Conflict Avoidance in Software Transactional Memory

Pinar TOZUN

LPD, I&C, EPFL

Abstract—Transactional Memory aims to become a superior alternative to lock-based concurrent programming. However, it is not widely deployed because it does not scale well under high contention scenarios. Aborts affect performance of a concurrent program that uses Transactional Memory negatively when there is high contention. A high abort rate is a huge burden for the performance of the program since aborts cause wasted work and execution time. Therefore, the transactions should run in parallel in a way that the abort rate is as low as possible. Previous research shows that reducing abort rates can improve performance significantly. This research proposal presents some previous work on avoiding conflicts in Software Transactional Memory and what can be done to further improve the existing techniques.

Index Terms—Concurrent Programming, Software Transactional Memory, Contention Management, Conflict Avoidance

I. INTRODUCTION

CONCURRENT programming becomes increasingly important in order to take advantage of the fast developing multi-core architectures effectively. Developing reliable concurrent programs is a challenging task. One has to maintain the consistency of the shared data as it is being used by multiple threads in parallel.

Proposal submitted to committee: May 31st, 2010; Candidacy exam date: June 07th, 2010; Candidacy exam committee: Willy Zwaenepoel, Rachid Guerraoui, George Candea.

This research plan has been approved:

Date: __________________________

Doctoral candidate: __________________________
(P. Tozun) (name and signature)

Thesis director: __________________________
(R. Guerraoui) (name and signature)

Thesis co-director: __________________________
(if applicable) (name and signature)

Doct. prog. director: __________________________
(R. Urbanke) (signature)

So far, lock-based techniques are used to protect the shared data. However, these techniques are error prone (i.e., they suffer from deadlocks, data races), especially when fine-grained locking is used, or do not scale as parallelism increases with the increasing number of cores, when coarse-grained locking is used. Therefore, the task of writing concurrent programs has to be simplified so that more programmers can benefit from it.

Transactional Memory (TM) [1] is a concurrent programming abstraction. It is meant to become a better alternative to lock-based concurrent programming since it is as easy as using coarse-grained locking and as efficient as using fine-grained locking.

A transaction is a sequence of accesses to shared memory locations. It can either be successful and commit, making the changes it performed on the shared data visible, or fail and abort, rolling back all of the changes it has done. To ensure the safe access to the shared data invisible to the programmer, TM checks for conflicts among the transactions and tries to resolve them.

A transaction is a sequence of accesses to shared memory locations. It can either be successful and commit, making the changes it performed on the shared data visible, or fail and abort, rolling back all of the changes it has done. To ensure the safe access to the shared data invisible to the programmer, TM checks for conflicts among the transactions and tries to resolve them.

A transaction occurs between two transactions running in parallel when they access the same shared memory location, and at least one of these accesses is a write. The conflicts can be detected eagerly, lazily, or in a mixed way. In eager detection, whenever a transaction writes to a shared memory region, it is checked whether any other transaction has read from or written to that region. This allows the conflicts to be detected earlier so that a transaction that is bound to abort is prevented from performing further wasted work. However, checking for conflicts in each write access might incur too much overhead, especially under low contention scenarios (i.e., when there is not much shared data). In lazy detection, the conflicts are checked at the commit time of a transaction. Although lazy detection eliminates some unnecessary conflict checks that happen in eager detection, a transaction that is bound to abort will run in vain. The mixed techniques combine both eager and lazy detection methods by preferring one over the other in different cases.

When a conflict is detected by TM, it is resolved by aborting one of the transactions that are in conflict and letting the other one proceed. Choosing which transaction to abort and how to restart the aborted transaction is the conflict resolution problem. It is solved based on a policy, which is usually called the contention management policy. Several contention management policies are proposed, each having their advantages and disadvantages. Some abort the transaction that discovers the conflict [2], which might result in
a livelock when two transactions abort each other repetitively. Another policy, Polka [3] is based on giving priority to the transactions. A transaction’s priority is increased as it waits due to being aborted, and the transaction with the lower priority is aborted. It usually gives better results than the one presented in [2]. However, it can also cause livelocks. Another approach, Greedy [4] is based on timestamps. Each transaction’s timestamp is increased monotonically after it starts, and the transaction with the lower timestamp is aborted. Although Greedy prevents livelocks, it prioritizes the long transactions over the short ones, which might not be preferred all the time.

Overall, aborts continue to be a bottleneck when a concurrent program using TM has bad performance. They increase the execution time and waste computing resources because they are wasted work performed by some transactions. They prevent both transactional and non-transactional code from performing useful work.

