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

Shlomi Dolev[*]
dolev@cs.bgu.ac.il

Danny Hendler
hendlerd@cs.bgu.ac.il

Adi Suissa[†]
adisuis@cs.bgu.ac.il

Department of Computer Science
Ben-Gurion University of the Negev
Be'er Sheva, Israel

ABSTRACT

Transactional memory (TM) is a key concurrent programming abstraction. Several software-based transactional memory (STM) implementations have been developed in recent years. All STM implementations must guarantee transaction atomicity but different STM implementations may provide different progress guarantees. In order to ensure progress, an STM implementation must resolve transaction conflicts. This is done either by the implementation itself or by delegating conflict resolution to a separate *contention manager* module that tries to resolve transaction collisions once they are detected.

We present CAR-STM, a scheduling-based mechanism for STM Collision Avoidance and Resolution, that can be incorporated into existing STM implementations. CAR-STM maintains per-core transaction queues and schedules a thread while it is performing a transaction. CAR-STM employs the following two novel collision reduction techniques: (1) *seriailizing contention managers* resolve conflicts by aborting one transaction and moving it to the transactions queue of the other, effectively serializing the execution of these transactions and ensuring they will not collide again. (2) *Proactive collision reduction* allows applications to provide information about transactions' collision-probability. CAR-STM uses this information to pre-assign transactions that are more likely to collide to the same core.

We have incorporated CAR-STM into the University of Rochester's STM (RSTM) and compared the performance of the new implementation with that of the original RSTM by using STMBench7. Our results show that the new implementation provides orders-of-magnitude reduction of execution times and improved throughput for almost all concurrency levels. Additionally, since CAR-STM greatly reduces the unpredictable influence of operating-system scheduling on STM performance, the new implementation provides a much more *stable performance.* In contrast, the performance of the original RSTM implementation on STMBench7 workloads exhibits extremely high variance. Though our paper focuses on software transactional memory, we believe the ideas introduced by CAR-STM may prove useful also for hybrid implementations of transactional memory.

[*]Supported in part by Rita Altura Trust Chair in Computer Sciences.

[†]Supported in part by grants from Intel Corporation and the Lynne and William Frankel Center for Computer Sciences.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: [Concurrent Programming - Parallel Programming]

General Terms

Algorithms, Experimentation, Performance

Keywords

transactional memory, contention management, synchronization, collision avoidance and reduction, scheduling

1. INTRODUCTION

The emergence of multi-core architectures is adding impetus to the shift from single-threaded applications to concurrent, multi-threaded applications. Efficiently synchronizing accesses to shared memory is a key challenge posed by concurrent programming. Conventional techniques for inter-thread synchronization use lock-based mechanisms, such as mutex locks, condition variables and semaphores, and result in implementations that are either error-prone (if fine-grained locks are used) or might not scale (if coarse-grained locks are used).

Transactional memory (TM) [13, 23] is a concurrent programming abstraction that is viewed by many as having the potential of becoming a superior alternative to lock-based programming. Transactions allow a thread to execute a sequence of shared memory accesses whose effect is atomic: similarly to database transactions [24], A TM transaction either has no effect (if it fails) or appears to take effect instantaneously (if it succeeds).

Hardware transactional memory (HTM) was introduced by Herlihy and Moss [13]. They proposed the addition of a fully-associative transactional cache and the use of cache coherence protocols for guaranteeing transaction atomicity. A few additional HTM designs were proposed since then [1, 9, 17, 18, 19]. *Software transactional memory* (STM), introduced by Shavit and Touitou [23], ensures transactional semantics through software mechanisms. Many STM implementations have been proposed in recent years [5, 10, 11, 12, 15, 16, 21]. *Hybrid transactional memory* uses hardware whenever possible and falls back to software otherwise [4, 14]. Although this paper studies software transactional memory, we believe the ideas we introduce here may prove useful also for hybrid implementations.

STM implementations guarantee transaction atomicity. The method that is used for resolving transaction conflicts determines the progress guarantees. This is done either by the STM implementation itself or by delegating conflict resolution to a separate *contention manager* module [12] that tries to resolve transaction collisions once they are detected. When a transaction detects a conflict, it consults the contention manager in order to determine how to proceed. The contention manager can then decide which of the two conflicting transactions should continue and when and how the other transaction should be resumed.

We present CAR-STM, a scheduling-based mechanism for STM Collision Avoidance and Resolution, that can be incorporated into existing STM implementations. CAR-STM maintains per-core transaction queues and schedules a thread while it is performing a transaction. CAR-STM utilizes its scheduling capability in two key ways. First, it employs a novel type of contention management that we call *serializing contention management.* Rather than letting a pair of transactions collide over and over again, a serializing contention manager, upon detecting their first collision, aborts one transaction and moves it to the transactions queue of the other; this effectively serializes their execution and ensures they will not collide again. The experimental results we provide in Section 4 establish that this technique greatly improves STM performance.

A second technique introduced by CAR-STM is *proactive collision avoidance.* Rather than handle conflicts post factum (i.e., after the conflicting transactions have already started), proactive collision avoidance allows CAR-STM to pre-assign transactions that are more likely to collide to the same core. CAR-STM allows applications to provide information about transactions' collision-probability and uses this information for deciding where to put a new transaction. For some applications, this may eliminate even the first collision between transaction pairs. A similar approach was examined by Bai et al. [3]. They focus on *dictionary*-based data-structures and show that their implementation greatly improves transaction data locality. We generalize this idea by supporting an interface through which an application can provide its own unique collision-probability information.

We have incorporated CAR-STM into the University of Rochester's STM (RSTM) [16] and compared the performance of the new implementation with that of the original RSTM by using STMBench7 [8], a benchmark that generates realistic workloads for STM implementations. Our results show that the new implementation provides orders-of-magnitude speedup of execution times and improved throughput for almost all concurrency levels. Additionally, CAR-STM provides a much more *stable performance* as compared with the baseline RSTM, probably since it greatly reduces the unpredictable influence of operating-system scheduling on STM performance. In contrast, the performance of the original RSTM implementation on STMBench7 workloads exhibits extremely high variance. Since we found no efficient collision-probability function for the STMBench7-generated workloads, the dramatic performance gains are the result of serializing contention management.

