On the Impact of Serializing Contention Management on STM Performance

Tomer Heber*, Danny Hendler and Adi Suissa**

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

Abstract. Transactional memory (TM) is an emerging concurrent programming abstraction. Numerous software-based transactional memory (STM) implementations have been developed in recent years. STM implementations must guarantee transaction atomicity and isolation. In order to ensure progress, an STM implementation must resolve transaction collisions by consulting a *contention manager* (CM). Recent work established that *serializing contention management* - a technique in which the execution of colliding transactions is serialized for eliminating repeat-collisions - can dramatically improve STM performance in high-contention workloads. In low-contention and highly-parallel workloads, however, excessive serialization of memory transactions may limit concurrency too much and hurt performance. It is therefore important to better understand how the impact of serialization on STM performance varies as a function of workload characteristics.

We investigate how serializing CM influences the performance of STM systems. Specifically, we study serialization's influence on STM *throughput* (number of committed transactions per time unit) and *efficiency* (ratio between the extent of "useful" work done by the STM and work "wasted" by aborts) as the workload's level of contention varies. Towards this goal, we implement CBench - a synthetic benchmark that generates workloads in which transactions have (parameter) pre-determined length and probability of being aborted in the lack of contention reduction mechanisms. CBench facilitates evaluating the efficiency of contention management algorithms across the full spectrum of contention levels.

The characteristics of TM workloads generated by real applications may vary over time. To achieve good performance, CM algorithms need to monitor these characteristics and change their behavior accordingly. We implement adaptive algorithms that control the activation of serialization CM according to measured contention level, based on a novel low-overhead serialization algorithm. We then evaluate our new algorithms on CBench-generated workloads and on additional well-known STM benchmark applications. We believe our results shed light on the manner in which serializing CM should be used by STM systems.

* Supported by grants from Intel Corporation and the Lynne and William Frankel Center for Computer Sciences.

** Supported by the *Israel Science Foundation* (grant number 1344/06).

1 Introduction

The advent of multi-core architectures is accelerating 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. An implementation that uses coarse-grained locks will not scale. On the other hand, implementations that use fine-grained locks are susceptible to problems such as deadlock, priority inversion, and convoying. This makes the task of developing scalable lock-based concurrent code difficult and error-prone.

Transactional memory (TM) [14, 20] is a concurrent programming abstraction that is viewed by many as having the potential of becoming a viable alternative to lock-based programming. Memory transactions allow a thread to execute a sequence of shared memory accesses whose effect is atomic: similarly to database transactions [22], a TM transaction either has no effect (if it fails) or appears to take effect instantaneously (if it succeeds).

Unlike lock-based programming, transactional memory is an optimistic synchronization mechanism: rather than serializing the execution of critical regions, multiple memory transactions are allowed to run concurrently and can commit successfully if they access disjoint data. Transactional memory implementations provide hardware and/or software support for dynamically detecting conflicting accesses by concurrent transactions and for resolving these conflicts (a.k.a. collisions) so that atomicity and progress are ensured. *Software transactional memory* (STM), introduced by Shavit and Touitou [20], ensures transactional semantics through software mechanisms; many STM implementations have been proposed in recent years [6, 7, 11–13, 15, 16, 18].

TM implementations typically delegate the task of conflict resolution to a separate *contention manager* (CM) module [13]. The CM tries to resolve transaction conflicts once they are detected. When a transaction detects a conflict with another transaction, it consults the CM in order to determine how to proceed. The CM can then decide which of the two conflicting transactions should continue, and when and how the other transaction should be resumed.

Up until recently, TM implementations had no control of transaction threads, which remained under the supervision of the system's transaction-ignorant scheduler. Consequently, the contention managers of these "conventional" (i.e., non-scheduling) implementation [2, 8, 9, 19] have only a few alternatives for dealing with transaction conflicts. A conventional CM can only decide which of the conflicting transactions can continue (this is the *winner transaction*) and whether the other transaction (the *loser transaction*) will be aborted or delayed. A conventional CM can also determine how long a loser transaction must wait before it can restart or resume execution.

1.1 Transaction Scheduling and Serializing Contention Management

A few recent works introduced transaction scheduling for increasing the efficiency of contention management. Yoo and Lee [23] introduced *ATS* – a simple user-level transaction scheduler, and incorporated it into RSTM [16] – a TM implementation from the University of Rochester – and into LogTM [17], a simulation of a hardware-based TM system. To the best of our knowledge, ATS is the first adaptive scheduling-based CM algorithm.

ATS uses a *local* (per thread) adaptive mechanism to monitor the level of contention (which they call *contention intensity*). When the level of contention exceeds a parameter threshold, transactions are being serializied to a single scheduling queue. As they show, this approach can improve performance when workloads lack parallelism. Ansari et al. [1] proposed *steal-on-abort*, a transaction scheduler that avoids wasted work by allowing transactions to "steal" conflicting transactions so that they execute serially.

