On the Impact of Serializing Contention Management on STM Performance[1]

Tomer Heber, Danny Hendler and Adi Suissa
Ben-Gurion University

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 changes. 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 mechanism. We then evaluate our new algorithms on CBench-generated workloads and on additional well-known STM benchmark applications.

Our results shed light on the manner in which serializing CM should be used by STM systems. We show that adaptive contention managers are susceptible to a phenomenon of mode oscillations – in which serialization is repeatedly turned on and off – which hurts performance. We implement a simple stabilizing mechanism that solves this problem. We also compare the performance of *local* and *global* adaptive CM algorithms and demonstrate that local adaptive algorithms are superior for applications with asymmetric workloads.

[1]A preliminary version of this paper appeared in the proceedings of the *13th International Conference On Principles Of Distributed Systems*, Nimes, France, 2009, pages 225-239. This work was supported in part by by the *Israel Science Foundation* (grant number 1227/10).

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) [21; 29] 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 [31], 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 [29], ensures transactional semantics through software mechanisms; many STM implementations have been proposed in recent years [9; 12; 13; 17; 18; 19; 23; 24; 26].

TM implementations typically delegate the task of conflict resolution to a separate *contention manager* (CM) module [19]. 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 [3; 14; 15; 27] 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 works introduced transaction scheduling for increasing the efficiency of contention management. Dolev, Hendler and Suissa [10] 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.

Yoo and Lee [32] introduced *ATS* – a simple user-level transaction scheduler, and incorporated it into RSTM [24] – a TM implementation from the University of Rochester – and into LogTM [25], 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. [2] proposed *steal-on-abort*, a transaction scheduler that avoids wasted work by allowing transactions to "steal" conflicting transactions so that they execute serially.

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_1 has to resume waiting (typically for a longer period of time), or, alternatively, the CM may now decide that T_1 wins and so T_2 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_1 is too long, then T_1 may be unnecessarily delayed beyond the point when T_2 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.

1.2 Our Contributions

We investigate how serializing CM influences the performance of STM systems in terms of both *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. Our contributions towards obtaining this goal are the following.

(1) 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. We evaluate these algorithms on workloads generated by CBench (described shortly), by RSTM's micro-benchmarks and Swarm application, and by the STAMP benchmark suite [8]. Our adaptive algorithms are

based on a novel low-overhead serializing CM implementation that we describe in Section 2. Our results establish that applying serializing CM adaptively can improve STM performance considerably for high-contention workloads, while incurring only negligible overhead for low-contention, highly-parallel workloads.

(2) 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.

(3) Our empirical evaluation shows that adaptive contention managers are susceptible to a phenomenon of mode oscillations, in which the adaptive algorithm oscillates between serializing and conventional modes of operation. We show that these mode oscillations hurt performance, thus highlighting the importance of *stabilized* adaptive algorithms. We implement a simple stabilizing mechanism and show that it improves performance considerably for some workloads.

(4) We compare local-adaptive and global-adaptive CM algorithms. Our results establish that they provide comparable performance for symmetric workloads, but that local-adaptive algorithms are superior for asymmetric applications, where the workloads performed by different threads are significantly different.

The rest of this 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. In Section 5, we compare the performance of local-adaptive and global-adaptive CM algorithms when used by asymmetric applications. In Section 6, we prove the correctness of our low-overhead serialization algorithm. We summarize the paper and discuss related work in Section 7.

2. ALGORITHMS

In this section we describe the new serializing CM algorithms we've implemented and report on the results of our experimental evaluation.

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 (such as the well-known Polka or Greedy algorithms [27]) 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.

—*fully-adaptive*: these algorithms (henceforth simply referred to as *adaptive* algorithms) collect and maintain simple statistics about past commit and abort events. They maintain an estimate cl of the contention level, that is updated whenever a commit or abort event occurs. Similarly to [32], 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 and 0 otherwise. Thus, unlike partially-adaptive

Table 1: Novel 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

algorithms, adaptive algorithms collect statistics *across transaction boundaries.* 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.

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 updates its own local *cl* variable and so may change its modus operandi independently of other threads. In *global* adaptive algorithms, all threads share a single system-wide *cl* variable. 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}. 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 below *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 novel low-overhead serializing algorithm, henceforth referred to as LO-SER. The pseudo-code of algorithm LO-SER appears in Figures 1-4. 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 [7] conditional variables and mutexes.

The implementation of LO-SER is complicated somewhat because of the need to deal with the following requirements: (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 unnecessarily incur high overhead in low-contention workloads; (2) deadlocks caused by cycles of processes 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 win-

Fig. 1: The structures used by LO-SER

```
1  struct extendedStatus
2      status: 2, winner: 12, winnerTs: 50: int
3  end
4  struct TransDesc
5      m: mutex
6      c: cond
7      ts = 0: int
8      release = false: boolean
9      eStat: extendedStatus
       // RSTM descriptor fields
10     CM: CMFunction, ...
11 end
```

ner already completed its transaction must not occur. A user-level serializing CM algorithm that is similar to LO-SER is described in [22]; however, that algorithm does not satisfy requirement (1) and has to perform expensive broadcast system calls also in the lack of contention. We now describe algorithm LO-SER in more detail, explaining how it satisfies the above requirements. We prove the correctness of the LO-SER algorithm in Section 6.

We implemented LO-SER on top of RSTM [24]. In RSTM, each transactional thread owns a static structure called the "Transaction Descriptor" which is used by all transactions of that thread. The LO-SER algorithm uses an extended RSTM descriptor (see Figure 1). The transaction descriptor contains the status of the current transaction (ACTIVE, COMMITTED or ABORTED), a contention manager instance and some other data required by RSTM. The (per-thread) variables required by LO-SER were added as new fields to RSTM's transaction descriptor (lines **5**–**9**). 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 and structure-fields are described soon).

The pseudo-code for handling collisions is shown in Figure 2. Upon a collision between transactions t and t' (line **12**), the current thread's contention manager function is called (line **16**) and, according to its return code, either the current thread's transaction is set to be aborted (line **20**) or the other thread's transaction (line **18**) is set to be aborted by performing a CAS operation.[2] The CAS operation of lines 18 and 20 atomically writes 3 values: a code indicating that the transaction is aborted, the index of the winning thread, and the time-stamp 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 maintains a time-stamp, initialized to 0 and incremented whenever a transaction of the thread commits or aborts[3]. As we soon describe, timestamps are used for preventing race conditions that may cause a thread to serialize when it shouldn't.

A thread that aborts its transaction in line **20** proceeds immediately to execute

[2]A failure of the CAS operation or WAIT returned by the contention manager (see line **16**) are dealt with by RSTM as usual. Both scenarios are not described by the high-level pseudo-code provided here.

[3]Since timestamps monotonically increase, they might overflow. We assume that on a 64-bit machine an overflow is an unrealistic scenario.