In Section 4.5, we test CAR-STM on a synthetic application for which a natural collision-probability function exists. Although our proactive collision reduction feature increases throughput considerably, it is noteworthy that serializing contention management obtains an even greater performance boost also for this application. When both features are combined, CAR-STM obtains its maximum performance boost.

Very few prior art works study the issue of scheduling TM transactions. In a recent work, Yoo and Lee [25] incorporated a simple transaction scheduler into RSTM and LogTM. Their implementation allows the serialization of transactions to *a single scheduling queue* once high contention is detected. As they show, this approach can improve performance when the workload lacks parallelism. Rossbach et al. [20] introduce TXLinux, a variant of Linux that can manage HTM transactions in the Linux scheduler. They also propose a novel mechanism that allows critical sections to be protected by both locks and transactions.

The rest of the paper is organized as follows. We provide an overview of CAR-STM in Section 2. Section 3 gives a more detailed description of CAR-STM's architecture and operation. Performance analysis is provided in Section 4. Section 5 concludes with a short discussion of our results and future work.

2. CAR-STM OVERVIEW

In this section we provide a high-level overview of CAR-STM and the principles that underlie its design. This is followed by a more technical discussion in Section 3.

Figure 1-(a) presents the high-level architecture of CAR-STM. CAR-STM maintains a single *transactions queue* per every core in the system. Transactions are enqueued by the *transaction dispatcher*, described shortly. Once enqueued, the transaction's thread is put to sleep and is awakened by CAR-STM when the transaction completes. Each transactions queue is handled by a single *transactions queue thread* (TQ thread). Thus the number of TQ threads equals the number of cores in the system. CAR-STM guarantees that the transactions in the queue of each core are performed in order, one at a time, unless they are moved by a serializing contention manager (see Section 2.1).

2.1 Collision Resolution: Serializing Contention Managers

Prior-art contention managers [2, 6, 7, 22] have only a few alternatives for dealing with transaction conflicts. The contention manager decides which of the conflicting transactions can continue and whether the other transaction will be aborted or delayed. If a transaction is aborted or delayed, the contention manager also determines how long it must wait before it can restart or resume execution.

Scheduling-based STM implementations, on the other hand, have much greater control of which transactions are allowed to execute concurrently. This opens up new and effective ways in which contention managers can resolve transaction conflicts.

CAR-STM introduces a new type of contention managers that we call *serializing contention managers.* Serializing contention managers resolve conflicts by transferring one of the conflicting transactions to the transactions queue of the other.

We have implemented and evaluated two serializing contention managers. Upon identifying a conflict between two transactions, the *basic serializing contention manager* (BSCM) aborts the newer transaction T_n and moves it to the transaction queue of the older transaction T_o. We say that T_n *is serialized after* T_o.

BSCM greatly reduces the probability that two transactions will collide more than once but this is still possible. Consider a scenario, in which a conflict between T_a and T_b is identified and resolved by serializing T_b after T_a. If a conflict between T_a and a third transaction T_c is identified later on, then T_a might be serialized after T_c and T_a and T_b may collide once again.

The second contention manager we implemented, called the *permanent serializing contention manager* (PSCM), does guarantee that any pair of transactions can collide at most once. Assume a conflict between transactions T_a and T_b is identified and that these transactions have not been involved in any conflict before. Similarly to BSCM, PSCM resolves this conflict by aborting the newer transaction (assume this is T_b) and serializing it after T_a. In addition, however, T_b is marked as a *subordinate transaction* of T_a. If T_a is found to conflict with a third transaction T_c at a later stage of its execution and if T_c is older than T_a, then *both* T_a *and* T_b *are serialized after* T_c. More generally, whenever a transaction is moved to a different queue, it is moved together with all its subordinate transactions. It follows that once T_a is serialized after T_b, it can only be executed after T_a.

2.2 Collision Avoidance, Dispatching and Conflict-Probability Methods

Two transactions conflict if they both access the same object and at least one of them modifies it. Existing STM implementations detect transaction conflicts only after the pair of conflicting transactions start their execution. Once such a conflict is detected, one of the transactions has to be aborted or delayed and later restarted or resumed, possibly after a waiting period. Aborting a transaction may be a costly operation. It is also wasteful, since it causes all the prior changes done by the aborted transaction to be undone. If the aborted transaction is allowed to restart immediately, then it might be aborted once again by the same conflicting transaction or, even worse, abort the other transaction in a live-lock manner. On the other hand, if the aborted transaction has to wait before it may resume execution, then its completion time is delayed, possibly even beyond the point when its conflicting transactions terminated.

Rather than handle collisions only after the conflicting transactions have started to execute, CAR-STM tries to avoid collisions in the first place by pre-assigning transactions that are more likely to collide to the same core. When a thread wishes to start a new transaction, the transaction dispatcher is consulted. The dispatcher decides into which transactions queue the new transaction should be enqueued. Applications can influence the dispatcher's decisions by passing a pointer to a *conflict-probability method* as a parameter when calling *BEGIN TRANSACTION*. A conflict-probability method receives two transaction information (T-Info) structures as its input and computes an estimate of the probability that these transactions will collide if executed concurrently. Roughly speaking, the dispatcher