Dolev et al. [3] introduced CAR-STM, a user-level scheduler for collision avoidance and resolution in STM implementations. CAR-STM maintains per-core transaction queues. Whenever a thread starts a transaction (we say that the thread becomes *transactional*), CAR-STM assumes control of the transactional thread instead of the system scheduler. Upon detecting a collision between two concurrently executing transactions, CAR-STM 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.

We emphasize that serializing contention management cannot be supported by conventional contention managers, since, with these implementations, transactional threads are under the control of a transaction-ignorant system scheduler. Conventional contention managers only have control over the length of the waiting period imposed on the losing transaction before it is allowed to resume execution. In general, waiting-based contention management is less efficient than serializing contention management. To exemplify this point, consider a collision between transactions T_1 and T_2. Assume that the CM decides that T_1 is the winner and so T_2 must wait.

- If T_2 is allowed to resume execution too soon, it is likely to collide with T_1 again. In this case, either T_2 has to resume waiting (typically for a longer period of time), or, alternatively, the CM may now decide that T_2 wins and so T_1 must wait. In the latter case, T_1 and T_2 may end up repeatedly failing each other in a livelock manner without making any progress.
- On the other hand, if the waiting period of T_2 is too long, then T_2 may be unnecessarily delayed beyond the point when T_1 terminates.

Contrary to waiting-based contention management, with serializing contention management the system is capable of resuming the execution of T_2 immediately after T_1 terminates, resulting in better performance.

Dolev et al. incorporated CAR-STM into RSTM [16] and compared the performance of the new implementation with that of the original RSTM implementation, by using STMBench7 [10], a benchmark that generates realistic workloads for STM implementations. Their results show that serializing contention management can provide orders-of-magnitude speedup of execution times for high-contention workloads. However, excessive serialization of memory transactions may limit concurrency too much and hurt the performance of low-contention, highly-parallel workloads. It is therefore important to better understand how the impact of serialization on STM performance varies as a function of workload characteristics and how CM algorithms can adapt to these variations.

1.2 Our Contributions

The performance of an STM system can be characterized in terms of both its *throughput* (number of committed transactions per time unit) and *efficiency* (ratio between the extent

of "useful" work done by the STM and work "wasted" by aborts) as the workload's level of contention changes. We approximate the efficiency of an STM execution by using the formula $commits/(aborts + commits)$, where *commits* and *aborts* respectively denote the numbers of committed and aborted transactions in the course of the execution.

We investigate how serializing CM influences STM throughput and efficiency. Our contributions towards obtaining this goal are the following.

- The characteristics of TM workloads generated by real applications may vary over time. To achieve good performance, CM algorithms need to monitor these characteristics and change their behavior accordingly. We implement and evaluate several novel adaptive algorithms that control the activation of serialization CM according to measured contention level. Both local-adaptive (in which each thread adapts its behavior independently of other threads) and global-adaptive policies are considered. Our adaptive algorithms are based on a novel low-overhead serialization mechanism. We evaluate these algorithms on workloads generated by CBench (described shortly), by an RSTM micro-benchmark application, and by Swarm - a more realistic application that is also provided by RSTM's benchmark suite.
- We introduce *CBench* - a synthetic benchmark that generates workloads in which transactions have pre-determined length and probability of being aborted (both provided as CBench parameters) when no contention reduction mechanisms are employed. CBench facilitates evaluating the effectiveness of both conventional and serializing CM algorithms across the full spectrum of contention levels.

Our results establish that applying serializing CM adaptively can improve STM performance considerably for high-contention workloads, while incurring no visible overhead for low-contention, highly-parallel workloads. Our empirical evaluation also highlights the importance of *stabilized* adaptive mechanisms, used to prevent frequent oscillations between serializing and conventional modes of operation that hurt performance.

The rest of the paper is organized as follows. We describe our new algorithms in Section 2. The CBench benchmark is presented in Section 3, followed by a description of our experimental evaluation in Section 4. Section 5 concludes with a short discussion of our results.

2 Algorithms

Table 1 summarizes the novel adaptive algorithms that we introduce and evaluate. These algorithms can be partitioned to the following two categories.

- *Partially-adaptive*: these are algorithms in which the contention-manager uses a conventional CM algorithm (we use the well-known Polka algorithm [19]) until a transaction collides for the k'th time, for some predetermined parameter k. Starting from the k'th collision (if the transaction fails to commit before that), a partially-adaptive CM starts using a serialization-based CM algorithm (which we describe shortly) until the transaction commits. We call k the *degree* of the algorithm and let PA_k denote the partially adaptive algorithm of degree k.