Ideally, transactions should be scheduled in a way that the number of aborts is minimized. Therefore, the conflict resolution problem is also a scheduling problem. Until recently, the work on contention management has not put much emphasis on scheduling based solutions for reducing aborts in TM. Thus, the previous contention management policies depend on transaction-ignorant operating system scheduler for scheduling the threads that execute transactions, and give unpredictable performances.

Finding the ideal scheduling for transactions is possible in theory, but very challenging in practice. Some information related to the applications that use TM can be given or all the possible schedules for the transactions can be tried a priori. However, these types of solutions are not feasible in most of the cases.

In this proposal, §II describes some of the recent work on avoiding conflicts by using scheduling-based solutions in Software Transactional Memory (STM) [5]. Then, §III presents how I plan to improve the existing work in this area.

II. Survey of the Selected Papers

In this section, I present three previous techniques that have been proposed recently in order to reduce the number of aborts to get better performance and not to waste the computing resources when STM is used.

§II-A, §II-B, and §II-C describe CAR-STM [6], Steal-on-abort [7], and transaction-aware OS-scheduler [8], respectively.

A. CAR-STM: Scheduling-Based Collision Avoidance and Resolution for Software Transactional Memory

CAR-STM maintains per-core transaction queues, which keep the transactions to be executed, and threads, which execute the transactions from the corresponding transaction queue one at a time. It proposes two scheduling-based techniques to reduce the abort rates in an application that uses STM: serializing contention manager and pro-active collision reduction. Serializing contention management aims to prevent repeated aborts, and pro-active collision reduction aims to prevent conflicts before they ever happen.

Repeated aborts happen when a restarted transaction is aborted because of the same conflict that had caused it to abort in the previous case. When a conflict is detected between the two transactions, the contention manager decides which transaction should abort and which transaction should continue its execution. The aborted transaction is restarted after some time, depending on the contention management policy. However, if the other transaction has not finished its execution yet, these two transactions may conflict again. This second conflict is called a repeat conflict, and the abort that happens due to this conflict is called a repeated abort.

To avoid repeated aborts, previous work has focused on applying exponential delays before restarting the aborted transaction [3]. However, this technique is not sufficient to get good performance in general for TM. First, exponential delays might not be enough when there is a conflict between a long and a short transaction, and the short transaction is aborted. This case may result in many repeated aborts. Second, an aborted transaction might be delayed more than necessary. Third, exponentially delaying a long transaction does not reduce its chances of being in conflict again. Therefore, CAR-STM proposes a new contention management policy.

Serializing contention management is an alternative contention management policy to prevent repeated aborts. The policy of this contention manager is: when two transactions conflict and one of them is aborted, the aborted transaction is moved to the queue of the other transaction in order to serialize the two transactions and ensure they will never conflict again.

CAR-STM proposes two types of serializing contention managers: the basic serializing contention manager (BSCM) and the permanent serializing contention manager (PSCM). For both BSCM and PSCM, every transaction is initialized with a timestamp. When a conflict is detected between the two transactions $T_{old}$ and $T_{new}$, the newer transaction, $T_{new}$, is aborted, and the older one, $T_{old}$, continues its execution.

In BSCM, $T_{new}$ is moved to the end of the transaction queue of $T_{old}$. The probability of these two transactions conflicting again is reduced but not eliminated. There might be another transaction, $T_{other}$, which conflicts with $T_{old}$. $T_{other}$ can cause $T_{old}$ to abort and be moved to the transaction queue of $T_{other}$. In this scenario, $T_{new}$ and $T_{old}$ are not serialized anymore. Thus, they may run in parallel in the future and conflict again.

In PSCM, however, two transactions can be in conflict only once. PSCM maintains a subordinate transactions list for each currently running transaction. Whenever a transaction is aborted and has to be moved, it is moved together with its subordinate transactions to the subordinate transactions list of the other transaction. When a transaction finishes its execution and commits, its subordinate transactions are scheduled right after it. For the above scenario, $T_{new}$ will be put into $T_{old}$’s subordinate transactions list after they conflict and $T_{new}$ is aborted. Then, when $T_{old}$ has to be aborted and moved due to its conflict with $T_{other}$, it will be moved together with $T_{new}$ to the subordinate transactions list of $T_{other}$. This way it is guarantied that the two previously conflicted transactions will
CAR-STM Architecture

never conflict again in the future since they are guaranteed to not to run in parallel.

However, it would be even better if a conflict is avoided without being encountered at all. CAR-STM proposes pro-active collision reduction to achieve this goal.

**Pro-active collision reduction** aims to prevent the first occurrence of some possible conflicts. CAR-STM allows the applications to provide some a priori information about the conflict probability of the transactions, and puts transactions that have a higher probability of conflicting into the transaction queue of the same core. This way the transactions that are more likely to conflict are serialized from the start, and their conflict is prevented before it is ever encountered.