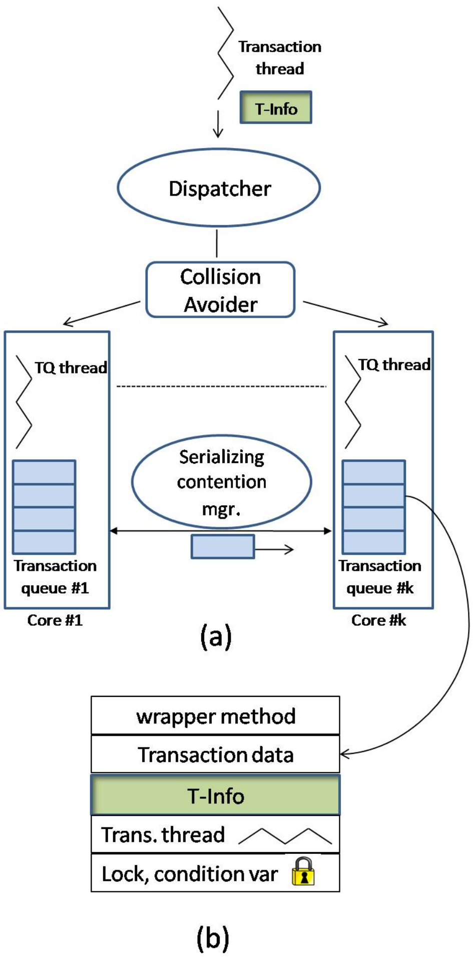

Figure 1: (a) CAR-STM high-level architecture. (b) The TQ-entry structure.

enqueues a transaction T to a queue that contains the transactions with maximum probability of colliding with T. By placing T in the same queue with these transactions, the scheduler decreases the probability that they will execute concurrently with T, hence also the probability that they will collide with T. The dispatcher is described in more detail in Section 3.2.

2.3 Eliminating Pseudo-Parallelism

Conventional (non-scheduling) STM implementations have very limited control of transaction threads or no control at all. Consequently, the performance of these implementations depends to a large extent on the operating system scheduler, whose policy does not take transactions into consideration. This is disadvantageous for several reasons.

1. As our experiments in Section 4 show and as observed by [8], the inherent nondeterminism of transaction-ignorant operating system scheduling results in extremely high variance of STM performance, especially for workloads dominated by long transactions.

2. Conventional STMs do not (and, in general, cannot) prevent the execution of multiple transaction threads

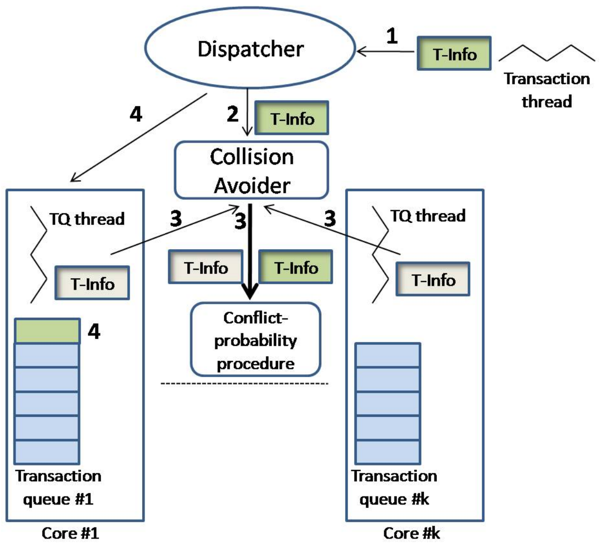

Figure 2: Transaction Dispatching

on the same core. We call the concurrent execution of several transaction threads on the same core *pseudo-parallelism*. Pseudo-parallelism makes little sense: since transactions are, in general, not allowed to perform I/O operations, same-core transaction threads are not very useful for overlapping communication and computation. On the other hand, and as observed by [25], pseudo-parallelism can greatly degrade performance. This is because same-core transactions may collide with each other, especially if transactions are long in comparison with the scheduler's computation quantum. A large number of concurrent threads assigned to the same core can also increase the frequency of cache misses and page faults. Pseudo-parallelism also increases the probability of collisions between transaction threads across cores, simply because a larger number of transaction threads execute concurrently.

3. In multi-programming systems, pseudo-parallelism is also likely to degrade the performance of other applications that execute in parallel, since cores must be shared by a larger number of threads.

By leveraging its transaction scheduling capability, CAR-STM eliminates pseudo-parallelism and guarantees that, at all times, at most a single transaction executes on any single core.

3. CAR-STM: ARCHITECTURE AND OPERATION

This section provides a more detailed description of CAR-STM. CAR-STM's structure allows it to be easily incorporated into existing STM implementations. We have incorporated CAR-STM into the RSTM implementation from the University of Rochester [16]. CAR-STM was mostly added as an external library and only a few minor "hooks" to existing RSTM code were required.

3.1 Initialization

Upon initialization, an application calls a method that initializes CAR-STM's data structures. In particular, a transactions queue is allocated and initialized per every system core (see Figure 1-(a)). A TQ thread is created per every core and its core-affinity is set accordingly. Every TQ thread has a single condition variable associated with it. Initially, each TQ thread sleeps on its condition variable waiting for the arrival of new transactions. Optionally, the application can pass a pointer to a *conflict-probability method* upon initialization. This application-specific method is used by the dispatcher to reduce the number of collisions. See Section 3.3 for a more detailed description of the dispatcher.

In addition to application-level initialization, each application thread must also call an RSTM thread-initialization method in order to be able to execute transactions. Upon initializing, the thread notifies RSTM which transaction manager it will use, whether transactional reads should be visible or not, and whether its validation policy is eager or lazy.

3.2 Transaction Dispatching

Figure 2 depicts the stages of the transaction dispatching process.

1. A transaction thread calls the dispatcher and passes a pointer to its transaction information structure (T-Info) as an argument. Observe that multiple threads, running on multiple cores, may call the dispatcher concurrently.

2. The dispatcher calls the collision avoider, passing the T-Info structure of the new transaction as an argument.

3. The collision avoider calls the application-specific conflict-probability (CP) method, provided upon initialization. This method receives two T-Info arguments and computes an estimate of the probability that the two transactions conflict. In general, the collision avoider may call the CP method multiple times per every TQ, in order to check possible conflicts with a few TQ transactions. In our implementation, the CP method is called at most once per TQ and checks possible conflicts with the currently active TQ transaction (if any).