Table 1. Adaptive and partially-adaptive algorithms

PA_k	Partially adaptive algorithm of degree k: serialize starting from the k'th collision
A_L	Fully adaptive algorithm, local (per-thread) mode control
A_G	Fully adaptive algorithm, global (system-wide) mode control
A_{LS}	Fully adaptive algorithm, local (per-thread) stabilized mode control
A_{GS}	Fully adaptive algorithm, global (system-wide) stabilized mode control

– *Fully-adaptive*: these algorithms (henceforth simply referred to as *adaptive* algorithms) collect and maintain simple statistics about past commit and abort events. Unlike partially-adaptive algorithms, adaptive algorithms collect these statistics *across transaction boundaries*. With adaptive algorithms, threads may oscillate an arbitrary number of times between serializing and conventional modes of operation, even in the course of performing a single transaction. An adaptive algorithm can be either local or global. With *local* adaptive algorithms, each thread maintains its own statistics and may change its modus operandi independently of other threads. In *global* adaptive algorithms, centralized statistics is maintained and the modus operandi is system-wide rather than thread-specific. As we describe later in this section, under certain circumstance, adaptive algorithms may be susceptible to a phenomenon of *mode oscillation* in which they frequently oscillate between serializing and conventional modes of operation. We have therefore also evaluated adaptive algorithms enhanced with a simple *stabilizing mechanism*. We henceforth denote the local and global non-stabilized adaptive algorithms by A_L and A_G, respectively, and the respective stabilized versions by A_{LS} and A_{GS}.
  All our adaptive algorithms maintain an estimate *cl* of the contention level, that is updated whenever a commit or abort event occurs. Similarly to [23], *cl* is updated by performing exponential averaging. That is, upon a commit/abort event *e*, *cl* is updated according to the formula $cl_{new} \leftarrow \alpha \cdot cl_{old} + (1-\alpha)C$, for some $\alpha \in (0,1)$, where C is 1 if *e* is an abort event and 0 otherwise. With A_L and A_{LS}, each thread updates its own *cl* variable; with A_G and A_{GS}, all threads share a single global *cl* variable. Under algorithm A_L (A_G), a thread employs serializing CM if the value of its local (the global) *cl* variable exceeds a parameter threshold value t, and employs a conventional CM algorithm otherwise. Algorithms A_{LS} and A_{GS} use both a low threshold value tl and a high threshold value $th > tl$. They both switch to a serializing modus operandi when the value of the corresponding *cl* variable exceeds th, and revert to a conventional modus operandi when *cl*'s value decreases beyond tl. As our results establish, this simple stabilizing mechanism improves the performance of our adaptive algorithms considerably.

2.1 The Underlying Serialization Algorithm

The serialization mechanism used by both our partially-adaptive and adaptive algorithms (when they operate in serialization mode) is a low-overhead serializing algorithm, henceforth referred to as LO-SER. The pseudo-code of algorithm LO-SER appears in Figure

1. The basic idea is that every transactional thread has a condition variable associated with it. Upon being aborted, a *loser* transaction is serialized by sleeping on the condition variable of the winner transaction. When a winner transaction commits, it wakes up all threads blocked on its condition variable, if any. In our implementation, we use Pthreads conditional variables and mutexes.

The implementation of LO-SER is complicated somewhat because of the need to deal with the following issues: (1) operations on condition-variables and locks are expensive; we would like such operations to be performed only by threads that are actually involved in collisions. Specifically, requiring every commit operation to broadcast on its condition variable would incur high overhead in low-contention workloads; (2) deadlocks caused by cycles of threads blocked on each other's condition variables must be avoided, and (3) race conditions such as having a loser transaction wait on a winner's condition variable after the winner already completed its transaction must not occur.

Fedorova et al. [5] (see Section 5) implement a serializing mechanism that is very similar to LO-SER. Unlike LO-SER, however, that mechanism requires *every* commit operation to broadcast on a condition variable. We now describe algorithm LO-SER in more detail, explaining how it copes with the above issues.

We have implemented LO-SER on top of RSTM [16]. The (per-thread) variables required by LO-SER were added as new fields of RSTM's transaction descriptors (lines **4**–**7**). (To simplify presentation, field names are not prefixed with a descriptor name in the pseudo-code of Figure 1. Variables of a thread different than the one executing the code are subscripted with that thread's index.) With LO-SER, each thread has a condition variable c, and a mutex m that ensures the atomicity of operations on c and prevents race conditions, as described below (the rest of the new variables are explained when they are used.)

Upon a collision between the transactions of threads t and t' (line **25**), the current thread's contention manager function is called (line **26**) and, according to its return code, a CAS operation is called to abort either the current thread's transaction (line **33**) or the other thread's transaction(line **30**).[1] The CAS operation of line 30 atomically writes 3 values: a code indicating that the transaction is aborted, the index of the winning thread[2], and the timestamp of the winning transaction. To facilitate atomic update of these 3 values, they are packed within a single 64-bit word (see line **1**). Each thread t maintains a timestamp, initialized to 0 and incremented whenever a transaction of t commits or aborts. As we soon describe, timestamps are used for preventing race condition that may cause a transaction to serialize behind a wrong transaction.