Fig. 2: Handling a collision between two transactions

```
12 upon Collision(t, t')
13     int r = t'.ts
14     int v = t'.eStat
15     if v.status == ACTIVE then
16         int res = t.CM(t')
17         if res == ABORT_OTHER then
18             t'.eStat.CAS (v,<ABORT,t,t.ts>)
19         else if res == ABORT_SELF then
20             t.eStat.CAS(<ACTIVE,0,0>,<ABORT,t',r>)
21         end
22     end
23 end
```

the abort pseudo-code (see Figure 3). A thread whose transaction was aborted by another thread will execute this code when it identifies that its transaction was aborted. Upon abort, a thread first rolls back its transaction (line **25**) by initializing its descriptor and releasing transactional objects acquired by it (not shown in Figure 1). It then increments its time-stamp (line **26**). If the thread, w, causing the abort, is in the midst of an active transaction (lines **27**–**28**) then thread v prepares for serialization. It tries to acquire w's lock (line **29**). After acquiring w's lock, it checks whether w's active transaction is still the respective winning transaction (line **30**). If this is not the case, then thread v unlocks m_w and does not serialize behind w.

Otherwise, v sets w's *release* flag (line **31**) to indicate to w that there are waiting threads it has to release when it commits. If w did not yet change its time-stamp since it collided with v's transaction (line **32**), then v goes to sleep on w's condition variable by calling $c_w.wait(m_w)$ (line **33**). A reference to m_w is passed as a parameter so that the call releases m_w once v blocks on c_w; this is guaranteed by the Pthreads implementation. Pthreads guarantees also that when v is released from the waiting of line **33** (after being signaled by w) it holds m_w. Thread v therefore releases m_w in line **36**.

Fig. 3: Handling a thread's transaction abort

```
24 upon ABORT of thread v's transaction
25     Roll back v's transaction
26     ts++
27     w=eStat.winner
28     if $eStat_w$.status == ACTIVE then
29         $m_w$.lock()
30         if $ts_w$==eStat.winnerTs then
31             $release_w$=true
32             if $ts_w$==eStat.winnerTs then
33                 $c_w$.wait($m_w$)
34             end
35         end
36         $m_w$.unlock()
37     end
38 end
```

Fig. 4: Handling a thread's transaction commit

```
39 upon COMMIT by thread w
40     boolean b = eStat.CAS(<ACTIVE,0,0>,<COMMITTED,0,0>)
41     ts++
42     if $b \wedge release$ then
43         m.lock()
44         c.broadcast()
45         release=false
46         m.unlock()
47     end
48 end
```

Upon committing, thread w first attempts to change the status of its transaction to COMMITTED by performing a CAS on its status (line **40** in Figure 4. See Footnote 2). Then w increments its time-stamp (line **41**) to prevent past losers from serializing behind its future transactions. If the CAS succeeds and its *release* flag is set (line **42**), then w acquires its lock(line **43**), *wakes up* any threads that may be serialized behind it, resets its *release* flag and finally releases its lock (line **46**).

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 [16], Swarm [30] and STAMP [8]). 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 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 trnasaction is determined randomly and uniformly before the iteration starts. CBench then creates a single thread per system core and lets these threads run 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.[4] 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 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 5 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 5.

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. 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.

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

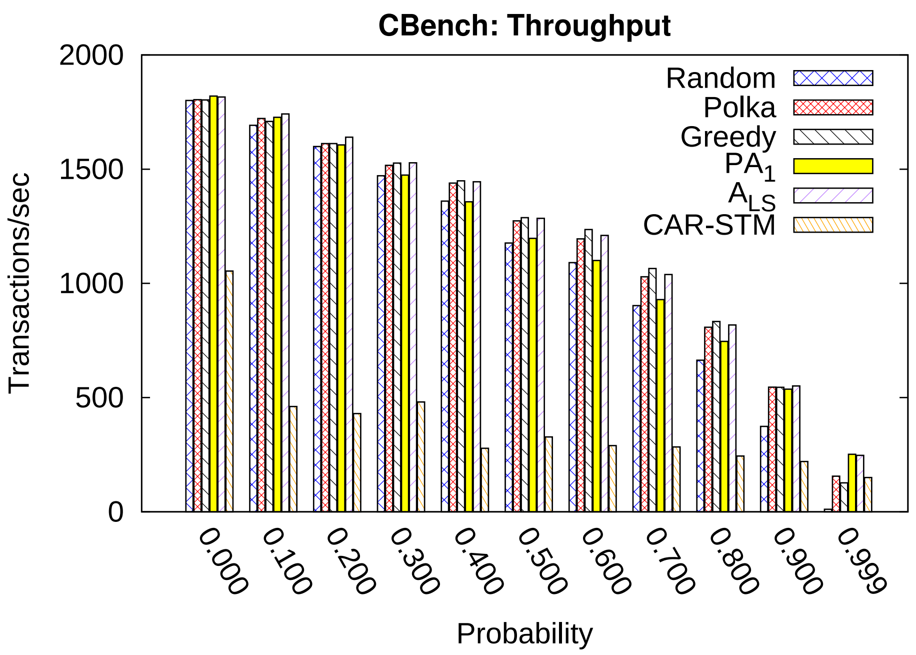

Fig. 5: CBench Throughput Results

Comparing PA_1 and PA_{100}/Polka 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 the same. In medium contention-levels there are a few collisions, but the probability that two colliding transactions will collide again is still 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 6) 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 modes of operation: when contention exceeds A_L's threshold, A_L starts serializing. When serializing reduces 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 6. 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 7 shows the efficiency of the evaluated algorithms. Efficiency deteriorates