4. Based on the collision avoider's output, the dispatcher decides where to enqueue the new transaction. It then creates a new TQ-entry structure and enqueues a pointer to it to the selected TQ. The atomicity of the insertion is ensured by using the TQ's lock and condition variable. The structure of a TQ entry is shown in Figure 1-(b). Each entry contains a pointer to a structure storing the data required by the transaction method and a wrapper method that calls the transaction method with the required arguments (see Section 3.3 for more details on transaction execution). A third field stores the transaction's T-Info structure. This structure is used by the collision avoider to compute the probability of collision with new transactions. Additional two fields contains the transaction's thread data required for resuming it after its transaction is completed, and a lock/condition-variable used for synchronizing with it. After the TQ-entry is added, the dispatcher signals the appropriate TQ-thread to notify it of available work.

The transaction thread is then put to sleep until a TQ-thread notifies it of transaction completion.

3.3 Transaction Execution

Transactions are executed by TQ-threads. A TQ-thread sleeps on the transaction queue's condition variable until the dispatcher wakes it up. It then starts executing the transaction at the head of the queue.

An RSTM transaction is executed by its transaction thread and consists of a sequence of instructions surrounded by a *BEGIN TRANSACTION / END TRANSACTION* macros-pair. As mentioned already, a CAR-STM transaction is performed by a TQ-thread and not by its transaction thread. This is made possible by packaging each CAR-STM transaction as a *transaction method.* Moreover, since transactions are not executed in the context of their original thread, a mechanism is required for allowing them to access the data they require. To this end, a transaction thread calls the dispatcher with two more operands in addition to the T-Info structure. The first is an application-specific structure storing the transaction data (if any). The second is a reference to a wrapper method. When executing a transaction, the TQ thread calls the transaction's wrapper method. The wrapper method, in turn, calls the transaction method and passes to it a reference to its transaction-data structure.

After the transaction completes, the TQ-thread signals the corresponding transaction-thread, dequeues the transaction entry and proceeds to execute additional transactions. When the queue becomes empty, the TQ-thread resumes sleeping on its condition variable.

3.4 Serializing Contention Management

RSTM's conflict resolution is implemented by the contention manager's *ShouldAbort* method. This method receives the contention manager of the "enemy transaction" and returns a boolean value indicating whether or not the enemy transaction should be aborted. If the enemy transaction is aborted, the transaction continues until either another conflict is identified or the transaction commits. If the enemy transaction is not aborted, the transaction retries, possibly after some waiting period.

As mentioned in section 2, CAR-STM incorporates two serializing contention managers: the *basic serializing contention manager* (BSCM) and the *permanent serializing contention manager* (PSCM). Upon initialization, each transaction receives a time-stamp. When a conflict is detected, the time-stamps of the conflicting transactions are compared and the newer transaction is aborted. The serializing contention manager then moves the TQ-entry of the losing transaction (actually a pointer to it) to the transactions queue of the enemy transaction. This decreases (with BSCM) or completely eliminates (with PSCM) the probability that the two transactions will collide again.

BSCM moves the losing transaction to the end of the transactions queue of the enemy transaction. With PSCM, a transaction's TQ entry points to a list of subordinate transactions that are serialized after it. When a transaction T is aborted, it is moved to the end of the subordinate transactions list of T's enemy transaction; if T itself has subordinate transactions they are moved with it. When a transaction terminates, the TQ thread will first execute the transactions in its subordinate transactions list (if any) before proceeding to the next TQ entry. Figure 3 illustrates the operation of PSCM.

4. EXPERIMENTAL EVALUATION

In this section we provide the results of our experimental evaluation. We have tested CAR-STM by incorporating it into RSTM. Our first set of tests evaluates the new implementations on STMBench7 workloads. Since we found no efficient conflict-probability function for STMBench7 workloads, these tests only evaluate our serializing contention managers.

4.1 STMBench7: a Short Overview

STMBench7, due to Guerraoui et al. [8], is an STM benchmark that can create realistic transactional workloads corresponding to a wide variety of applications, such as CAD, CAM and CASE applications. STMBench7 was designed with the goal of comparing different STM implementations based on their performance on a wide range of workloads and concurrency patterns.

STMBench7's data structure consists of several types of objects and is organized in a tree-like structure. The root of the data structure is called the *module object.* Internal tree nodes consist of *complex assembly objects*, each having 3 children, and the leaves are called *base assembly objects.* Each base assembly object has several *composite part* objects that might be shared by several base assembly objects. A composite part consists of a *document* and several *atomic parts* which are connected by *connection* objects. In addition, there is a *manual* object connected to the module object. Every object has a unique ID and there is also an index for each object type, that enables direct access to the object. See [8] for a more detailed description.

STMBench7 supports 45 operation types. Some are read-only operations, while others also update values and/or change the hierarchical structure. The operations are categorized as follows.

1. *Long traversals* - Go through all assemblies and/or all atomic parts and may update some of them. These are long operations and have the highest probability of conflicting with other operations.

2. *Short traversals* - These traversals start from a module, a document, or an atomic part and choose a random path. The operation's path is chosen after it starts. Consequently, there is no way of knowing in advance whether two short traversal operations will conflict. Moreover, if operations A and B conflict and A is restarted, it might choose a different path that will not conflict with B.

3. *Short operations* - These operations choose an object and perform some operations on the object and its local neighborhood. Clearly, two short operations that work in the same neighborhood are more likely to conflict.

4. *Structure modifications* - These operations create or delete data-structure objects. It it hard to estimate in advance whether two such operations will conflict, since they operate on randomly chosen objects.