A transaction that aborts itself in line **33** proceeds immediately to execute the code of lines lines **10–24**. A transaction aborted by another thread will execute this code when it identifies it was aborted. Upon abort, a thread first rolls back its transaction (line **11**) by initializing its descriptor and releasing objects captured by it (not shown in Figure

[1] An RSTM CM may also return a WAIT code in line **26**, indicating that the calling transaction should retry its conflicting operation. Our implementation does not change RSTM's behavior in this case. Similarly, failures of CAS operations are dealt with by RSTM as usual. Both scenarios are therefore not described by the high-level pseudo-code provided by Figure 1.

[2] Upon starting, a transactional thread registers and receives an index to the transaction descriptors table. When it exits, the corresponding descriptor becomes available.

Fig. 1. Low-Overhead Serialization Algorithm (LO-SER) Pseudo-code for thread t

```
1  struct extendedStatus
2      status: 2, winner: 12, winnerTs: 50:
       int
3  end
4  struct TransDesc
5      m: mutex, c: cond, ts initially 0: int,
6      release = F, waiting = F: boolean,
7      eStat: extendedStatus,
       // RSTM descriptor fields
8      CM: CMFunction, ...
9  end
10 upon ABORT of thread t's transaction
11     Roll back t's transaction
12     ts++
13     w=eStat.winner
14     if eStat_w.status == ACTIVE then
15         waiting=true
16         m_w.lock()
17         release_w=true
18         if ts_w==eStat.winnerTs
           ∧(¬waiting_w) then
19             c_w.wait(m_w)
20         end
21         m_w.unlock()
22         waiting=false
23     end
24 end
```

```
25 upon Collision(t, t')
26     int res = t.CM(t')
27     if res == ABORT_OTHER then
28         int v = t'.eStat
29         if v.status == ACTIVE then
30             t'.eStat.CAS
                   (v,<ABORT,t,t.ts>)
31         end
32     else if res == ABORT_SELF then
33         t.eStat = <ABORT,t',t'.ts>
34     end
35 end
36 upon COMMIT by thread t
37     ts++
38     if eStat.CAS(<ACTIVE,0,0>,
               <COMMITTED,0,0>)
       then
39         if release then
40             m.lock()
41             c.broadcast()
42             release=false
43             m.unlock()
44         end
45     end
46 end
```

1). It then increments its timestamp (line **12**). If the thread, w, causing the abort, is in the midst of an active transaction (lines **13–14**), it may still be performing the respective winning transaction. Thread t now prepares for serialization. It first sets its *waiting* flag (line **15**) to prevent other transactions from serializing behind it and avoid waiting cycles. It then tries to acquire w's lock (line **16**). After acquiring w's lock, t sets w's *release* flag (line **17**) to indicate to w that there may be waiting threads it has to release when it commits. If w did not yet change its timestamp since it collided with t's transaction, and if w is currently not serialized behind another thread itself (line **18**), then t goes to sleep on w's condition variable by calling $c_w.wait(m_w)$ (line **19**). A reference to m_w is passed as a parameter so that the call releases m_w once t blocks on c_w; this is guaranteed by the Pthreads implementation. The Pthreads implementation guarantees also that when t is released from the waiting of line **19** (after being signalled by w) it holds m_w. Thread t therefore releases m_w in line **21** and resets its *waiting* flag (line **22**).

Upon committing, thread t first increments its timestamp (line **37**) to prevent past losers from serializing behind its future transactions. Then t commits by performing a

CAS on its status word (line **38**) (see Footnote 1). If it succeeds and its *release* flag is set (line **39**), then t acquires its lock, wakes up any threads that may be serialized behind it, resets its *release* flag and finally releases its lock (lines **40–43**).

We prove the following properties of the LO-SER algorithm in the full paper.

Lemma 1. *The LO-SER algorithm satisfies the following properties:*

1. *At all times, threads waiting on condition variables (at line* **19***) do not form a cycle.*
2. *If a thread waits (at line* **19***) on the condition variable of some thread w, then w is in the midst of performing a transaction. That is, its status is either ACTIVE or ABORTED.*
3. *Assume thread p performs a broadcast (at line* **41***) at time t, and let $t' < t$ denote the previous time when p performed a broadcast, or the time when the algorithm starts if no broadcast by p was performed before t. Then p won some collision during the interval (t',t).*

3 CBench

Several benchmarks have been introduced in recent years for evaluating the performance of TM systems. Some of these benchmarks (e.g., RSTM's micro-benchmarks) implement simple TM-based concurrent data-structures such as linked list and red-black tree, while others use more realistic applications (e.g. STMBench7 [10] and Swarm [21]). However, to the best of our knowledge, none of these benchmarks allow generating workloads with an accurate pre-determined contention level.

