearly show that the execution time of a transaction increases with the use of $\mathcal{CS}$. For instance, the total time it takes to execute a transaction is almost the same for both TSTM-base and TSTM at one thread. However, as the number of threads increases, TSTM's transactions take longer due to the overheads. In fact, at 24 threads, TSTM transactions are almost 5 times longer for LL and 1.5 times longer for RBT.

7.4.3 Throughput

Figure 7 shows the throughput (successful transactions per second) for the three benchmarks and the seven implementations described against the number of threads. We discuss the throughput of each application in turn.



(a) LL, key range: 0-16383, 1:1:1 ins/del/lookup ratio.



(b) RBT, key range: 0-65535, 1:1:1 ins/del/lookup ratio.

Figure 6. Timing breakdowns at 1/8/16/24 threads.

For LL, the throughput of CGL remains constant at around 20,000 txs/sec, which is about 4 times better than the throughout of all STMs tested. The performance of RSTM remains constant but that of TSTM increases with the number of threads and surpasses that of CGL at 24 threads. TSTM-ver performs even better than TSTM because of its lower abort rates, especially at higher number of threads. In fact, the throughput of TSTM-ver exceeds that of CGL at 20 threads and outperforms it by 25% at 28 threads. TSTM-adaptive's throughput follows that of TSTM-ver because it switches to using CS and multi-versioning early on during execution. The higher throughputs of TSTM and TSTM-ver are despite of the fact the transactions execute five times slower compared to TSTM-base or RSTM. This clearly shows that the use of $\mathcal{CS}$ and multi-versioning to enforce serializability results in benefits that overcome the associated overheads. Indeed, the performance of TSTM and TSTM-ver surpass that of CGL without the use of any programmer directions, such as early release or open-nested transactions. FGL does not perform as well as CGL because of lock overheads and contention.

In the case of RBT, CGL at 1 thread performs 15 times better than the STM systems. However, as the number of threads increase, the throughput of CGL quickly drops. This is because of the contention over the lock, which is more significant in RBT than in LL because of the short-running transactions in RBT. RSTM performs well for this benchmark, but not as well as FGL. TSTM and TSTM-ver perform poorly, mainly because with the low rate of aborts in this benchmark. These versions of TSTM incur the overhead of testing for $\mathcal{CS}$ without reaping any benefits. However, TSTM-adaptive performs just as well as RSTM by selecting 2PL for this benchmark. The best performing case is FGL.

For GR, CGL at 1 thread is 9 times better than the STM systems, but its throughput gradually decreases with increasing the number of threads. RSTM does not perform well because of the high rates of aborts. In contrast, TSTM and TSTM-ver scale well up to 24 threads by reducing the number of aborts. TSTM-adaptive's performance follows that of TSTM-ver. The performance of TSTM drops sharply after 24 threads due to excessive overhead. TSTM-ver suffers less from these overheads due its lower abort rates. Finally, similar to LL, FGL performs poorly compared to CGL due to the overheads and contention.

For all the benchmarks, the throughput of TSTM-adaptive exceeds that of TL2. This is not surprising given that TL2 is highly optimized for short-running transactions with low abort rates.

8. Related Work

$\mathcal{CS}$ is a well-known concept in the database domain [11]. Thus, our contribution in this regard is to successfully employ and implement $\mathcal{CS}$ in the domain of STM systems. Our SON-based technique testing for $\mathcal{CS}$ is similar to that of Lindström and Raatikainen [17] for adjusting the serialization ordering of database transactions at runtime using timestamp intervals. However, our technique assumes write-buffering and adjusts the SON ranges while transactions are active. In contrast, they update their timestamp ranges only when transactions commit.

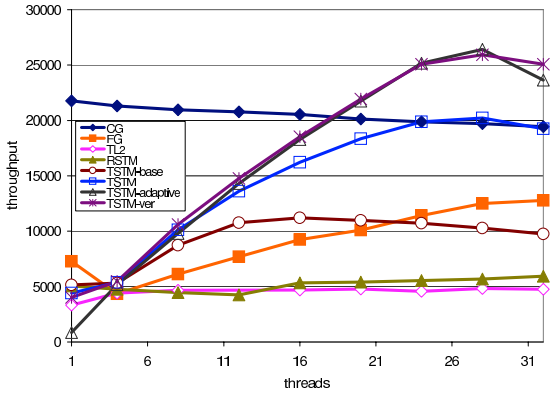
We also implement multi-versioning which is similar to *Multi-Version Concurrency Control* [11] used in the domain of databases. However, in Multi-Version Concurrency Control, transactions are typically assigned SON values when they start, whereas multi-versioning selects SONs for transactions when they commit based on their conflicting accesses. This effectively provides more concurrency by allowing more schedules to be serializable in the expense of higher commit time complexity.