An STMBench7 workload is determined by choosing a set of enabled operations, specifying the number of threads and

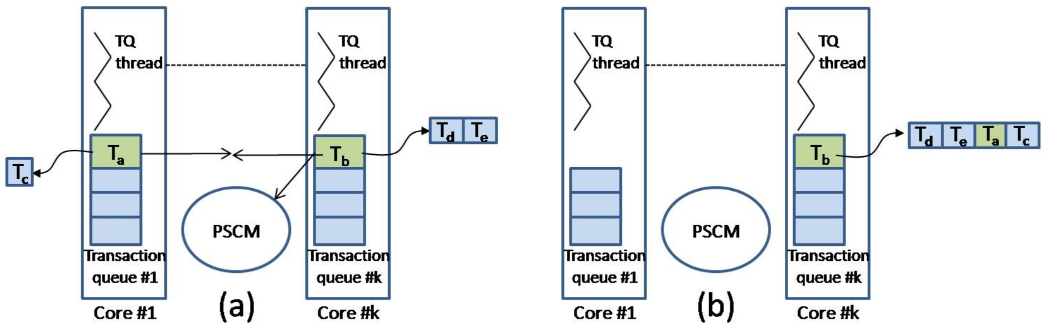

Figure 3: PSCM operation. (a) Transactions T_a and T_b collide. (b) PSCM moves transaction T_a along with its list of subordinate transactions to T_b's list of subordinate transactions.

Table 1: STMBench7 ratios of operations per workload type

Category	Write-Dom.	Read-Write	Read-Dom.
Read ops.	10	60	90
Write ops.	90	40	10

selecting a workload type (read-dominated, write-dominated, or read/write-dominated).

4.2 Adapting STMBench7 for Using CAR-STM

The following simple change to STMBench7 was required to allow CAR-STM usage. Each STMBench7 transaction was packaged as a single transaction-method. As mentioned before, this is required for separating between transactions and the threads that initiate them so that transactions can be executed by TQ-threads and moved between them. This change was straightforward: transactional code was simply packaged as a method and the dispatcher was invoked with a pointer to the transaction method as an argument.

4.3 Evaluation Methodology

We have tested CAR-STM on a 2.60 GHz 8 core 4xXEON-7110M server, with 16BG RAM and 4MB L2 cache and with HyperThreading disabled, running the 64-bit Gentoo Linux operating system over the 2.6.22 kernel. The baseline for our evaluation is the C++ version of STMBench7 using RSTM.

When running the baseline RSTM, we used *Polka*, the default contention manager for STMBench7, which was found to give top or near-top performance in most RSTM benchmarks [22]. The default validation approach is visible readers with eager acquire (*vis-eager*), providing early detection of all conflicts. We have tested CAR-STM with the basic serializing contention manager against the 3 types of workloads available in STMBench7 (see Table 1), and evaluated the permanent contention manager under the read-write workload.

In what follows, we let RSTM refer to the baseline RSTM implementation. We compare RSTM with 2 new implementations that integrate it with CAR-STM:

• RSTM/CAR-B - integrates RSTM and CAR-STM using the BSCM contention manager.

• RSTM/CAR-P - integrates RSTM and CAR-STM using the PSCM contention manager.

In our tests, we evaluated the three implementations according to the following metrics.

- *throughput* - the number of transactions per second. Throughput is measured for workloads that do not include long transactions and last 5 minutes. Threads generate new transactions all throughout the test.
- *execution time* - test duration in seconds. Execution time is measured for workloads that include long transactions. The test is initially run for 5 minutes while threads generate new transactions. After 5 minutes, threads stop generating new transactions and the test continues until all transactions terminate. We call this later period the test's *quiescence time.* Long quiescence times are an indication that transactions continually collide with each other in a live-lock manner.
- *stability* - measured by the standard deviation of the above metrics. A large standard deviation implies that many tests of a certain workload and concurrency level are far from the averaged performance. This measures the extent to which an implementation provides stable performance.

Each implementation was evaluated according to its average throughput, execution and quiescence times, where the average is computed for 10-test batches. We also evaluated the implementation's *throughput stability* and *execution time stability.* That is, we computed the standard deviation of implementations' throughput, execution- and quiescence-times.

4.4 STMBench7 Workloads Results

We have evaluated the throughput, execution and quiescence times obtained by RSTM, RSTM/CAR-B and RSTM/CAR-P for STMBench7 read/write-, read- and write-dominated workloads. Figure 4 presents the results for read/write-dominated workloads. Both RSTM/CAR-B and RSTM/CAR-P outperform RSTM for all metrics and at all concurrency levels and the gap increases with the concurrency level. RSTM/CAR-B is the clear winner in terms of throughput for all concurrency levels and achieves a speedup of almost 4 in 16-thread tests and 15.7 for 32-threads as compared with RSTM. While RSTM's throughput deteriorates quickly as concurrency level increases, RSTM/CAR-B approaches its peak performance with 8 threads and maintains it as the concurrency level is increased to 32. RSTM/CAR-B's higher throughput as compared with RSTM/CAR-P is

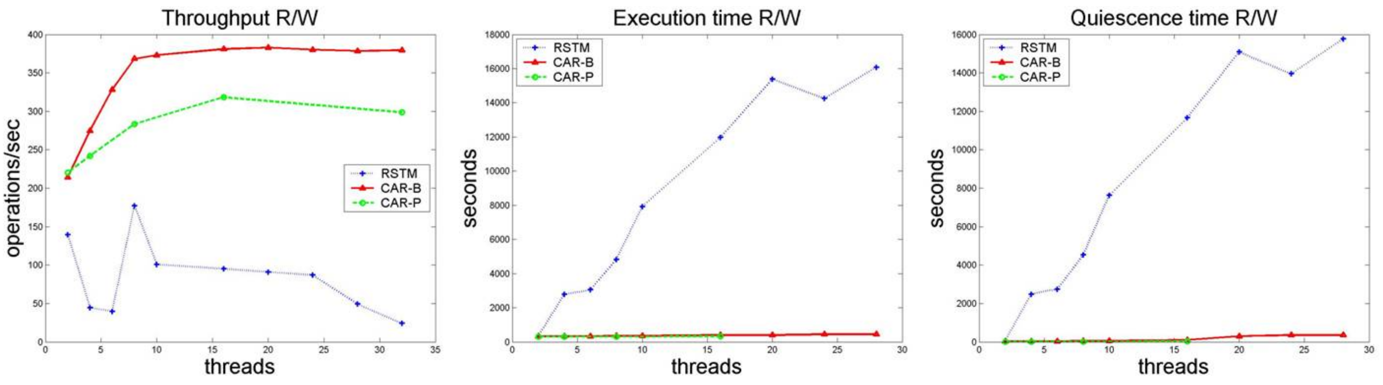

Figure 4: Throughput, execution and quiescence times under the Read-Write workload