We introduce *CBench* - a novel benchmark that generates workloads of pre-determined length where each transaction has a pre-determined probability of being aborted when no contention reduction mechanism is employed. Both transaction length and abort-probability are provided as CBench parameters. CBench facilitates evaluating the performance of both serializing and conventional CM algorithms as contention varies, which, in turn, makes it easier to study the impact of these algorithms, in terms of their effect on throughput and efficiency, across the contention range.

Using CBench is composed of two stages: *calibration* and *testing*. In the calibration stage, CBench is called with a *TLength* parameter, specified in units of accesses to transactional objects. E.g, when called with *TLength=1000*, the total number of transactional object accesses made by each transaction is 1000. A CBench transaction accesses two types of transactional objects: *un-contended* objects and *contended* objects. Un-contended objects are private to each thread, and accesses to them are used to extend transaction length. Contended objects, on the other hand, may be accessed by all transactions and are used to control contention level.

The calibration stage of CBench proceeds in iterations. At the beginning of each iteration, CBench fixes the numbers of reads and writes of contended objects (respectively called *contended-reads* and *contended writes*) to be performed by each transaction; which objects are to be accessed by each transaction is determined randomly and uniformly before the iteration starts. CBench then creates a single thread per system core and lets these threads repeatedly execute transactions for a pre-determined period of time,

during which it counts the total number of commits and aborts incurred. During this execution, the contention manager used is the *RANDOM* algorithm.[3] Let *commits* and *aborts* respectively denote the total number of commits and aborts that occur during the execution of the iteration's workload, then its respective abort probability is given by *aborts/(commits+aborts)*. This value is recorded by CBench (in a CBench data file) and is associated with the respective numbers of contended reads and writes. The calibration process continues performing iterations, varying the numbers of contended reads and writes, until a fine-resolution "coverage" of all contention levels has been generated. That is, for every abort probability p, the CBench data file contains an entry specifying the numbers of contended reads and writes required to create a workload with abort probability p' such that $|p - p'| < 1\%$.

In the testing stage, CBench is called with an abort-probability parameter. The entry with the closest abort-probability is read from CBench's data file (created for the required transaction length) and is used to generate workloads with the input contention level. CBench then creates one thread per core and starts generating its test workload.

4 Experimental Evaluation

All our experiments were performed on a 2.60 GHz 8 core 4xXEON-7110M server, with 16GB RAM and 4MB L2 cache and with HyperThreading disabled, running the 64-bit Gentoo Linux operating system, and using RSTM. Our experiments evaluate two partially-adaptive algorithms (PA_1 and PA_{100}), all our fully-adaptive algorithms (A_L, A_{LS}, A_G and A_{GS}), a few conventional CM algorithms (Random, Polka and Greedy, configured with the invisible-readers and lazy abort settings), and CAR-STM.

4.1 CBench

Figure 2-(a) shows the throughput of some of the algorithms we evaluated, across the full contention spectrum of workloads generated by CBench with *TLengh=1500*. We observe that the throughput of PA_{100} and Polka is practically identical (and we therefore only show a single bar representing both). The reason for that is that PA_{100} employs Polka until a transaction collides for the 100'th time, thus it rarely performs serialization. Since all fully-adaptive algorithms behave similarly for low and medium contention levels, we only show A_{LS} in Figure 2-(a).

CAR-STM has high overhead, causing its throughput to be less than 60% that of all other algorithms in the lack of contention. The key reason for that is that, with CAR-STM, a transaction is executed by a dedicated transaction-queue thread and not by the thread that created it (see [3]). Our serialization algorithms, on the other hand, do not seem to incur any overhead in the lack of contention and perform as well as the conventional CM algorithms when contention is low. When contention increases, the throughput of the conventional CM algorithms quickly deteriorates, and, for very high contention levels, they are significantly outperformed by the serializing algorithms.

[3] Upon a collision, *RANDOM* randomly determines which transaction should abort and the aborted transaction immediately restarts. Since *RANDOM* does not apply any contention-reduction technique, it is a good baseline for assessing the impact of contention reduction algorithms.

Comparing PA_1 and PA_{100} reveals that they provide more-or-less the same throughput for low contention levels. For medium contention levels (0.3-0.8), PA_{100}'s throughput is higher by up to 8%, but for very high contention-levels PA_1 is the clear winner and exceeds PA_{100}'s throughput by up to 60%. This behavior can be explained as follows. In low contention-levels there are hardly any collisions so both algorithms behave in the same manner. In medium contention-levels there are a few collisions, but the probability that two colliding transactions will collide again is low; under these circumstances, it is better to use Polka rather than to serialize, and serialization incurs the cost of waiting until the winner transaction terminates. Finally, under high-contention, serialization is the better strategy and using a conventional CM such as Polka is inefficient. A_{LS} obtains the "the best of all worlds" performance: it is as good as PA_{100} when contention is low, and as good as PA_1 when contention is high.