CAR-STM’s architecture has the following components; *dispatcher, collision avoider, per-core transaction queues* and *threads* (see Figure 1). Each cores’ thread, *TQ-thread*, takes the transaction that it will execute, from the head of the transactions queue, *TQ*, that is kept for the corresponding core. Each transaction is kept as a structure which has the transaction’s wrapper method, data, information, thread, and a condition variable for the transaction’s thread. To make a transaction, *T*, be executed by the *TQ-thread* instead of the transaction’s thread, *t*, that is provided by the application, when *t* wants to execute *T*, it calls the *dispatcher*. *Dispatcher* first gives the thread information to the *collision avoider*. *Collision avoider* computes the probability of *T* being in conflict with any of the currently running transactions, based on the conflict probability function and the transaction information provided by the application. Then, the *dispatcher* puts *T* into the *TQ* whose corresponding *TQ-thread* is executing the transaction that has the highest possibility of conflicting with *T*. *T’s* condition variable is held by the *TQ-thread* that executes *T* until *T* commits. *TQ-thread* executes *T*, based on its data and wrapper method, which indicates what code-block *TQ-thread* should execute.

CAR-STM is incorporated to RSTM [9]. It is evaluated by using STMBench7 [10]. Since STMBench7 does not provide a conflict-probability function, a synthetic application, which allows them to use a natural conflict-probability function is implemented to evaluate the pro-active collision reduction technique. STMBench7 is chosen because it allows to simulate various real-world workloads for an application that uses STM. It has options for long and short transactions; read, write and read/write dominated workloads; deterministic and non-deterministic transactions.

The results indicate that CAR-STM improves both the efficiency and stability of performance. It improves efficiency because the repeat conflicts are reduced or completely eliminated effectively. The waste of execution time and computing resources caused by repeated aborts are prevented. It improves stability because it reduces the unpredictable scheduling of the transactions due to transaction-ignorant OS-scheduler. Depending on how transactions are scheduled by the OS-scheduler initially, the transactions that have to run in parallel might change significantly, which means the probability of conflicting transactions running in parallel might change significantly among different executions. However, CAR-STM greatly reduces this unpredictability by re-scheduling the transactions based on the previous conflicts.

BSCM gives better results than PSCM. The reason is that maintaining the data structures in PSCM requires more work and overhead since a block of transactions has to be kept and moved between transaction queues. Moreover, the serializing contention manager is much more effective than the pro-active collision reduction for improving the performance of an application because having a precise conflict probability function for an application is not that straightforward.

CAR-STM has some drawbacks. First, it is not orthogonal with the existing contention managers. Although, it is easy to incorporate CAR-STM with the existing STMs, one has to change the existing contention manager policy of the STM, which might not be preferred all the time. There might be better policies while deciding which transaction to abort depending on different scenarios.

Second, the overhead introduced by CAR-STM to maintain its necessary data-structures might hurt performance for the cases where there is low concurrency or there are mostly short transactions being executed. In both of these cases, the chances of the repeated aborts are smaller. Since the evaluation results show that the main impact of CAR-STM is on reducing the repeated aborts, when there are not many repeated aborts, CAR-STM overhead might be a bottleneck for performance.

Finally, CAR-STM reduces the level of parallelism in many ways. First, it does not allow any parallelism in the same core. Although, preempting a thread that executes a transaction might increase the chances of a conflict for that transaction, due to the increasing execution time, not allowing preemptions in the same core brings some problems. For example, in the real world people use multi-programming systems. Not letting parallelism in a core in such a system might harm the performance of the other applications. Second, it does not...
perform any load balancing among the cores. Since different transactions have different lengths, and the transactions are moved from one transaction queue to another, some transaction queues might become empty before the other ones. This causes some cores to stay idle when some cores are still working. Third, let’s think of a scenario where the transactions $T_A$ and $T_B$, and $T_A$ and $T_C$ have conflicted. Both $T_B$ and $T_C$ are aborted and moved to the transaction queue of the core where $T_A$ is being executed. Here, $T_A$, $T_B$ and $T_C$ are serialized all together, although $T_B$ and $T_C$ might not be conflicting transactions. CAR-STM prevents $T_B$ and $T_C$ to run concurrently, although they might run in parallel without any problems, and thus hinders parallelism.

B. Steal-on-abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering

Like CAR-STM, Steal-on-abort also aims to prevent the repeat conflicts(§II-A). When two transactions, $T_1$ and $T_2$, conflict, and $T_2$ is aborted, it is not easy to detect this first conflict. However, it is logical to serialize these two transactions after their first conflict in order to prevent their repeat conflicts because due to close temporal locality the immediately restarted aborted transaction, $T_2$, has a high chance of conflicting with $T_1$ again.