There has been several approaches to improving the performance of applications with long-running transactions and/or transactions with high abort rates. These approaches are reviewed below. Common to these approaches is that they require the intervention of the programmer in one way or another. In contrast, our work successfully reduces conflicts in such applications without the need for any programmer effort. We believe it is the first to achieve such automatic reduction of conflicts in obstruction-free STM systems for long-running transactions.

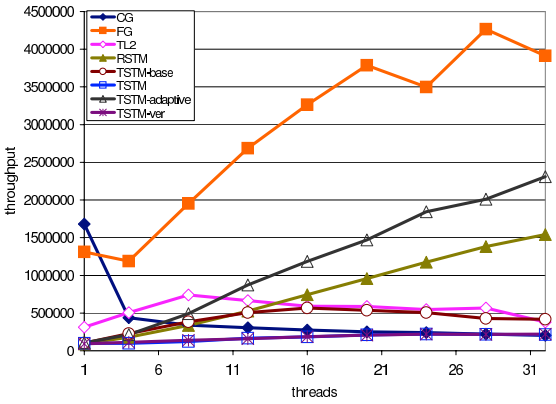
Herlihy et al. [4] and Skare et al. [8] have proposed the early-release technique and successfully use it to reduce the number of conflicts in applications that use linear data structures, such as linked lists. However, the use of early-release requires that the programmer be aware of the semantics of the application and ensure that accesses to the particular address do not affect the serializability of the transaction.

Ni et al. [9] and Calstrom et al. [10] propose open nested transactions as a mean of reducing conflicts for long-running transactions. With open-nesting, the programmer divides a transaction into a series of shorter transactions to avoid conflicts. This technique also requires knowledge of the application semantics and, thus, increases programmer effort.

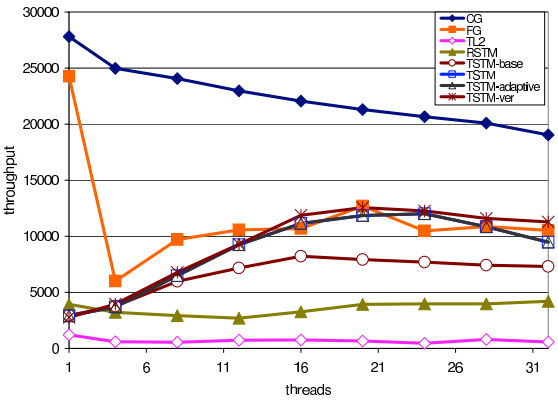
Calstrom et al. [7] assert that long-running transactions are important for the ease of parallel programming and recognize that the use of 2PL limits concurrency. They propose transactional collection classes to reduce the number of conflicts and demonstrate its use for some example data-structures. They manually determine which conflicts must abort and which conflicts should not. Hence, their approach requires knowledge of the semantics of data structures and the dependencies that exist. In contrast, our work provides a means for reducing conflicts in the same type of applications without programmer intervention.



(a) LL, key range: 0-16383, 1:1:1 ins/del/lookup ratio.



(b) RBT, key range: 0-65536, 1:1:1 ins/del/lookup ratio.



(c) GR, key range: 0-4095, 1:1:1 ins/del/lookup ratio.

Figure 7. Throughput(transactions/second).

Riegel et al. [5, 6] investigate the use of snapshot isolation, which allows transactions access old versions of shared data when there exist conflicts over the new versions. Snapshot isolation does not provide serializability nor linearizability when old versions are used. This requires that programmer should carefully analyze the accesses and transform some read accesses to write accesses for correctness. If linearizability is required, snapshot isolation again resorts to 2PL to ensure correctness. In contrast, our work provides serializability using $\mathcal{CS}$, which more relaxed than 2PL and linearizability.

Riegel et. al [22] contemplate the use of serializability in STM systems to increase concurrency and provide an algorithm for building a serializability graph and testing for cycles. However, they deem the approach as having too much overhead to be justified for all applications. In contrast, we use an efficient approximation of serializability and show that it is beneficial for applications. We further develop an adaptive approach that results in good performance across all applications.