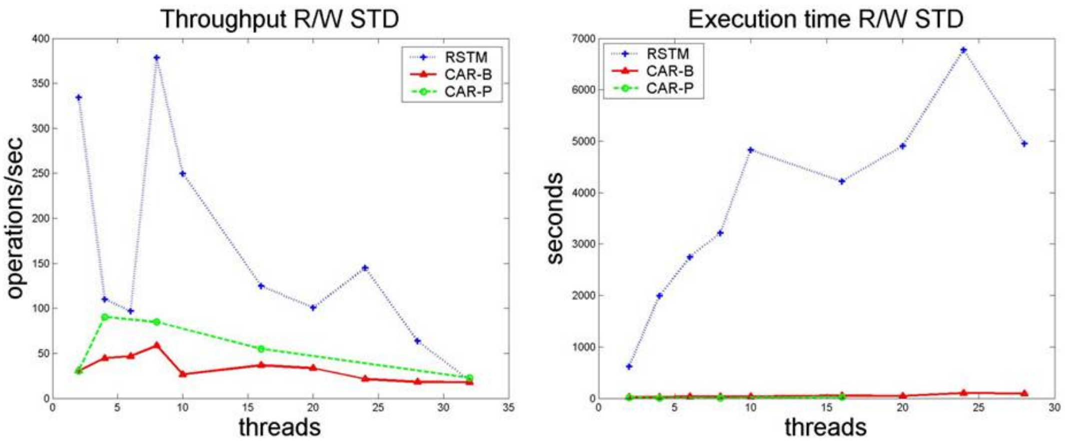

Figure 5: Throughput and execution time standard deviation under the Read-Write workload

probably due to the higher overhead of permanent serialization.

Even more impressing is the dramatic reduction in execution and quiescence times obtained by both RSTM/CAR-B and RSTM/CAR-P as compared with the baseline RSTM. Both variants achieve a speed-up in execution of between 1.7 (for 2 threads) and 36 (for 28 threads) in comparison with RSTM. Measuring execution time actually undermines the improvement obtained by CAR-STM, since no scheme can reduce test duration bellow the initial 5-minute period. When measuring the more meaningful quiescence times, both RSTM/CAR-B and RSTM/CAR-P achieve a speed-up factor of between 11 (for 2 threads) and 118 (for 16 threads). The huge quiescence time of RSTM is a clear indication of live-locked behavior, in which a set of transactions continually abort one another. Our experimental results establish that CAR-STM managed to serialize these transactions very quickly and avoid this live-lock behavior.

Figure 5 presents the standard deviations of the throughput and execution times[1]. The performance of the baseline RSTM is extremely unpredictable and its execution times are highly variate. Both variants of CAR-STM, on the other hand, provide execution times that are orders of magnitude more stable. For example, RSTM/CAR-B's execution time standard deviations for 2, 4, 8 and 16 threads are 21, 20, 30 and 52, respectively, as compared with 614, 1993, 3209 and 4220 (!) for the baseline RSTM. The problematic behavior of RSTM with these workloads can be explained by the high probability of live-lock between long transactions and the unpredictable influence of the OS scheduler on whether live-lock actually occurs. CAR-STM does not suffer from this problem because the serializing contention manager either greatly decreases live-lock probability (CAR-B) or eliminates it completely (CAR-P).

RSTM/CAR-B's throughput standard deviation is also significantly more stable as compared with the baseline RSTM, but "only" by a factor of between 2.5 (4 threads) and 7.5 (8 threads). RSTM/CAR-P's throughput standard deviation is better than RSTM in all concurrency levels but is significantly worse than that of RSTM/CAR-B.

Figure 6 presents the performance results for the write-dominated workload. The performance boost obtained by serializing management is even more significant here since write-dominated workloads have higher collision probability. Although the baseline RSTM's throughput initially improves as the number of threads increases, it decreases drastically once the number of threads exceeds the number of

[1]The quiescence time standard deviation graph is very similar to that of the execution time and was omitted.

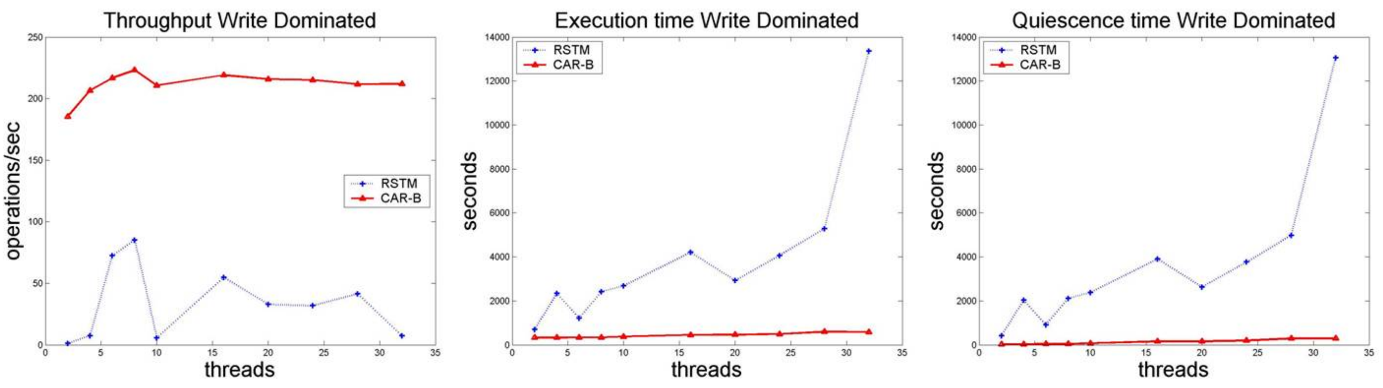

Figure 6: Throughput, execution and quiescence times under the Write-dominated workload