Steal-on-abort maintains per-thread transaction queues, which keep the transactions to be executed by each thread. It proposes the idea of the abortee transaction stealing the aborted transaction to serialize the two transactions. When two transactions, $T_1$ and $T_2$, conflict, and $T_2$ is aborted, the thread that executes $T_1$ steals $T_2$ from the thread that executes $T_2$, and releases it only after $T_1$ commits. This way $T_2$ is scheduled after $T_1$, and their repeat conflicts are prevented.

Steal-on-abort is implemented in DSTM2 [11], which basically creates some number of threads to execute the transactions concurrently. Steal-on-abort extends DSTM2’s model by using a thread pool model. When an application thread wants to execute a transaction, the transaction is submitted to the transactional thread pool (TTP) as a transactional job, which keeps the information needed to execute a transaction. Moreover, a thread-safe work queue and steal queue, which are called $mainDeque$ and $stealDeque$ respectively, are added to each transactional thread in TTP. The transactional jobs, that are submitted into the TTP, are distributed to the threads’ $mainDeques$ in a round-robin manner.

Two types of steal operations are implemented; abort-steal and work-steal. Abort-steal is for applying the steal-on-abort technique. When the two transactions, who are currently being executed by two threads, $t_a$ and $t_b$, conflict, and the transaction executed by $t_b$ is aborted, $t_a$ steals the current transaction of $t_b$, and places it into its $stealDeque$. After $t_a$ finishes stealing $t_b$’s current transaction completely, $t_b$ gets another transaction execute to execute from the head of its $mainDeque$. The transaction that has been stolen by $t_a$ due to this abort-steal operation is called an abort-stolen transaction. Work-steal aims to provide a good load-balance among threads. When both the $mainDeque$ and the $stealDeque$ of a thread $t$ become empty, $t$ randomly picks a thread, and takes the transactional job at the tail of the randomly picked thread’s $mainDeque$. If the randomly picked thread’s $mainDeque$ is also empty, then $t$ randomly picks another thread. If the $mainDeques$ of all the threads are empty, then a transactional job from a randomly picked thread’s $stealDeque$ is stolen. The transaction that has been stolen due to the work-steal operation is called a work-stolen transaction.

Steal-on-abort investigates four different strategies that differs in how the stolen transactions should be released. These strategies are steal-tail, steal-head, steal-keep, and steal-block.

- In steal-tail, the transactions in the $stealDeque$ of a thread are inserted at the tail of the $mainDeque$ of that thread after the current transaction commits or becomes abort-stolen. Therefore, the abort-stolen transactions will be executed last by that thread, unless they are work-stolen by another thread. This technique is good when the shared data accesses of a transaction is closely related to its creation time. In this case, the transactions that are created at close times will have higher chances of conflicting. Since the transactions are distributed in a round-robin manner, the transactions that are created at close times will have more chances of being executed in parallel because they will be put into different threads’ $mainDeques$. For this scenario placing an abort-stolen transaction at the end of the $mainDeque$ and executing it last will reduce its possible conflicts with other transactions.

- In steal-head, the transactions in the $stealDeque$ of a thread are inserted at the head of the $mainDeque$ of that thread after the current transaction commits or becomes abort-stolen. Therefore, the abort-stolen transactions will be executed first by that thread unless they are work-stolen by another thread. Since an abort-stolen transaction has just executed recently, some of the data elements that it has to use might be still in cache from its previous execution. Therefore, this technique might be good for benefiting from cache-locality, if the scenario presented in steal-tail is not the case.

- In steal-keep, the transactions in the $stealDeque$ of a thread are not moved to the $mainDeque$ of that thread when the current transaction commits or becomes abort-stolen. They are moved only after there are no more transactions left in the $mainDeque$ of that thread. Therefore, the abort-stolen transactions cannot be work-stolen so quickly like in the case of steal-tail. With this technique, the overhead of moving transactions from $stealDeque$ to $mainDeque$ for each committed or abort-stolen transaction is avoided. However, the overhead in work-steal is increased because the $mainDeques$ become empty sooner.

- In steal-block, when a transaction will be abort-stolen, it is stolen with all the transactions it caused to be abort-stolen. Steal-block is useful under the scenarios where the transactions have a strong shared data access affinity; meaning that when transactions $T_A$ and $T_B$, and $T_A$ and $T_C$ conflict, $T_B$ and $T_C$ have high chances of conflicting. This way the conflicting transactions are
serialized faster. However, stealing transactions as blocks brings more overhead to the abort-steal operation since the stealDeque has to be traversed for each aborted transaction.