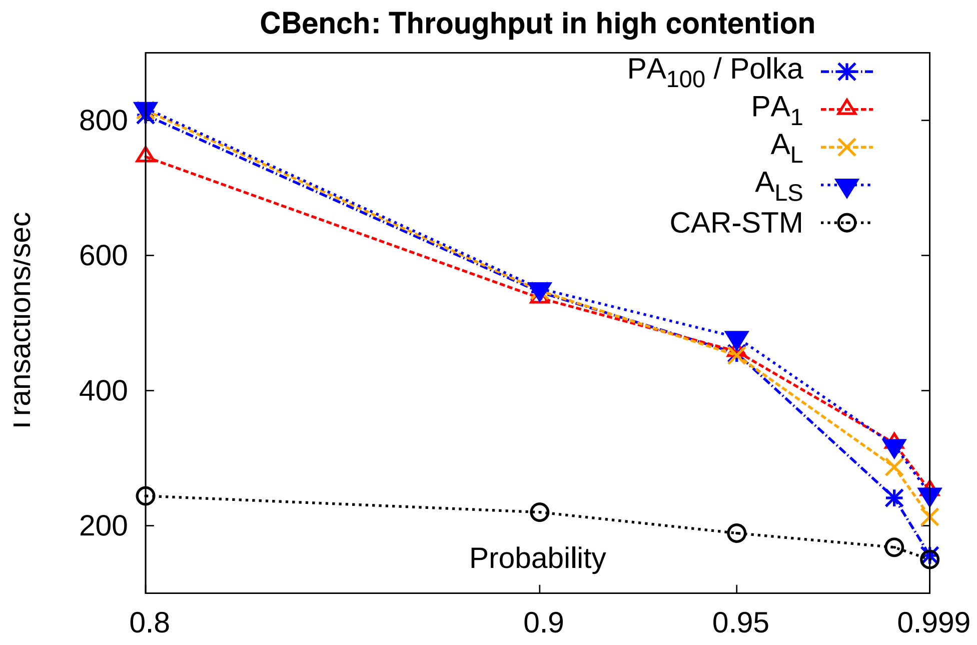

Fig. 6: CBench High Contention Throughput Results

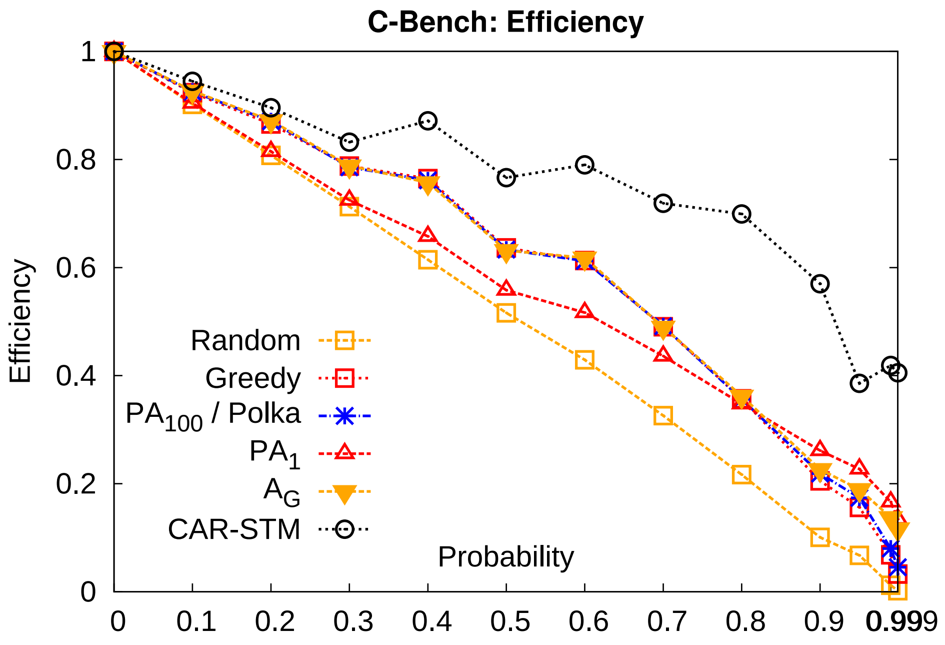

Fig. 7: CBench efficiency Results

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 obtain between 0.09-0.12 efficiency, whereas Polka, Greedy and Random obtain the lowest efficiency figures (0.05, 0.03 and less than 0.01, respectively).

4.2 Random Graph

RandomGraph [23] 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 8. (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 algo-

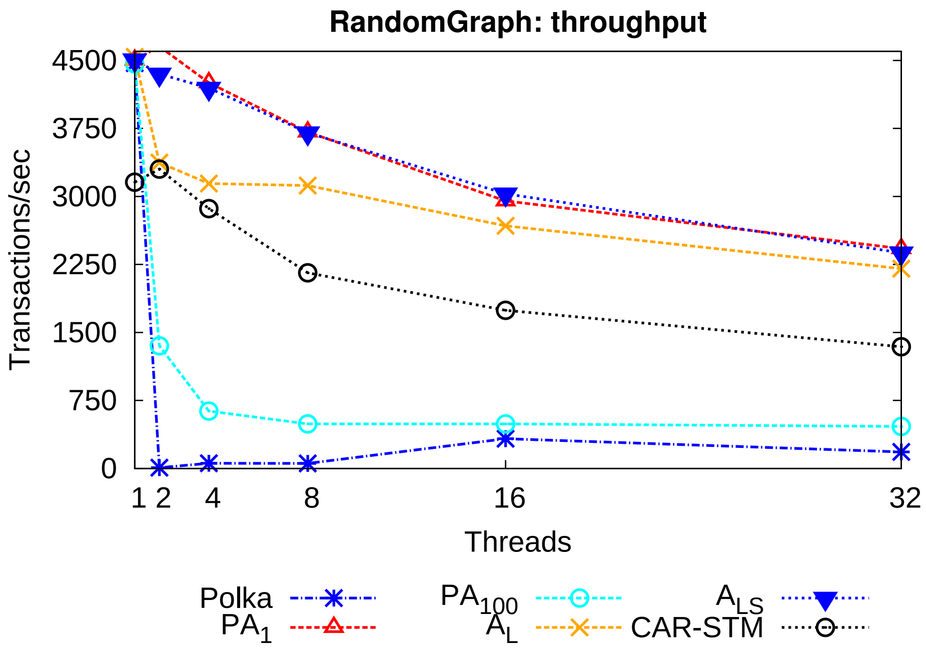

Fig. 8: RandomGraph Throughput Results

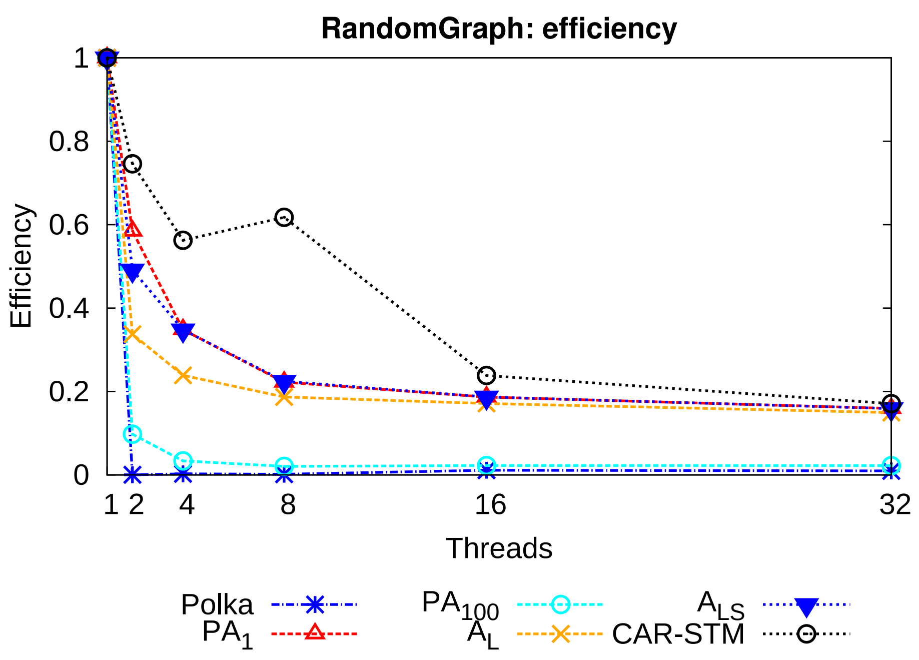

Fig. 9: RandomGraph Efficiency Results

rithm is somewhat smaller but is also significant, and reaches its maximum for 2 threads (19%).

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 maximal when concurrency level is 1.

Figure 9 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 [23] 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.

