

On the Performance of Window-Based Contention Managers for Transactional Memory

Gokarna Sharma and Costas Busch
Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803, USA
{gokarna, busch}@csc.lsu.edu

Abstract—This paper considers a family of greedy contention managers for transactional memory for executing *windows of transactions*, which aim to provide both good theoretical and practical performance guarantees at the same time. The main approach behind window-based contention managers is to use random delays at the beginning of the window, which have the property that the conflicting transactions are shifted inside the window and their execution times may not coincide. Thus, conflicting transactions can execute at different time slots and potentially many conflicts are avoided. In this paper, window-based contention managers are considered for eager conflict management software transactional memory systems and evaluated using sorted link list, red-black tree, skip list, and vacation benchmarks. The performance of window-based contention managers is compared through experiments with Polka, the published best contention manager, Greedy, the first contention manager with provable theoretical and practical performance properties, and Priority, a simple priority based contention manager. The results show that window-based contention managers have comparable performance with Polka, and outperform Greedy and Priority, sometimes by significant margins. The evaluation results confirm their benefits in practical performance throughput and other transactional metrics such as aborts per commit, execution time overhead, etc., along with their non-trivial provable properties. This is a significant step toward the design of scalable transactional memory schedulers.

Keywords—Transactional memory, Contention managers, Execution window, Transaction scheduling, Shared memory, Concurrency control

I. INTRODUCTION

A. Transactional Memory

The recent progress in multi-core architectures presents both an opportunity and challenge for multi-threaded softwares. The opportunity is that threads will be available to an unprecedented degree, and the challenge is that more programmers will be exposed to concurrency related synchronization problems that until now were of concern only to a selected few. Writing concurrent programs is a non-trivial task because of the complexity of ensuring proper synchronization. Due to the well known limitations of conventional lock-based synchronization (i.e., mutual exclusion), researchers considered non-blocking transactions

as a viable alternative. Herlihy and Moss [1] proposed transactional memory (TM), as an alternative implementation of mutual exclusion, which avoids many of the drawbacks of locks (i.e., deadlock, priority inversion, etc.) and seeks to reduce programming effort, while maintaining or improving execution performance. Support for the TM model on multi-core architectures has been the focus of several recent research efforts, both in hardware [2–5] and software implementations [6–12], and also in hybrid implementations [13–15].

Shavit and Touitou’s work [10] is the first extension of the TM idea to software transactional memory (STM), where they present a novel software method for supporting flexible transactional programming of synchronization operations. The advantage of STM is the convenience that it offers to the programmer in accessing shared memory resources in a wait-free manner. The fundamental module in STM is the *transaction* which represents a sequence of shared memory operations (reads and writes) that are performed *atomically* to a set of shared resources (e.g. shared memory locations). Transactions may conflict when they access the same shared resources. If a transaction T finds that it conflicts with another transaction T' (because they share a common resource), it has two choices, it can give T' a chance to finish and commit by aborting itself, or it can proceed and commit by forcing T' to abort; the aborted transaction then retries again until it eventually commits. The aborted transactions waste computing resources, energy, and reduce the overall performance of the system, sometimes drastically. Ideal execution of concurrent transactions should order them to execute in such a way that it would minimize the number of aborts, but such an ordering may be difficult to obtain because transactions usually act on dynamic data and the conflicts are produced dynamically with no a priori knowledge not even of the data items to be accessed.

In the heart of any STM system is the *contention manager* which handles the transaction conflicts and schedules appropriately the transactions. Dynamic STM (DSTM) [6], proposed for dynamic-sized data structures, is the first STM implementation that uses a contention manager as an independent module to resolve conflicts between two

transactions and ensure *progress* – some useful work is done in each time step of execution. Of particular interest are *greedy contention managers* where a transaction starts again immediately after every abort. Several (greedy and non-greedy) contention managers have been proposed in the literature [9, 16–27].

Empirical studies on most of the contention manager proposals ranging from simple exponential back-off to complex priority-based techniques have shown that the choice of a contention manager can significantly affect the performance of the STM systems, sometimes drastically [6, 7, 17, 20]. Most of the contention managers available in the literature have been assessed only experimentally (e.g., [9, 20–22, 24, 25, 28–30]) by specific benchmarks (simple benchmarks such as link list [6], red-black tree [6], and skip list [31] and more complex benchmarks such as STAMP [32] and STMBench7 [33]). There is a small amount of work in the literature which analyzes formally the theoretical performance of contention managers [16–19, 21, 23, 26, 27, 34] (details in Section I-D).