Steal-on-abort implementation for this paper is evaluated with four different benchmarks; sorted linked list [12], red-black tree [12], STAMP-vacation [13], and Lee-TM [14]. Sorted linked list and red-black tree benchmarks have short transactions while the other two have long transactions. The benchmarks are run with inputs that cause high contention. Polka [3] is used as the contention manager. However, since steal-on-abort acts only after the contention manager makes its decision on which transaction to abort, it can be coupled with any contention manager. The evaluation shows that steal-tail and steal-keep give the most stable results. The effectiveness of steal-head and steal-block change from benchmark to benchmark. However, in general all the strategies improve the performance of the application, and reduce wasted execution time and work caused by aborts.

Although, the evaluation shows the benefits of steal-on-abort, the technique has some limitations. The main limitation is that the technique is useful only for repeat conflicts. It cannot prevent the first occurrence of a conflict, and is not useful when there are not many repeat conflicts. Moreover, in order to have repeat conflicts the STM must check for the conflicts eagerly. When the conflicts are checked lazily, they are detected between two transactions when one is active and the other one has already committed. Since, one of the transactions in the conflict has already committed, the two transactions cannot be in a repeat conflict again. Therefore, steal-on-abort is not suitable for all STM implementations. Also, like in CAR-STM, for low concurrency levels and applications that have mostly short transactions, maintaining the steal-on-abort data-structures might introduce a bottleneck for the performance.

C. Scheduling Support for Transactional Memory Contention Management

Like CAR-STM and Steal-on-abort, the main aim of this paper is to reduce the repeat conflicts(§II-A, §II-B). However, unlike the previous user-level solutions, this paper investigates kernel-level scheduling techniques in order to solve the repeated conflicts problem.

The user-level techniques for serializing conflicting transactions have several problems. They incur high overhead for low concurrency levels and short transactions. The reason for this overhead mainly comes from maintaining the data-structures that keep the transactions to be executed, and increased kernel-level and user-level communication due to replicating the kernel-level scheduling operations in the user-level.

In the user-level approaches, when a transaction is aborted, it is moved to some data-structure (i.e., transactions queue in CAR-STM, stealDeque in steal-on-abort), that is maintained for the conflicting transaction’s thread, so that the conflicting transactions are serialized. The thread that has been executing the aborted transaction has to request another transaction to execute. Assume the conflicted transactions are short transactions. If the aborted transaction had been restarted immediately, before its restart, the conflicting transaction might have already committed. Hence, by not letting the aborted transaction restart, moving it to another data-structure, and requesting another transaction to execute, the overhead of the user-level serialization techniques becomes a bottleneck in this case.

Moreover, the user-level approaches replicate the kernel-level scheduling operations in the user-level. Taking some scheduling decisions for the transactions at the user-level, requires more communication between the OS-scheduler and user-level STM library. This communication is done by system calls, which can be costly if they are needed frequently.

The main problem when scheduling the threads that execute transactions is that the OS-scheduler is transaction-ignorant. Thus, applications that use TM, can give very unstable performances depending on how the transactions are scheduled by the transaction-ignorant OS-scheduler, as it is also reported in CAR-STM( §II-A).

This paper proposes several user and kernel-level scheduling support policies for STM and ways to handle communication between the kernel-level and user-level efficiently.

The different scheduling support strategies presented in this paper are called SER-u (spin), SER-u (cond), SER-k, YIELD, SOFтвер-u, and SOFтвер-k. The paper aims to investigate which strategy is better under what circumstances. For the explanation of these strategies $t_w$ and $t_l$ denote two threads who are running in parallel and executing two transactions, $T_w$ and $T_l$ respectively, that are in conflict. Due to the conflict, $T_l$ is aborted while $T_w$ continues running.

- SER-u (spin) is not a contention management strategy proposed by this paper. It has been provided by TINYSTM [2]. In this strategy; when $T_l$ is aborted, $t_l$ waits on a spinlock held by $t_w$. When $T_w$ commits, $t_w$ releases its spinlock so that $T_L$ can be restarted. This way the two conflicting transactions are serialized after their first conflict is encountered, and their repeat conflict is prevented. However, this technique can waste the computing resources drastically since it does not let the thread that executes the aborted transaction leave the CPU for other threads to execute.

- SER-u (cond) is also not proposed by this paper. It tries to solve the problem in SER-u (spin) by using condition variables instead of spinlocks. Using condition variables for serializing two conflicting transactions instead of spinlocks, makes the aborted transaction be blocked until the condition variable is released by the other conflicting transaction. This way the thread that executes the aborted transaction leaves the CPU to other threads, thus the computing resources are not wasted. However, this technique requires communication between the user and kernel level whenever an abort or commit happens. For the above example, when $T_L$ aborts, the OS-scheduler has to be informed about $t_l$, which has to wait for the condition variable that is held by $t_w$, so that it can remove $t_l$ from the ready queue of the OS-scheduler after $T_L$ is aborted, and when $T_w$ commits, OS-scheduler
again has to be informed about $t_w$ releasing its condition variable, so that it can move back $t_l$ to the ready queue. In SER-u (cond), this communication is handled by system calls. Since for each abort and commit a system call has to be made, SER-u (cond) can degrade performance.