Fig. 10: Throughput on Swarm-generated workload

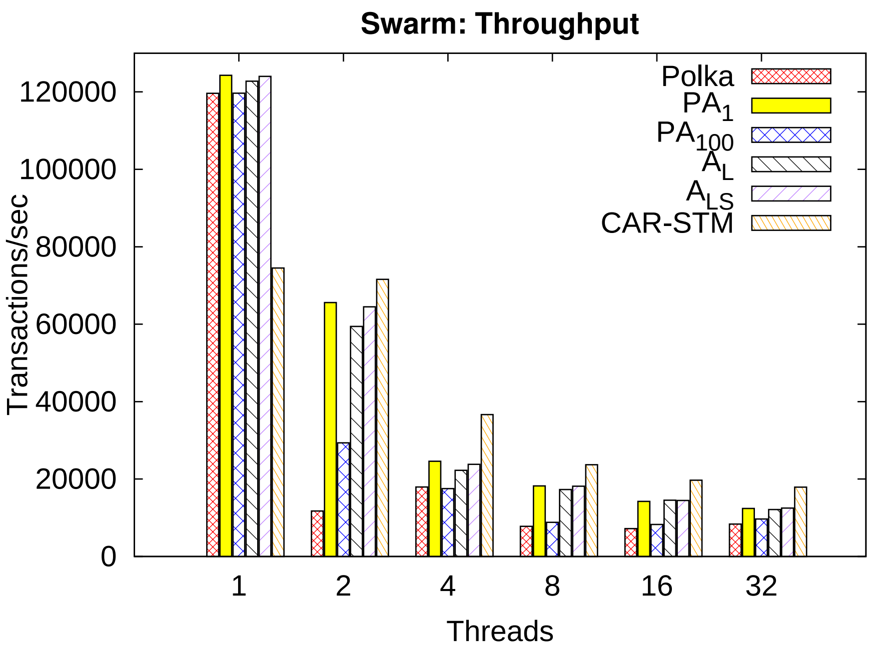

Figure 10 shows the throughput of some of the algorithms we evaluated on the Swarm benchmark. The throughput of all the conventional algorithms (Polka, Greedy and Random) is the lowest; since they behave roughly the same, only Polka is shown. The throughput of PA_{100} also deteriorate very quickly and, with the exception of 2 threads, its performance is very similar to that of Polka. The best algorithm on Swarm in almost all 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. This follows from the fact that Swarm's contention levels are so high, that A_L and A_G operate almost constantly in serialization mode. Also here, the throughputs of A_L and A_G are almost identical, hence we only show the graphs of the local-adaptive algorithm. The relation between the efficiency of evaluated algorithms on Swarm is similar to that seen on RandomGraph workloads, hence an efficiency graph is not shown.

4.4 The STAMP benchmark

The STAMP benchmark suite [8] was introduced to evaluate both hardware and software transactional memory implementations. This suite is comprised of 8 applications with various transactional workloads which we now briefly describe.

The *bayes* application learns a Bayesian network structure which is represented as a directed acyclic graph and uses transactions to protect the calculation and the addition of dependencies (edges) between the variables (nodes). This application is characterized by high contention workloads because threads spend most of their time executing long transactions, each accessing many transactional objects. The

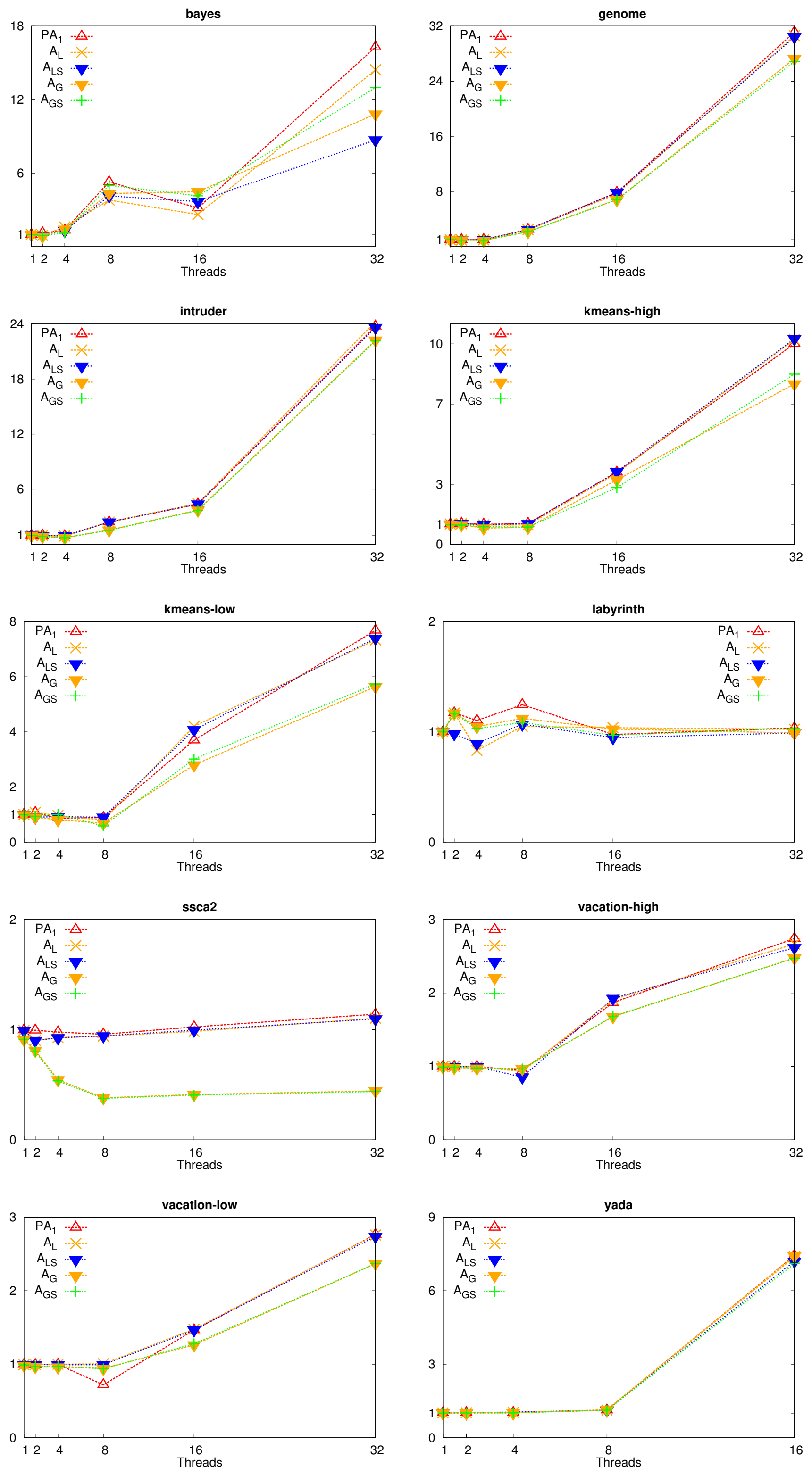

Fig. 11: Speedup of STAMP benchmarks compared to the suicide contention manager throughput

genome application reconstructs a source genome from many small DNA segments and, to this end, uses transactions to access a hash set and a pool. These are both highly parallel data structures and therefore the *genome* application generates low contention workloads. The *intruder* application emulates a network intrusion detection system which captures fragmented packets, reassembles the fragments of a single session to a single packet and compares the packet against a known set of intrusion signatures. Transactions access a simple FIFO queue and a self-balancing tree which results in high contention workloads.