9. Conclusions

In this paper, we implemented and evaluated the performance benefits of using $\mathcal{CS}$ and multi-versioning to enforce the serializability of transactions in obstruction-free STM systems. Our evaluation of our prototype implementation (TSTM) on a 32-processor system with 3 benchmarks reveals that the use of $\mathcal{CS}$ significantly reduces the abort rates and leads to performance improvements in spite of overheads. The evaluation also shows that the performance of our prototype surpasses that of two competitive STM systems that use 2PL.

Existing STM systems have been shown effective on applications with short-running transactions and low abort rates. However, their performance to date lacks for applications that have long-running transactions and high abort rates. The use of $\mathcal{CS}$ is more effective in such applications, leading to better and more scaling performance. Thus, our approach of using $\mathcal{CS}$ makes STM systems more effective for these applications. Furthermore, our adaptive approach ensures that the 2PL and $\mathcal{CS}$ are used when each is more effective. Indeed, to the best of our knowledge, we believe that our approach is the first to outperform coarse-grain locking for a linked list application without the use of manual techniques such as early release or open nesting.

10. Acknowledgments

We thank the anonymous reviewers for their comments.

References

[1] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, “Mcrt-stm: a high performance software transactional memory system for a multi-core runtime,” in *Proc. of PPoPP*, pp. 187–197, 2006.

[2] D. Dice and N. Shavit, “What really makes transactions faster?,” in *TRANSACT, http://research.sun.com/scalable/pubs/TRANSACT2006-TL.pdf*, 2006.

[3] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, “Lowering the overhead of software transactional memory,” Tech. Rep. TR 893, Computer Science Department, University of Rochester, 2006.

[4] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, “Software transactional memory for dynamic-sized data structures,” in *Proc. of PODC*, pp. 92–101, 2003.

[5] T. Riegel, C. Fetzer, and P. Felber, “Snapshot isolation for software transactional memory,” in *TRANSACT, http://wwwse.inf.tu-dresden.de/papers/preprint-riegel2006sistm.pdf*, 2006.

[6] T. Riegel, P. Felber, and C. Fetzer, “A lazy snapshot algorithm with eager validation,” in *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, vol. 4167, pp. 284–298, Sep 2006.

[7] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun, “Transactional collection classes,” in *In Prof. of PPoPP*, pp. 56–67, 2007.

[8] T. Skare and C. Kozyrakis, “Early release: Friend or foe?,” in *Proc. of the Workshop on Transactional Memory Workloads*, pp. 45–52, 2006.

[9] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman, “Open nesting in software transactional memory,” in *Proc. of PPoPP*, pp. 68–78, 2007.

[10] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun, "The atomos transactional programming language," in *Proc. of PLDI*, pp. 1–13, 2006.

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

[12] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM TOPLAS*, vol. 12, no. 3, pp. 463–492, 1990.

[13] N. Shavit and D. Touitou, "Software transactional memory," in *Proc. of PODC*, pp. 204–213, 1995.

[14] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Proc. of OOPSLA*, pp. 388–402, 2003.

[15] K. Fraser, "Practical lock freedom," Tech. Rep. UCAM-CL-TR-579, Cambridge University Computer Laboratory, Feb 2004.

[16] C. H. Papadimitriou, "The serializability of concurrent database updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, 1979.

[17] J. Lindstrom and K. Raatikainen, "Dynamic adjustment of serialization order using timestamp intervals in real-time DBMS," in *Proc. of the Int'l Conf. on Real-Time Computing Systems and Applications*, 1999.

[18] V. J. Marathe, W. N. S. III, and M. L. Scott, "Adaptive software transactional memory," in *Proc. of the Int'l Symp. on Distributed Computing*, 2005.

[19] "http://www.cs.rochester.edu/research/synchronization/rstm/."

[20] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott, "Conflict detection and validation strategies for softwaretransactional memory," in *Proc. of ISDC*, pp. 179–193, 2006.

[21] S. Hanke, T. Ottmann, and E. Soisalon-Soininen, "Relaxed balanced red-black trees," in *Proc. of CIAC*, pp. 193–204, 1997.

[22] T. Riegel, H. Sturzrehm, P. Felber, and C. Fetzer, "From causal to z-linearizable transactional memory," Tech. Rep. RR-I-07-02.1, Universite de Neuchatel, Institut d'Informatique, Feb 2007.