- **SER-k** is proposed to solve the problem in SER-u (cond) with a shared memory based implementation. It aims to provide a more efficient communication between the user and kernel level compared to system call based communication. To achieve this goal, it uses some memory regions as shared regions between the OS-scheduler, kernel-level, and STM library, user-level. The communication between the OS-scheduler and the STM library is simply maintained by filling data in these shared regions, thus they do not have to interact directly.

This technique requires some kernel-level implementation, too. When the conflict happens, $T_L$ invokes the OS-scheduler in order to remove $t_l$ from the ready queue. However, the OS-scheduler performs two checks for all threads, $t$; it removes all the threads, that have to wait on the condition variable held by $t$, from the ready queue, and moves all the threads, that has been waiting on the condition variable released by $t$, back to the ready queue. By turning these two operations to kernel-level checks rather than user-level to kernel-level system calls, this technique eliminates all the system calls that have to be made in each commit, and it reduces the overhead. However, since the kernel-level check is not performed in every commit, there is some latency on moving the threads like $t_l$ back to ready queue after $T_W$ commits.

- **YIELD** is proposed to omit the system call in SER-u (cond) at commit time and the check in SER-k that is performed in order to awake the blocked thread, $t_l$. When $T_L$ aborts, it performs a yield system call in order to yield $t_l$. The yielded thread, $t_l$, will have a low priority, so it has to leave the CPU to other threads for sometime. Then, it will be rescheduled and $T_L$ will be restarted. This technique is good if $T_W$ is a short transaction because $T_W$ will probably manage to finish its execution, and commit before $T_L$ is restarted. Since, the check for the blocked threads is eliminated, the overhead will be lower. However, if $T_W$ is a long transaction, probably it will not commit before $T_L$ is restarted, and the two transactions might be in conflict again. Therefore, this technique does not prevent all the repeat conflicts.

- **SOFTSER-u** aims to eliminate the strictness in serialization of the transactions in the SER-u technique. SER-u like techniques are based on the fact that if two transactions have conflicted, they have a very high chance of conflicting again when the aborted transaction is restarted. However, a transaction might be non-deterministic. For example, if $T_L$ makes some branching decisions based on some shared data, the shared data might be changed during the time period that passed between the first and second execution of $T_L$. This might cause $T_L$ to follow a completely different branch in its second execution, and not be in conflict with $T_W$ again. For these types of non-deterministic scenarios, serializing $T_L$ and $T_W$ might be too strict.

SOFTSER-u maintains a conflict matrix $C$. $C[T_L][T_W] = true$ means $T_L$ is aborted due to its conflict with $T_W$. When $T_L$ aborts, it performs a system call to set the priority of $t_l$ to LOW, and when $T_W$ commits, the priority of each thread, that executes a transaction that has to be aborted due to $T_W$, is set back to NORMAL by a system call unless they are not in conflict with any other transactions. Although, this technique eliminates the drawbacks of strictly serializing the non-deterministic transactions, it might incur a high overhead due to the checks and system calls being made upon each commit.

- **SOFTSER-k** is the shared memory based implementation of SOFTSER-u. The difference between SOFTSER-k and SOFTSER-u is same as the difference between SER-k and SER-u.

The paper also proposes the **time-slice extension policy** which aims to reduce the risk of conflicts for a transaction. When there are more threads than cores, threads are preempted when their time slice has expired or when a higher priority thread is unblocked. However, preempting a thread that executes a transaction will increase the execution time, hence the chances of conflicts for that transaction. Therefore, the time-slice extension policy checks whether a thread that is about to be preempted executes a transaction or not. If it does, then it gives that thread one more time slice, a time slice extension, so that it can finish its execution without any preemptions caused by time slice expiration. However, to prevent long transactions from monopolizing the CPU, a counter is associated with each thread, that keeps the number of extensions it got, and a thread is prevented from getting a time slice extension after some number of times. Moreover, to prevent threads that execute transactions from monopolizing the CPU, a thread is yielded after the transaction it executes commits if it has got an extension before. Time-slice extension policy is implemented based on shared memory regions since there are no system calls in Linux for time-slice extension. It can be coupled with any of the scheduling strategies above since it affects each thread that executes a transaction individually rather than in pairs.