The *kmeans* application is a clustering application that uses short transactions to synchronize two threads that access the same cluster center. The benchmark supports two variants: *kmeans-low* and *kmeans-high*, which produce very low and low contention workloads, respectively. The *labyrinth* application attempts to find non intersecting paths between many pairs of cells in a three dimensional maze. Each thread iteratively attempts to find and acquire a path between a single pair of cells by first searching for an unused path, and then transactionally marking path cells as occupied. The intersection of paths produces moderate contention workloads.

The *ssca2* application creates an adjacency array based graph. The addition of new vertices and edges requires very short transactions which results in low contention workloads. The *vacation* application emulates a traveling agency reservation system in which each transaction either creates a new reservation, updates a current reservation or cancels a reservation. The benchmark supports two variants: *vacation-low* and *vacation-high*, which produce low and moderate contention workloads, respectively. Finally, the *yada* application is a Delaunay mesh refinement algorithm where each transaction takes a triangle that needs to be refined from a single queue, re-triangulates it and adds the outcome back to the queue until a refinement threshold is met. This application generates moderate contention workloads characterized by long transactions that spend most of their time in re-triangulating operations.

To evaluate our algorithms on STAMP applications, we implemented our partially adaptive and fully adaptive contention management algorithms in TinySTM [12] (STAMP applications are not supported by the current object-based implementation of RSTM). Figure 11 presents the speedup of some of our algorithms compared to *suicide*, TinySTM's default contention manager, which always aborts the transaction that identifies the conflict. We evaluate all STAMP applications using $1, 2, 4, 8, 16$ and 32 threads[5], and present the average result of five runs for each thread level.

4.4.1 *Evaluation Results.* With the exception of *labyrinth* and *ssca2*, all applications exhibit improved performance under our contention management algorithms, as compared with the *suicide* contention manager, when the number of threads exceeds the number of cores. This is because, under these circumstances, our serializing contention management algorithms suspend aborted transactions thus allowing winning transactions to proceed faster.

It is noteworthy that the *bayes* speedup curve is not as "smooth" as that of the other STAMP applications. This is due to the sensitivity of the *bayes* application to the order in which network dependencies are learnt [8]. For 8 threads or more,

[5]The results for the *yada* application with 32 threads are not shown because, when *yada* is executed using the *suicide* contention manager, its running time exceeded our experiment time limit (30 minutes). Apparently the *yada* application enters a live-lock situation under these circumstances.

the speedup of the *bayes* application gained by using our CM algorithms is 5 or more. When the number of threads is 32, the speedup is between 8 (when the A_{LS} algorithm is used) and 16 (when the PA_1 algorithm is used).

In the *genome* and the *intruder* benchmarks, all our algorithms yield throughput equal to that of the *suicide* contention manager when the number of threads is $1, 2$ and 4, but the speedup they provide increases quickly as the number of threads grows from 8 to 32. For 32 threads, the speedup factors gained by our CM algorithms in the *genome* and *intruder* benchmarks are 32 and 24, respectively. These large speedup factors are due to a dramatic decrease in the number of transaction aborts when using our CM algorithms.

Our CM algorithms provide comparable throughput to that of *suicide* in the *kmeans*-high, *kmeans*-low, *vacation*-high and *vacation*-low applications when the number of threads is up to 8. When the number of threads is exactly 8, our CM algorithms are slightly worse than the suicide contention manager (by up to 20%). This is due to excessive reduction of parallelism caused by serialization. When the number of threads is higher than 8, our algorithms' throughput scales linearly with the number of threads as compared with that of *suicide*, and when the number of threads is 32 the speedup of our local-adaptive algorithms is $8 - 10$ for the *kmeans* application and 2.7 for the *vacation* application. For these applications, the local-adaptive CM algorithms outperform their global-adaptive counterparts. The reasons for this are twofold. First, local-adaptive algorithms avoid reducing the level of parallelism when only a small number of aborts is encountered. In addition, a synchronization overhead is incurred by the global-adaptive algorithms when they update the global contention intensity estimation, whereas with the local-adaptive algorithms each thread maintains its own separate contention intensity estimation.

For the *ssca2* application, the PA_1 and local-adaptive algorithms give the same throughput as *suicide*. The throughput of the global-adaptive algorithms, on the other hand, decreases as the number of threads grows. This is due to the very short transactions which characterize the workloads generated by *ssca2*. The synchronized update of the global contention intensity estimator by the global algorithms incurs a non-negligible overhead under this workload.

For the *labyrinth* application, our contention management algorithms and the *suicide* algorithm provide roughly the same throughput for all thread levels. This is because the workloads generated by *labyrinth* are characterized by relatively short transactions. Moreover, the majority of aborts are explicitly initiated by the application rather than by the contention manager. TinySTM is unaware of the identity of the conflicting transaction when a transaction aborts explicitly, hence the aborting transaction is not serialized by our CM algorithms in this case.

Finally, the relative performance of our CM algorithm w.r.t. *suicide* on the *yada* application is similar to those of *genome* and *intruder* applications: throughput is significantly increased when the number of threads is 16.

5. LOCAL VS. GLOBAL ADAPTIVE CONTENTION MANAGEMENT

In the experiments we conduced in Section 4 we did not observe significant differences between the performance of local-adaptive and global-adaptive algorithms. We hypothesize that the reason these two adaptive strategies behave roughly the same is that the test applications we used in Section 4 are *symmetric*, that is, the workloads assigned to all threads under these applications are identical.

In this section we compare the local-adaptive and global-adaptive strategies by using *asymmetric* test applications, where different threads execute significantly different workloads. Our results clearly indicate that the local-adaptive strategy is superior for such applications.

Asymmetry Caused by Different Transaction Lengths. The first asymmetric test application we used is based on the *LinkedListRelease* [20] application. The data-structure underlying this application is a linked list that stores integers sorted in an increasing order. Each list element stores an integer value and a pointer to the next list element (the pointer of the last list element stores a special *null* value). The linked-list supports the *lookup*, *insert* and *remove* operations. Let v denote the operation operand. All operations traverse the linked-list until they reach the first element that stores an integer $w \geq v$ or until they reach the end of the list. Operations are implemented as transactions that maintain the following invariant: at all times after operation initialization, exactly two (consecutive) list items are in the transaction's write-set. This invariant is maintained by applying an *early release* operation (see [20]) to any traversed list element that points to an element holding a value $w < v$ (that can thus be evicted from the transactions data-set).

Two cases exist. (1) An element e holding value v is found. In this case, a *lookup* operation returns *true*, an *insert* operation fails, and a *remove* operation removes element e by changing the pointer of the preceding element (which is in the transactions write-set) so that it points to e's successor; (2) such an element e is not found. In this case, a *lookup* operation returns *false*, an *insert* operation inserts a new element storing v, points the previous list element (which is in the transaction's write-set) to it and returns a success indication, and a *remove* operation fails. As described in [20], the early release feature increases parallelism and reduces contention as two concurrent linked list operation may succeed if they apply their operations to different parts of the list. Conflicts are detected only when one transaction modifies a list element and another transaction concurrently accesses either this element or one of its immediate neighbors. (In this case, the write-set of the first transaction intersects with the data-set of the second.)