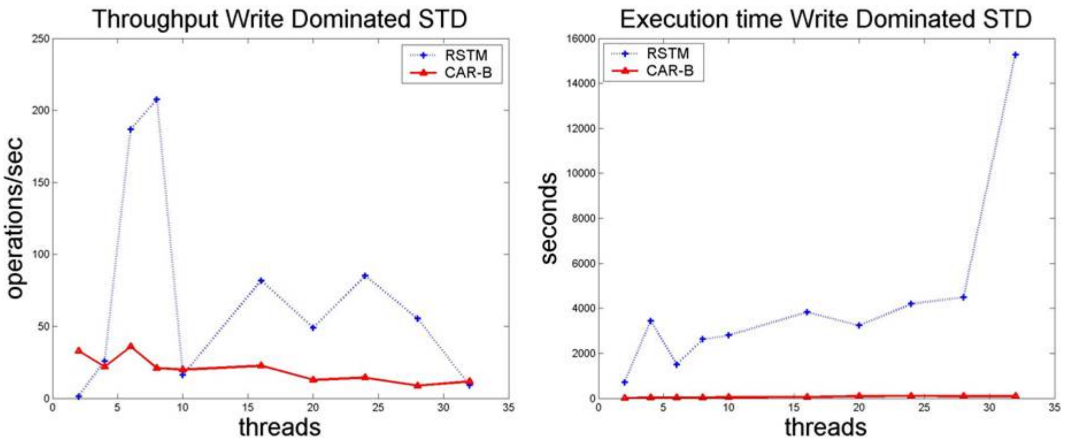

Figure 7: Throughput and execution time standard deviation under the Write-dominated workload

cores. RSTM/B achieves a 4-fold throughput increase for 16 threads and a factor of almost 30 for 32 as compared with RSTM.

The execution and quiescence times of the baseline RSTM implementation are very high, and vary from a factor of 2 (for 2 threads) to a factor of 23 (for 32 threads) when compared to the RSTM/CAR-B. It is worth mentioning that, even for high concurrency levels and in spite of the high conflict probability of write-dominated workloads, the execution times of RSTM/CAR-B and their corresponding standard deviations are low.

The standard deviation of the baseline RSTM in this workload type is still high but is significantly lower than in the read-write workload. This is because here RSTM behaves in a live-locked manner in almost all tests. While the standard deviation of RSTM/CAR-B is at most 106 (for 28 threads), RSTM's varies between 711 (for 2 threads) and 15274 (for 32 threads). It is noteworthy that when the throughput of RSTM is relatively high (85 for 8 threads) so is its standard deviation (207).

Read-dominated workloads include only 10% write operations, hence conflict-probability is significantly smaller. Here, the baseline RSTM throughput is better than RSTM/CAR-B by a factor of 2 for 2 threads due to RSTM/CAR-B's overhead. However, as the concurrency level increases, RSTM's (both relative and absolute) throughput decreases and with 32 threads the throughput of RSTM/CAR-B is higher by a factor of 2.5.

The execution and quiescence times of the baseline RSTM are still high but are lower as compared with read/write- and write-dominated workloads. RSTM/CAR-B accelerates execution time by a factor of between 1.7 (for 2 threads) and 19.2 (for 32 threads). The standard deviation of the baseline RSTM is highest with read-dominated workloads. Also here, as with read/write- and write-dominated workloads, RSTM/CAR-B dramatically improves execution stability. The graphs for the read-dominated workloads are not shown for lack of space.

4.5 Proactive Collision Avoidance Evaluation

We tested CAR-STM's proactive collision avoidance feature using a synthetic application which we name a *RegionedArray* (RA). An RA is an array composed of separate regions. Our tests use an RA of size 1000, partitioned to 8 regions, each containing 125 entries. Each RA entry stores a transactional object. Three operations are supported by the RA: read, write and delete (which initializes an entry to a default value). Each thread executes for 20 seconds, during which the following sequence of operations is repeated. 1) the thread randomly and uniformly selects a region on which to operate; 2) an integer *iter* is chosen uniformly and randomly from the range $[1, \ldots, 125]$; 3) an RA operation to apply is chosen uniformly at random, and 4) a transaction is started and performs *iter* iterations, in each of which it randomly and uniformly chooses an item within the region and applies the RA operation to it.

It is easily seen that the following constitutes a natural conflict-probability function for this application: return 1 if a pair of transactions operate on the same region or 0 otherwise. When a new transaction is dispatched, this function is invoked to compare it with the currently running transaction on each core. If there is a core in which the currently active transaction operates on the same region as the new transaction, it is put in that cores' queue; otherwise it is put in an arbitrary queue.

Figure 8 shows the results of the RA tests. We compared the throughput of (1) the baseline RSTM, (2) RSTM with proactive collision avoidance enabled (PROACTIVE), (3) RSTM/BSCM, and (4) RSTM/BSCM with proactive collision avoidance (PRO-BSCM). The Polka contention manager was used for the RSTM and PROACTIVE tests. It can be seen that for 2 threads, PROACTIVE increases throughput by a factor of more than 2.5 as compared with RSTM. As the concurrency level increases, however, acceleration decreases. The reason for this is that, at high concurrency levels, there's higher probability that the dispatcher code is called concurrently to assign multiple transactions. In such a scenario, transactions that operate on different regions may be (concurrently) assigned to the same core. Since new transactions are only compared with the currently active transaction, this increases the probability of future collisions.