The kernel-level scheduling support is implemented in the Linux and Solaris kernels. As the user-level STM library TINYSTM [2] is used. The benchmarks that are used to evaluate the proposed strategies are skip list [12], red-black tree [12], linked list [12], STMBench7 [10], and STAMP [13]. All these benchmarks show different characteristics which provides an evaluation under various workloads. The results show that different strategies work well under different workloads depending on the length of the transactions, the ratio of number of threads to cores, and the level of contention. Therefore, we cannot conclude that one strategy is strictly better than the other. However, it is shown that the strategies that are proposed here can be strong alternatives to the previous user-level based scheduling support strategies for STM.

There are two main limitations of the kernel-level scheduling support strategies. First one is, the latency they incur before restarting an aborted transaction. The aborted transactions are not moved back to the ready queue immediately after the trans-
action that caused them to abort commits, and they have to wait an abort event to happen so that a call to the OS-scheduler is made. Second one is, the kernel-level implementation is more complicated than a user-level implementation and it has to be changed from architecture to architecture.

III. RESEARCH PROPOSAL

In this section, I will present my initial ideas about how I plan to improve the existing work on scheduling based contention management techniques in STM.

A. Overview

Scheduling based contention management techniques provide powerful solutions for the contention management problem in STM. In general, they try to serialize the transactions that have a high probability of conflicting so that they do not run in parallel and conflict. They aim to reduce contention and the number of aborts in an application that uses STM. This way, the wasted execution time and computing resources that are caused by aborts can be reduced.

The previous scheduling based contention management techniques are usually based on the following assumption: when two transactions conflict and one of them is aborted, the aborted transaction is usually restarted immediately. If the other transaction is still running then these two transactions have a very high probability to be in conflict again. Therefore, they serialize the two transactions that have been in conflict once. Hence, they are focused on reducing repeat conflicts.

In the previous sections, I have described three approaches that are proposed for providing a scheduling support for STM. Their evaluation shows that each of them complements or improves the previous contention management techniques. However, they all have some limitations.

User-level scheduling support techniques, like CAR-STM (§II-A) and Steal-on-abort (§II-B) provide good benefits for performance. However, under low contention scenarios or when there are mostly short transactions in an application, the overhead of maintaining the data-structures of these techniques might be a bottleneck for the performance of the application. Kernel-level scheduling support techniques, like the ones presented in §II-C, mainly aim to reduce the overhead that comes from the communication between the kernel-level, OS-scheduler, and the user-level, STM library. They provide a communication between these two layers by using shared memory regions rather than costly system calls. Moreover, they move some scheduling operations required for these techniques to kernel-level to reduce the replication of some scheduling operations in user-level because the data-structures that have to be maintained for these techniques are usually due to this replication. However, the main drawback of these techniques is the latency that they incur before restarting the aborted transactions. They also require a more complicated implementation and the implementation has to be changed for the different system architectures.

Moreover, the user and kernel level approaches have some common drawbacks.

• If the default contention manager used by the STM inserts exponential delays before restarting the aborted transaction, then it might prevent the repeat conflicts anyway. Thus, these techniques might not be that useful when they are incorporated with such an STM.
• Applications with mostly non-deterministic transactions might not benefit from these techniques because a restarted transaction, if it is non-deterministic, might not be in a repeat conflict. Serialization of such transactions after they have been in conflict once, might reduce concurrency and degrade performance.
• These techniques cannot be used when the conflicts are detected lazily. In order to have repeat conflicts, the conflicts must be detected eagerly when the two transactions are still running. When lazy detection is used, the conflicts can only be detected at commit time between an active transaction, that is about to commit, and a passive one, that has already committed.
• They all consider just one execution of an application that uses STM. They do not consider to improve the future executions based on the observations from previous executions of an application.

The approach I plan to propose is mostly focused on the last two issues, although it can be used to eliminate all of them. The information that is gained from several executions of an application can be used to make the future executions better. When an application observes a bad behavior or a failure in an execution, this failure can be fingerprinted in a local folder for that application and avoided in the future executions. This way, the application gains immunity against the bad behavior of the program. These types of techniques are called failure immunity techniques in general and showed to be successful for lock-based concurrent programming previously, like in [15].

The idea of failure immunity can be used in TM by fingerprinting the conflicts when they are detected in some executions, and recording them in a local folder in order to avoid these conflicts in the future runs of the same application by serializing the conflicting transactions. With this approach not only repeat conflicts but even the first occurrence of a conflict can be eliminated completely in an execution. However, there are several challenges.

• The conflicts should be fingerprinted in a way that each conflict is recorded uniquely.
• The technique should not slow down the application. Fingerprinting the detected conflicts and later avoiding them should incur modest overhead.
• Non-determinism should be taken into account.
• The idea should be as general purpose as possible in order to incorporate it easily with all the STMs.