Based on the *LinkedListRelease* application, we implemented an *AsymmetricLinkedList* application that supports two transaction types. A *short transaction* applies a single operation (*lookup*, *insert* or *remove*) to the data-structure. A *long transaction* applies 10 operations of the same type, with 10 different operand values; these operations must be applied to the list *atomically.* Thus, the data-set of a long transaction may contain up to 20 list elements[6]. In our experiments using the *AsymmetricLinkedList* application we ran a total of 8 threads, varying the number of threads that execute short transactions from 0 to 8. Both the stabilized and the non-stabilized versions of the local- and global-adaptive CM algorithms were evaluated.

The results of our experimental evaluation are shown in Figure 12. The local-adaptive algorithms outperform the global-adaptive algorithms in all tests, often by a wide margin. First, we compare the throughput of the non-stabilized CM algorithms. The performance of the non-stabilized local CM algorithm is higher than that of the non-stabilized global algorithm by between 2% – when all threads execute long transactions – and 66%, when a single thread executes short transactions and all other threads execute long transactions. In between, the performance edge of the local algorithm decreases as the number of threads executing short

[6]The data-set may be smaller if some of the transaction's operations fail.

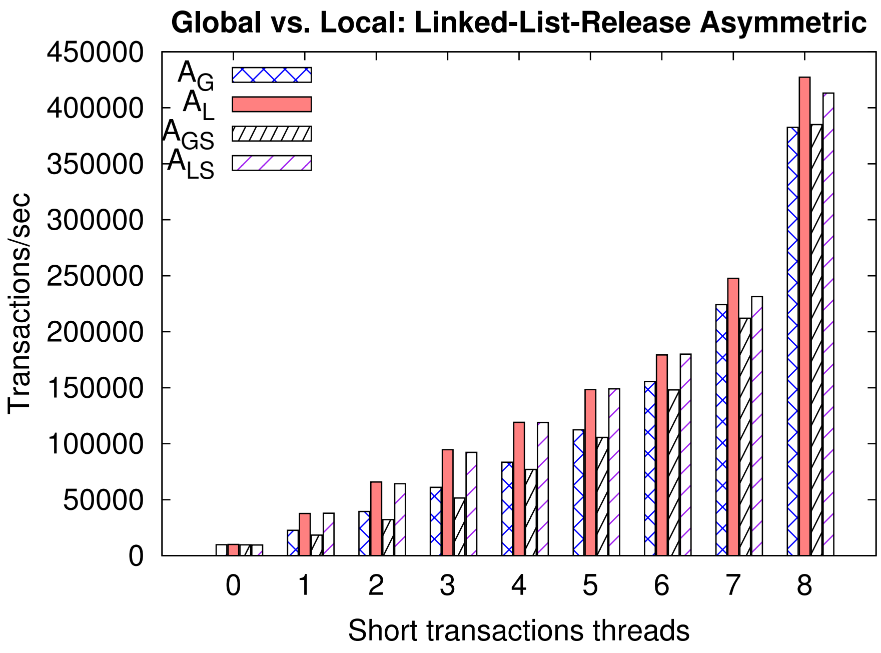

Fig. 12: AsymmetricLinkedList: Throughput as a function of the number of short transaction threads.

transactions increases from 1 to 8.

Why does the local algorithm yield significantly higher throughput than that of the global algorithm? This is because it allows different threads to concurrently apply different CM strategies. Short-transaction threads are less likely to collide, hence it is inefficient to apply serialization for these threads and, indeed, with the local-adaptive algorithm they are not serialized. When the global-adaptive algorithm is applied, however, the large number of collisions incurred by long-transaction threads changes the operation mode of *all threads* to serialization; this hurts the performance of short-transaction threads. The effect of this phenomenon diminishes when the number of short-transaction threads increases, because the probability that the mode of operation will change to serialization is reduced. When all threads perform long transactions, the advantage of the local algorithm stems from the fact that its overhead is smaller. With the local-adaptive algorithm, each thread updates its own private statistics. With the global-adaptive algorithm, on the other hand, threads need to update the same global statistics and their modifications must be synchronized.

The relative behavior of the stabilized versions of the local and global algorithms is similar. Also here, the local algorithm is superior and the maximum gap is even wider: in the case of a single short-transactions thread, the throughput of the local stabilized algorithm is twice that of global stabilized algorithm.

Asymmetry Caused by Different Data Localities. We call the second asymmetric test application we use for our evaluation *SplitArray*. Whereas the workloads-asymmetry of the *AsymmetricLinkedList* test application was caused by having different threads perform transactions of different length, the asymmetry of

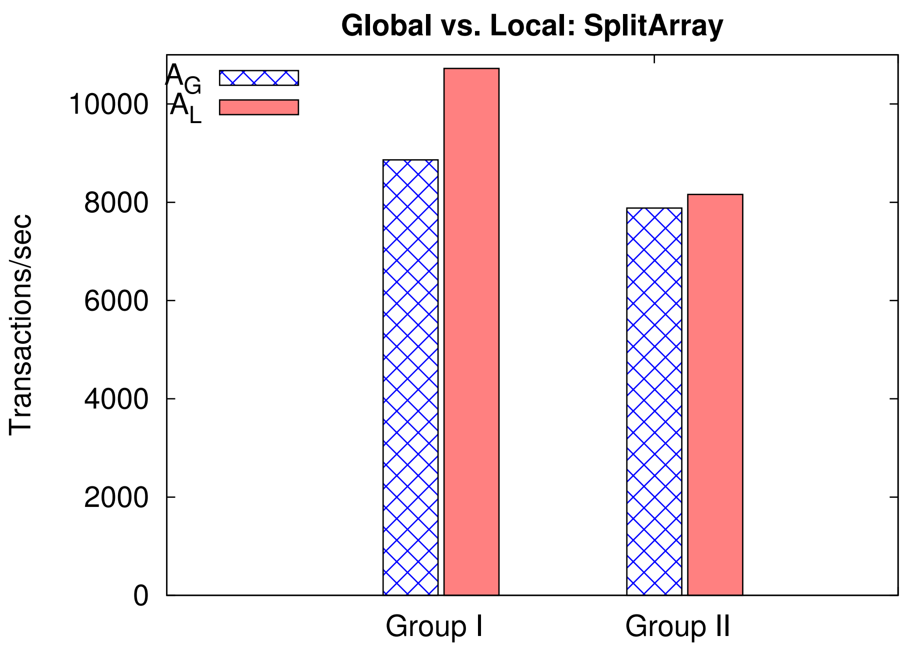

Fig. 13: AsymmetricLinkedList: Average number of committed transactions per thread.

the *SplitArray* test application is caused by having different threads access different parts of the data-structure.