Zooming into high-contention workloads (Figure 2-(b)) highlights the impact of the stabilization mechanism. Whereas A_L and A_{LS} provide the same throughput up to contention-level 0.8, under high-contention A_{LS} outperforms A_L by up to 16%. This is because, under high-contention, A_L oscillates between serialization and non-serializing modi operandi: when contention exceeds A_L's threshold, it starts serializing. When serializing decreases the level of contention, A_L reverts to non-serialization mode, which hurts its performance. A_{LS} eliminates this oscillation and therefore obtains higher throughput. The behavior of A_L and A_G is almost identical on CBench workloads, therefore A_G's curve is not shown in Figure 2-(b). Similarly to the local adaptive algorithm, A_{GS} provides higher throughput than A_G under high contention but by a smaller margin (up to 6%).

Figure 2-(c) shows the efficiency of the evaluated algorithms. Efficiency deteriorates as contention-level increases. CAR-STM's efficiency is best, since it limits parallelism more than all other algorithms. When contention is maximal, CAR-STM obtains the highest efficiency (0.41) among all algorithms, and PA_1 together with A_{LS} are second best (but far behind with 0.13 efficiency). All other low-overhead serialization algorithms have efficiency levels between 0.09-0.12, whereas Polka, Greedy and Random obtain the lowest efficiency figures (0.05, 0.03 and less than 0.01, respectively).

4.2 RandomGraph

RandomGraph [15] operates on a graph data-structure consisting of a linked-list of vertices and of linked-lists (one per vertex) for representing edges. Supported operations are: (1) insertion of a node and a set of random edges to existing vertices, and (2) the removal of a node and its edges. The number of vertices is restricted to 256, which generates workloads characterized by high-contention and relatively long transactions. The impact of all serialization algorithms on RandomGraph's throughput is considerable, as can be seen in Figure 3-(a). (Since the throughputs of A_L and A_G are very similar on RandomGraph, the curves of A_G and A_{GS} are not shown.)

Under high contention, all serializing CM algorithms provide throughput which is orders of magnitude higher than that of the conventional algorithms. A_{LS} and A_{GS} significantly outperform A_L and A_G, respectively, for all concurrency levels higher than 1. The stabilizing mechanism improves the throughput of the local adaptive algorithm by approximately 30% for 2 threads, and by almost 50% for 4 threads. The gap decreases

for 8, 16 and 32 threads (18%, 13% and 8%, respectively) but remains significant. The contribution of stabilization to the global adaptive algorithm is somewhat smaller but is also significant, and reaches its maximum for 2 threads (19%).

Fig. 2. CBench Evaluation Results

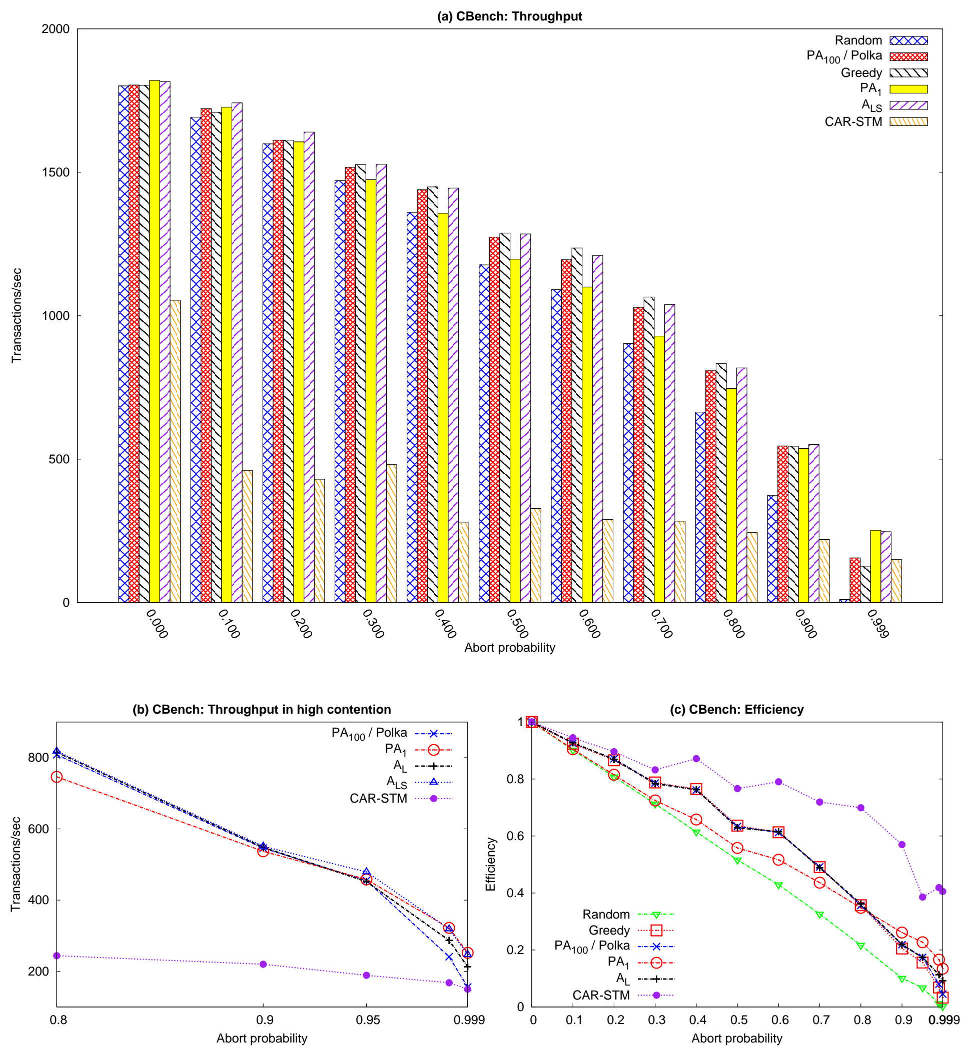
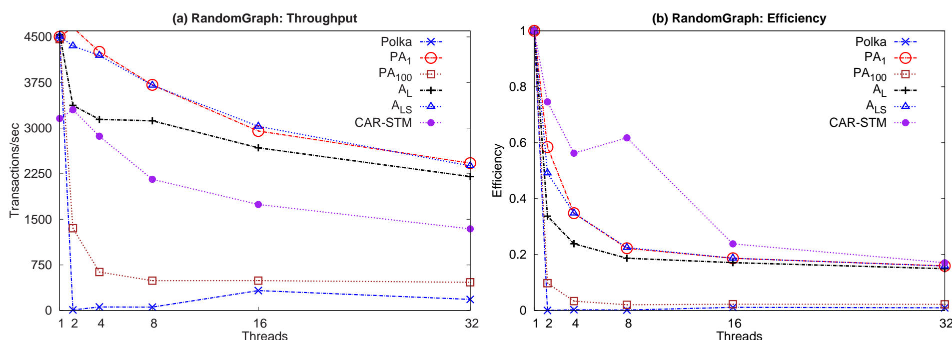