It is noteworthy that even for the RA application, where a natural conflict-probability function exists, BSCM reduces collisions more efficiently than PROACTIVE and the gap between the two increases with the concurrency level. While PROACTIVE and BSCM perform almost the same for 2 threads, for 16 threads BSCM accelerates throughput by a factor of more than 2 as compared with PROACTIVE and by more than 3.5 as compared with RSTM. Finally, PRO-BSCM performs best at all concurrency levels. The improvement as compared with BSCM is small, though: it is negligible for 12 threads and reaches its peak - approximately 21% - for 16 threads.

5. DISCUSSION

We presented CAR-STM, a scheduling-based mechanism for STM collision avoidance and resolution. CAR-STM incorporates novel and highly efficient contention managers that resolve conflicts by serializing the execution of colliding transactions. These contention managers either greatly reduce (with BSCM) or completely eliminate (with PSCM) the probability that a pair of colliding transactions will collide again. CAR-STM also supports proactive collision reduction by pre-assigning transactions that are more likely to conflict to the same core. This is made possible by allowing an application to provide a conflict-probability method which computes an estimate of the probability that two transactions will collide. This method is used by CAR-STM to determine to which core a new transaction should be assigned.

We incorporated CAR-STM within RSTM and compared the new implementation with the original RSTM on transaction workloads generated by STMBench7. Our results

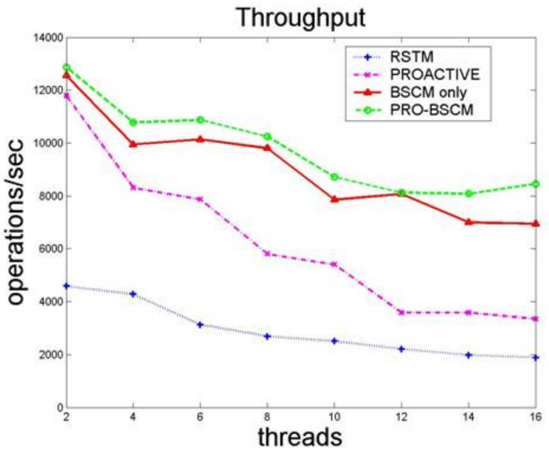

Figure 8: RegionedArray throughput

show that CAR-STM achieves dramatic reduction in execution time and significant throughput increase while, at the same time, providing orders-of-magnitude more stable performance as compared with RSTM. Since we did not find a natural conflict-probability function for STMBench7 workloads, this improvement is entirely the result of serializing contention management.

We have tested the effect of proactive collision reduction on a synthetic application for which a natural conflict-probability function exists and obtained significant throughput increase. Nevertheless, even for this application, the improvement obtained by serializing contention management was more significant. Combining proactive collision reduction and serializing contention management yielded the highest throughput for this application.

Although the quality of STM implementations is constantly improving, they have yet to demonstrate that they can provide performance comparable to that of lock-based applications across the concurrency range. We have shown that transaction-aware scheduling and serializing contention management can significantly speed-up STM execution and believe that they hold the potential of bringing further performance gains.

While our results show that proactive collision reduction can improve performance for some application, this work seems to indicate that serializing contention management is a more general and powerful collision reduction technique.

This work can be extended in several ways. First, more sophisticated serializing contention managers may permit more parallelism, while still greatly reducing the probability of repeated collisions. Second, our scheme incurs some overhead, which may be significant for low concurrency levels. An adaptive algorithm that turns collision reduction features on or off dynamically may improve performance for such workloads. Third, our current implementation employs non-preemptive scheduling: unless a transaction is aborted, a TQ thread must finish its execution before it can proceed to execute other transactions. Possibly a more sophisticated scheduling algorithm can resolve this robustness issue while still keeping pseudo-parallelism at a low level. Finally, this work suggests that making the operating-system scheduler "transaction-aware" may yield significant performance gains.

We leave these for future work.

6. REFERENCES

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327, 2005.

[2] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC*, pages 308–315, 2006.

[3] T. Bai, X. Shen, C. Zhang, W. N. Scherer III, C. Ding, and M. L. Scott. A key-based adaptive transactional memory executor. In *IPDPS*, pages 1–8, 2007.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, pages 336–346, 2006.

[5] K. Fraser. Practical lock-freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, 2004.

[6] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *DISC*, pages 303–323, 2005.

[7] R. Guerraoui, M. Herlihy, and B. Pochon. Towards a theory of transactional contention managers. In *PODC*, pages 316–317, 2006.

[8] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[9] L. Hammond, M. C. V.Wong, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabju, H. Wijaya, C. Kozyrakis, , and K. Olukotun. Transactional memory coherence and consistency. In *Proc. of the 31st Annual Intl. Symp. on Computer Architecture*, pages 102–113, 2004.

[10] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, 2003.

[11] M. Herlihy. SXM software transactional memory package for C#. http://www.cs.brown.edu/ mph.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[14] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPOPP*, pages 209–220, 2006.

[15] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *DISC*, pages 354–368, 2005.

[16] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, I. W. N. Scherer, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT06),*, 2006.

[17] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Conference on High Performance Computer Architecture*, pages 254–265, 2006.

[18] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[19] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, 2005.

[20] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *SOSP '07*, pages 87–102, New York, NY, USA, 2007. ACM.

[21] 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 *PPOPP*, pages 187–197, 2006.

[22] M. L. Scott and W. N. Scherer III. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248, 2005.

[23] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[24] G. Vossen and G. Weikum. Transactional information systems, , morgan kaufmann, 2001.

[25] R. M. Yoo and H. S. Lee. Adaptive transaction scheduling for transactional memory systems. Georgia Institute of Technology Technical Report GIT-CERCS-07-04, 2007.