The data-structure underlying *SplitArray* is an array consisting of 200 transactional objects. Our tests involve 8 threads partitioned to two groups, where group I contains 2 threads and group II contains the remaining 6 threads. The transactions executed by group I (group II) threads write to transactional objects in the first (second) half of the array. Thus, group I threads face relatively low contention whereas group II threads face significantly higher contention.

Figure 13 shows the average number of per-second committed transactions per thread in each of the two threads groups under the local and global adaptive algorithms. We first consider the performance of group I threads. The local algorithm increases the throughput of these threads by about 20% as compared with the global algorithm. In order to gain insights as to why this is happening, we measured the number of transactions that were serialized during our test; the results are shown in Table 2. With the global CM algorithm, more than 3300 transactions per group I thread were serialized on average, whereas, under the local CM algorithm the corresponding figure is almost 0. This behavior occurs because, with the local algorithm, the relatively low contention rate of group I threads almost never raises the local statistics to beyond the serialization threshold. With the global algorithm, on the other hand, statistics are shared between all 8 threads and so the serialization threshold is often surpassed, causing frequent serialization of transactions by group I threads and significantly reducing their throughput.

Analyzing the performance of group II threads, we see a mild increase of about 3.5% in throughput by local as compared with global. Looking at Table 2, we see that under the global algorithm less than 7800 transactions per thread are serial-

Table 2: Average # of serialized transactions per thread

Algorithm	Group I	Group II
Local	1.6	9444.66
Global	3322.8	7765.9

ized, whereas, under the local algorithm, more that 9400 transactions per thread are serialized. This behavior occurs because the local algorithm quickly detects the high contention levels of group II threads and so their mode of operation is set to serialization more often. The statistics of the global algorithms are somewhat influenced from the behavior of group I threads and so some of the colliding transactions of type II threads are not serialized, mildly hurting their performance.

6. LO-SER ALGORITHM CORRECTNESS

As the reader may recall from the description of the LO-SER algorithm provided in Section 2.1, the algorithm is required is to satisfy the following properties.

—Prevention of cyclic wait of aborted transactions on condition-variables.

—Prevention of race conditions which might cause an aborted transaction to wait on a winner's condition-variable after the winner finished its transaction.

—Ensuring that only threads that are involved in collisions will invoke operations on condition-variables and locks.

In this section, we establish that the algorithm meets these requirements by proving the following properties:

P1 At all times, threads waiting on condition variables (in line **33**) do not form a cycle.

P2 If a thread v waits (at line **33**) 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.

P3 Assume thread p performs a broadcast (at line **44**) 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).

The only place in the code where threads may block while waiting for another thread is line **33**. This happens when a thread activates the abort function to abort its transaction (after it loses in a collision). Thus, by proving property P1 we establish that no deadlock can occur due to a cyclic wait on condition-variables. Proving property P2 establishes that the algorithm is not subject to deadlocks caused by race conditions in which a thread waits on the condition variable of a thread that is no longer executing a transaction. Finally, property P3 establishes that only committing transactions which encountered a conflict and won it will acquire a lock and execute the broadcast operation on a condition-variable.

We assume a standard shared-memory model. An *execution* is a series of *steps*, where in each step a single thread performs a single local operation and/or a single shared-memory access. We say that a thread is *poised* to perform a step s if it will perform s when it is next scheduled. We also assume *fair executions* [5], that is, if a thread v is poised to perform step s, then eventually v is scheduled and s is performed. Our implementation uses Pthreads [7] condition variables and locks which are provided by the system. A thread can only *wait* or *broadcast* on a condition variable if it is the owner of the lock associated with it. Let c be a condition variable, m be a mutex lock, and v be a thread. When v executes statement $c.wait(m)$, then the Pthreads implementation guarantees that the following occurs atomically: (1) v is added to a queue of waiting threads associated with c, and (2) m is released.

Let v be a thread and let T_v be a transaction performed by v. We say that T_v *commits* when v exits the *COMMIT* code after performing a successful CAS operation in line **40**. Let t_1 be the time when T_v starts. Also, assume T_v commits and let $t_2 > t_1$ be the time when it commits. During the interval $[t_1, t_2]$, T_v may be aborted and resumed multiple times. We say that thread v *owns* T_v at time t if $t_1 \leq t \leq t_2$.

In the following proofs, we let v and w be threads whose owned transactions are respectively T_v and T_w. We say that v is *aborted by w at time t* if $eStat_v.status =$ *"ABORTED"* and $eStat_v.winner = w$ hold in t.[7] We further assume that T_v and T_w collide and that T_w wins this collision and aborts T_v.

We start by proving a few simple properties of the LO-SER algorithm.

LEMMA 1. *Let t be the time when v starts executing the* Abort *code. Assume v executes line* **33** *and let $t' > t$ be the time when this occurs. Then each of the following properties holds at some time during the interval $[t, t']$ (although they do not necessarily hold simultaneously):*

(1) $eStat_w.status =$ "$ACTIVE''$.

(2) The timestamp of the transaction performed by w is equal to the timestamp written to $v.eStat$ in line **18** *(if w identified the collision) or line* **20** *(if v identified the collision) of the* Collision *code.*

(3) $eStat_v.status =$ "$ABORTED''$.

PROOF. Property *(1)* holds because v executes line **33** only if the condition of line **28** is satisfied.

Process v executes line **33** only if the condition of line **30** is satisfied, so there is a time $t_2 \in [t, t']$ when w's timestamp equals $v.eStat.winnerTs$. From property *(1)* and the code, there is a time $t_1 \in [t, t'], t_1 \leq t_2$, when $eStat_w.status =$ "$ACTIVE''$. As timestamps are ever increasing, property *(2)* must hold at time t_1.

Property *(3)* holds since the ABORT code is executed by v only after $eStat_v.status$ is set to "ABORTED " and since, once set to this value, it is changed to "ACTIVE" by RSTM only after v finishes executing the *ABORT* code. □

LEMMA 2. *Let t_1 be a time when T_v is aborted by T_w and let $t_2 > t_1$ be the earliest following time when v exits the* ABORT *code (or ∞ if it doesn't). Then $eStat_v.status$ is not modified during the interval $[t_1, t_2]$* [8].

PROOF. In time t_1, $eStat_v.status =$ "$ABORTED''$ holds. There are only three pseudo-code lines where $eStat_v.status$ can be changed: line **20** where v tries to atomically CAS its own status from "ACTIVE" to "ABORTED"; line **18** where another thread tries to atomically CAS v's status from "ACTIVE" to "ABORTED"; and line **40**, where v tries to atomically CAS its status from "ACTIVE" to "COMMITTED". From the semantics of the CAS operation, any such CAS operations applied to $eStat_v.status$ during $[t_1, t_2]$ will fail. □

[7]In RSTM, once a thread detects that it was aborted, it proceeds to execute the *ABORT* code.
[8]After v exits the *ABORT* code, it modifies its status to "ACTIVE" and retries the transaction.