Fig. 3. RandomGraph Evaluation Results



When analyzing this behavior, we find that both the local and the global adaptive algorithms suffer from frequent oscillations between serializing and conventional modes of operation. The stabilization mechanism greatly reduces these oscillations. The local algorithm suffers from this phenomenon more than the global one, since the probability that a single thread will win a few collisions in a row is non-negligible; this thread will then shift to a conventional modus operandi. When *all threads* update history statistics, such fluctuations are more rare. CAR-STM's performance is much better than the conventional algorithms, but much worse than the low-overhead serializing algorithms. Observe that, due to the high-contention nature of RandomGraph workloads, the throughput of all algorithms is maximum when concurrency level is 1.

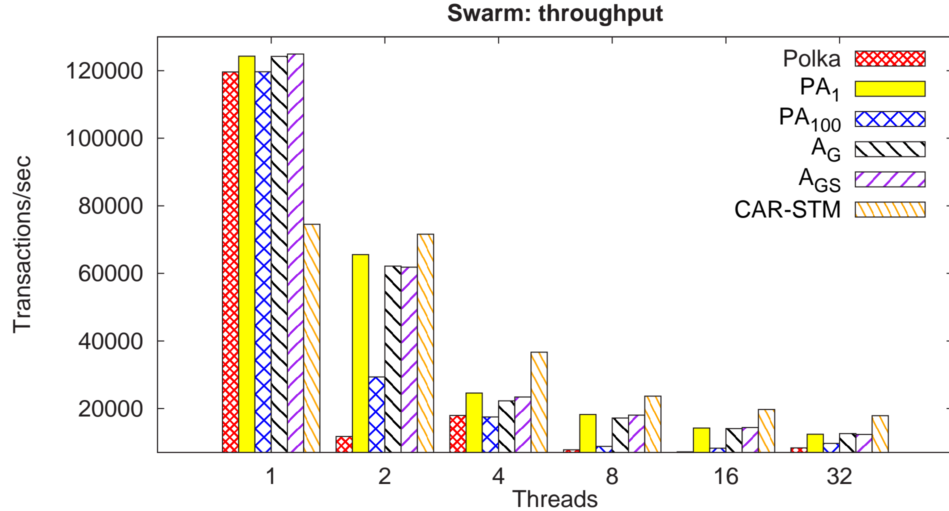
Figure 3-(b) presents the efficiency of the algorithms we evaluate on RandomGraph-generated workloads. It can be seen that, in general, algorithms that serialize more have higher efficiency. The conventional CM algorithms (Polka, Greedy and Random) have the lowest efficiency (since their efficiency is almost identical, we only show Polka's curve), followed closely by PA_{100}. CAR-STM has the best efficiency across all the concurrency range by a wide margin and PA_1 is second best.