B. Current Status - CAV-STM

Currently I implemented the approach in SwissTM [16] as CAV-STM, Conflict Avoidance in Software Transactional Memory.

When a conflict is detected, it is fingerprinted and recorded by CAV-STM. In SwissTM, transactions are lexical transactions, meaning that each transaction is given a unique lexical
ID. This is the unique lexical ID assigned by the compiler if a
compiler based STM is used. Therefore, a transaction’s ID
will not change unless the application is re-compiled. Most
of the real-world programs are already compiled versions
of the programs, and are being used without re-compiling
the application. Thus, it is reasonable to think that for a real-
world program that uses an STM library, a transaction’s ID
will be the same for all runs of an application. Based-on
this assumption, a conflict can be fingerprinted and recorded
by using the IDs of the transactions that are involved in the
conflict.

The way to get the IDs of the transactions changes for
write/write and read/write conflicts. It is easy to get the ID of
a transaction and record the conflict in the case of write/write
conflicts because in SwissTM, the ID of the conflicting trans-
action is known when a write/write conflict is detected. For
the read/write conflicts, it is sufficient to check the version of
the shared-data to see whether it is the same as the version
it was read by the transaction. Therefore, in SwissTM the
ID of the transaction that wrote the shared data and caused
the conflict is not known in a read/write conflict. To get the
conflicting transaction’s ID in this case, a table is maintained
which keeps the last transaction that wrote to a shared memory
location. It is not hard to maintain such a table because at
commit time the changed shared memory regions are known
to make the changes visible. The only thing to do is to update
the corresponding entry of the table at commit time when a
shared region is written to by a transaction.

For avoiding the conflicts, an array is maintained in order
to keep track of currently running transactions. In the array,
the ID of the currently executing transaction is kept for each
thread. Whenever a thread, \( T \), wants to start executing a
transaction, \( T' \), this array and the conflict history are checked
to see if any of the transactions that \( T \) previously conflicted
with is running or not. If they are then \( t \) is yielded, so that if
there are other threads who want to execute other transactions
that might not conflict with the currently active transactions
can run.

C. Future Work

The preliminary results showed that avoiding all the pre-
viously detected conflicts is not such a good idea because
transactions can be non-deterministic. Avoiding all the pre-
viously detected conflicts over-serializes the application and
hurts performance. Therefore, as the next step for CAV-STM
a frequency based approach is considered. The high level
view of this approach is to let the application run without
any conflict avoidance for a certain number of times. During
these runs the information on the frequency of two transactions
being in conflict will be collected for each pair of transactions.
If this frequency is more than some pre-defined threshold for
a pair of transactions, then in the following runs these two
transactions will be serialized. This way, over-serialization of
the non-deterministic transactions can be avoided.

In general, conflicts should be fingerprinted more precisely.
To do this, I can include more information while recording
a conflict; like some branching decisions, calling context,
or some information related to the shared data that is used by
the transactions that have been in conflict. This way
conflicts can be fingerprinted in a more specific way, and
the conflict avoidance component of CAV-STM would know
under what conditions two transactions should be serialized in
order to prevent their future conflicts. This would put more
overhead to CAV-STM components because there would be
more information to keep track of. However, since it allows
more parallelism by not serializing transactions unnecessarily,
it might give a better performance in the end.

D. Conclusion

CAV-STM is a scheduling based solution for conflict avoid-
ance in Software Transactional Memory. Its main aim is to
help an application that uses STM to improve its performance
in its future executions by learning from the good or bad
behaviors from its previous executions. Currently, it is focused
avoiding previously observed conflicts in the future executions
of an application. However, I aim to develop a more general-
purpose system.

REFERENCES

support for lock-free data structures,” SIGARCH, 1993.
for dynamic software transactional memory,” in PODC, 2005.
tional contention managers,” in PODC, 2005.
1995.
fusion avoidance and resolution for software transactional memory,”
in PODC, 2008.
Watson, “Steal-on-abort: Improving transactional memory perfor-
ance through dynamic transaction reordering,” in HiPEAC, 2009.
J. L. Lawall, and G. Müller, “Scheduling support for transactional
Scherer III, and M. L. Scott, “Lowering the overhead of software
transactional memory,” Computer Science Department, University of
implementing software transactional memory,” in Conf. on Object-
transactional memory for dynamic-sized data structures,” in PODC,
2003.
Stanford transactional applications for multi-processing,” in IISWC,
2008.
Jarvis, “Lee-tm: A non-trivial benchmark suite for transactional
ity: Enabling systems to defend against deadlocks,” in OSDI, 2008.