LEMMA 3. *Let t be a time when thread v evaluates the if condition of line* **32** *as true. Then w owns T_w at time t and did not yet execute line* **41** *since it aborted v.*

PROOF. Let t_1 be the time when v enters the *ABORT* code and let $t_2 > t_1$ be the time when v executes line **28**. From assumptions and property *(1)* of Lemma 1, w did not yet execute line **40** for T_w at time t_2 since it aborted v. When T_v is aborted, transaction T_w is owned by w and ts_w is written to $eStat_v.winnerTs$. From Lemma 2, we know that this happens before t_1 and that $eStats_v$ is not modified until v exits the *Abort* code. Since timestamps increase monotonically upon commit (line **41**) and abort (line **26**), $ts_w \geq eStats_v.winnerTs$ holds at time t_1. Let t_3 be the time when w executes line **41** while committing T_w. Assume towards a contradiction that $t_3 < t$, then $ts_w > eStats_v.winnerTs$ must hold at time t, implying that the condition of line **32** must evaluate to false. The lemma follows.

□

We now prove property P1 of the LO-SER algorithm.

LEMMA 4. *At all times, threads waiting on condition variables (line* **33***) do not form a cycle.*

PROOF. Given two threads v_i and v_j let $v_i \rightarrow v_j$ denote the fact that thread v_i waits on the condition variable of thread v_j. We will assume in contradiction that there is an execution in which a cycle is formed. Let $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_n \rightarrow v_0$ be the first cycle formed during that execution. For each $i \in \{0, \ldots, n\}$ let T_i be the transaction owned by thread v_i when the cycle is formed and let t_i be the time when T_i was aborted for the last time before the cycle was formed. Without loss of generality, assume that the cycle was created when v_0 started waiting on the condition variable of v_1. It follows that the cycle was formed when v_0 was executing the *ABORT* code. From property *(3)* of Lemma 1, $eStat_{v_0}.status =$ “$ABORTED''$ holds during the execution of this code. From property *(1)* of Lemma 1, while v_0 was executing line **28** $eStats_{v_1}.status =$ “$ACTIVE''$, implying that $t_0 < t_1$. Using the same argument, while t_1 was executing line **28**, $eStats_{v_2}.status =$ “$ACTIVE''$, meaning that $t_0 < t_1 < t_2$. By applying this argument inductively, while t_n was executing line **28** $eStats_{v_0}.status =$ “$ACTIVE''$, meaning that $t_0 < t_1 < t_2 < ... < t_n < t_0$. This is a contradiction.

□

We now prove property P2 of the LO-SER algorithm.

LEMMA 5. *If thread v waits on the condition variable of thread w (line* **33***) at time t, then w owns a transaction at time t.*

PROOF. Let $t' < t$ be the time when v evaluates the condition in line **32**. Clearly from the code, v evaluates the condition as true. From Lemma 3, w owns T_w in time t' and did not yet execute line **41** in the course of T_w since it aborted v. Let $t'' < t'$ be the time when v executes line **31** and sets $release_w$ to $true$. It follows from line **42** of the *COMMIT* code and the above that, starting from time t'', w cannot commit T_w without executing lines **43**–**46**. Specifically, w cannot commit T_w without acquiring its lock in line **43**. Since w's lock is released by v only after it starts waiting on w's conditional variable (line **33**) the lemma follows. □

We now prove property P3 of the LO-SER algorithm.

LEMMA 6. *Assume thread w performs a broadcast (line* **44***) at time t_1, and let $t_2 < t_1$ denote the previous time when w performed a broadcast, or the time when the algorithm starts if no broadcast by w was performed before t_1. Then w won some collision during the interval $[t_2, t_1]$.*

PROOF. Thread w performs a broadcast at time t_1 if $release_w = true$ (line **42**) holds at that time. $release_w$ is set to *true* by some thread v that executes line **31** in the $ABORT$ code. It follows that w aborted v, hence w won a collision with thread v. Let t_3 be the time when thread v lost such a collision to w (and was eventually aborted by it). We observe that, in line **45**, $release_w$ is set by w to $false$; clearly from the code, this occurs after time t_2. It follows that v executes line **31** during interval $[t_2, t_1]$ and so $t_2 < t_3 < t_1$. The claim follows.

□

7. 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 proved that it is deadlock-free and that it does not perform expensive lock or condition-variable synchronization operations in the lack of contention. Using LO-SER as a building block, we implemented several adaptive and partially-adaptive CM algorithms. We then evaluated these algorithms on workloads generated by CBench, by RSTM's micro-benchmark applications, by the Swarm application and by the STAMP benchmark applications suite. We also conducted a comparison of local-adaptive and global-adaptive algorithms on asymmetric test applications, where different threads execute different workloads.

Several significant conclusions can be drawn from our work. 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 only negligible overhead for low-contention workloads. We demonstrate the importance of incorporating a stabilizing mechanism to (both local and global) adaptive algorithms, for preventing mode oscillations that hurt performance.

Our comparison of local-adaptive and global-adaptive CM algorithms establishes that, while they provide comparable performance for symmetric workloads, local-adaptive algorithms are superior for asymmetric applications, where the workloads performed by different threads are significantly different.

Recent work addresses serializing TM scheduling from various perspectives. Dragojevic et al. [11] introduced "Shrink", an STM scheduler that maintains history of past transactions in order to reduce future conflicts between transactions. A similar approach is suggested by Blake et al. [6], who introduce a dynamic contention management strategy to minimize contention by using past history to identify hot spots that may reoccur in the future and to proactively schedule affected transactions around these hot spots.

Maldonado et al. [22] investigated ways of incorporating serialization and other TM scheduling techniques inside the operating system. Attiya and Milani [4] study transactional scheduling in the context of read-dominated workloads and present the BIMODAL transaction scheduler that is tailored to improve the performance of these workloads. Ansari et al. [1] introduced serialization mechanisms for *hardware transactional memory* (HTM). Sharma et al. [28] present and analyze randomized greedy algorithms for transaction scheduling.

Acknowledgments

We would like to thank the anonymous referees for many valuable comments on a previous draft of this article, which helped improve its quality.

REFERENCES

[1] M. Ansari, B. Khan, M. Lujn, C. Kotselidis, C. Kirkham, and I. Watson. Improving performance by reducing aborts in hardware transactional memory. In *High Performance Embedded Architectures and Compilers*, pages 35–49, Berlin,Heidelberg, 2010. Springer.

[2] 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.

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

[4] H. Attiya and A. Milani. Transactional scheduling for read-dominated workloads. In *OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems*, pages 3–17, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998.

[6] G. Blake, R. G. Dreslinski, T. N. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *ISCA*, pages 302–313, 2010.

[7] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley, 1997.

[8] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *DISC*, pages 194–208, 2006.

[10] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–134, 2008.

[11] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 7–16, New York, NY, USA, 2009. ACM.

[12] P. Felber, T. Riegel, and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 237–246, Feb. 2008.

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

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

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

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

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

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

[19] 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.

[20] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.

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

[22] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller. Scheduling support for transactional memory contention management. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–90, New York, NY, USA, 2010. ACM.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] G. Sharma, B. Estrade, and C. Busch. Window-based greedy contention management for transactional memory. In *Proceedings of the 24th international conference on Distributed computing*, DISC'10, pages 64–78, Berlin, Heidelberg, 2010. Springer-Verlag.

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

[30] 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.

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

[32] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA*, pages 169–178, 2008.