4.3 Swarm

Swarm [21] is a more realistic benchmark-application that uses a single transactional thread for rendering objects on the screen, and multiple transactional threads for concurrently updating object positions. Swarm generates workloads characterized by very high contention.

Figure 4 shows the throughput of some of the algorithms we evaluated on the Swarm benchmark. The throughput of all the conventional algorithms (Polka, Greedy and Random) is the lowest; since they behave roughly the same, only Polka is shown. The throughput of PA_{100} also deteriorate very quickly and, with the exception of 2 threads, its performance is very similar to that of Polka. The best algorithm on Swarm in almost all

Fig. 4. Throughput on Swarm-generated workload



concurrency levels is CAR-STM. It seems that its strong serialization works best for Swarm's workloads. The stabilized algorithms A_{LS} and A_{GS} are slightly better than their non-stabilized counterpart algorithms in almost all concurrency levels, but the gap is very small. The reason for that is that Swarm's contention levels are so high, that A_L and A_G operate almost constantly in serialization mode. Also here, the throughput of A_L and A_G is almost identical, hence we only show the graphs of the global adaptive algorithm. The relation between the efficiency levels of evaluated algorithms on Swarm is similar to that seen on RandomGraph, hence an efficiency graph is not shown.

5 Discussion

In this paper we investigated the influence of serializing contention management algorithms on the throughput and efficiency of STM systems. We implemented CBench, a synthetic benchmark that generates workloads possessing pre-determined (parameter) length and contention-level. We also implemented LO-SER - a novel low-overhead serialization algorithm and, based on it, several adaptive and partially-adaptive CM algorithms. We then evaluated these algorithms on workloads generated by CBench, by an RSTM micro-benchmark application, and by the Swarm application.

Several significant conclusions can be drawn from our work. First, we observe that CM algorithms that apply a low-overhead serializing mechanism adaptively can significantly improve both STM throughput and efficiency in high-contention workloads while, at the same time, incurring no visible overheads for low-contention workloads. Second,

we have demonstrated the importance of incorporating a stabilizing mechanism to (both local and global) adaptive algorithms, for preventing mode oscillations that hurt performance under high contention.

Recent work addresses serializing CM algorithms from various perspectives. Dragojevic et al. [4] presented *Shrink*, a user-level transaction scheduler that bases its scheduling decisions on the access patterns of past transactions. *Shrink* can therefore prevent collisions before they occur. *Shrink* uses a serialization mechanism similar to that of Yoo and Lee [23]. Fedorova et al. [5] propose and evaluate several kernel-level scheduling-based mechanisms for TM contention management. They also implement a serializing mechanism that is very similar to LO-SER. Unlike LO-SER, however, that mechanism requires *every* commit operation to broadcast on a condition variable; with LO-SER, broadcast system calls are avoided in the absence of collisions.

Our evaluation did not discover significant performance/efficiency differences between global and local adaptive serializing CM algorithms. However, all the applications we have used to evaluate our algorithms are symmetric, in the sense that all threads perform the same algorithm (with the exception of Swarm, where a *single* thread performs a different algorithm than all other threads). It seems plausible that local adaptive policies will be superior for asymmetric applications. We intend to investigate this issue in future work.

Adaptive serializing CM algorithms use threshold values to determine when to switch from a conventional mode to a serializing mode and vice versa. The adaptive algorithms presented in this paper use fixed thresholds that were tuned manually per application. A mechanism for automatically finding the optimal thresholds and for dynamically adjusting them will increase the usability of adaptive CM algorithms and is also left for future work.

References

1. M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *HiPEAC*, pages 4–18, 2009.
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. S. Dolev, D. Hendler, and A. Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *PODC*, pages 125–134, 2008.
4. A. Dragojevic, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC*, pages 7–16, 2009.
5. A. Fedorova, P. Felber, D. Hendler, J. Lawall, W. Maldonado, P. Marlier, G. Muller, and A. Suissa. Scheduling support for transactional memory contention management. *PPoPP* 2010, to appear.
6. P. Felber, T. Riegel, and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In *PPOPP*, pages 237–246, Feb. 2008.
7. K. Fraser. Practical lock-freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, 2004.
8. R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *DISC*, pages 303–323, 2005.

9. R. Guerraoui, M. Herlihy, and B. Pochon. Towards a theory of transactional contention managers. In *PODC*, pages 316–317, 2006.
10. R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.
11. T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, 2003.
12. M. Herlihy. SXM software transactional memory package for C#. http://www.cs.brown.edu/ mph.
13. 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.
14. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.
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. 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.
19. M. L. Scott and W. N. Scherer III. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248, 2005.
20. N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
21. M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and exploiting inevitability in software transactional memory. In *ICPP*, pages 59–66, 2008.
22. G. Vossen and G. Weikum. Transactional information systems, , morgan kaufmann, 2001.
23. R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA*, pages 169–178, 2008.