As the efficiency of the STM systems relies on the good performance of the contention managers [6, 7, 17, 20], it is of great importance to design contention managers which scale gracefully with the size and complexity of the distributed system (i.e. when the number of cores in a multiprocessor chip increases). The natural question is to design such scalable contention managers which have provable non-trivial good formal performance and promising empirical observable attributes. Formal analysis is important to provide worst-case performance guarantees, and reasoning about their formal correctness and progress properties, while empirical performance determines how efficient and scalable is the contention manager in various practical purposes. These attributes combined are essential toward the design and development of scalable transactional memory systems with guaranteed formal and practical performance which is sustained even when the system size and complexity scales.

B. Window-Based Execution of Transactions

A major challenge in guaranteeing scalable progress through transactional contention managers is to devise a policy which ensures that all transactions commit in the shortest possible time. The main goal is to minimize the *makespan* which is defined as the duration from the start of the schedule, i.e., the time when the first transaction is issued, until all transactions commit. In a dynamic scenario, the makespan translates to the *throughput*, measured as the ratio of committed transactions in unit of time. The makespan of the transactional scheduling algorithm can be compared to the makespan of an optimal off-line scheduling algorithm (which has the complete knowledge of the resource requests along with the arrival and execution time durations of transactions) to provide a competitive ratio. The makespan and competitive ratio primarily depend on the

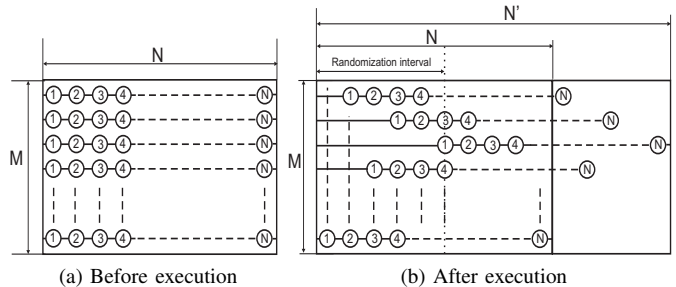


Figure 1: Execution window model for transactional memory

workload – the set of transactions, along with their arrival times, execution time duration, and resources they read and modify [26].

The competitive ratio results are not encouraging. For example, in [16] the authors give an $O(s)$ competitive ratio bound, where s is the number of resources. When the number of resources s increases, the performance degrades linearly. A difficulty in obtaining tight bounds is that the algorithms studied in [9, 16–19, 21, 23, 26] apply to the *one-shot scheduling problem*, where each thread issues a single transaction. One-shot problems are directly related with vertex coloring, where the problem of determining the chromatic number of a graph is reduced to finding an optimal time schedule for the one-shot problem. Since it is known that computing an optimal coloring given complete knowledge of the graph is a very hard problem to approximate, the one-shot problem is very hard to approximate too [35].

In order to obtain non-trivial provable properties along with the promising empirical performance, we consider, in [36], the performance of program executions in *windows of transactions* (see Fig. 1a), which has the potential to overcome some of the limitations of the coloring reduction in certain circumstances. A $M \times N$ window W consists of M threads with an execution sequence of N different transactions per thread. The execution window W can be viewed as a collection of N one-shot transaction sets with M concurrent transactions in each set.

In [36], we show that we can obtain better results by utilizing the *window* representation. We present a family of window-based greedy contention managers (details in Section II) where transactions are assigned priorities values, such that for some random initial interval in the beginning of the window each transaction is in low priority mode and then after the random period expires the transactions switch to high priority mode. In high priority mode a transaction can only be aborted by other high priority transactions. The random initial delays have the property that the conflicting transactions are shifted inside the window and their execution times may not coincide (see Fig. 1b). The benefit is that conflicting transactions can execute at different time slots and potentially many conflicts are avoided. The benefits

become more apparent in scenarios where the conflicts are more frequent inside the same column transactions and less frequent between different column transactions.

We also analyze the formal performance of window-based contention managers in [36]. Assuming that each transaction has duration τ and conflicts with at most C other transactions inside the window, we prove that the offline algorithm produces a schedule of length $O(\tau \cdot (C + N \cdot \log(MN)))$, with high probability. We prove that the online algorithm produces a schedule of length $O(\tau \cdot (C \cdot \log(MN) + N \cdot \log^2(MN)))$, with high probability. Both of the algorithms exhibited competitive ratio very close to $O(s)$, where s is the number of shared resources. We also give the adaptive version of the previous algorithms which achieves similar worst-case performance but adaptively guesses the value of the contention measure C .

C. Contributions

The main contribution of this paper is to analyze the practical performance of window-based algorithm variants given in [36] and demonstrate that efficient contention management is crucial for the good performance of the STM systems. The effective and scalable contention managers should exhibit both good theoretical and empirical performance guarantees to understand better the nature of the problem, where theoretical and practical issues are intertwined.

Window-based contention managers are implemented in DSTM2 [7], an eager conflict management STM implementation¹ that has been modified to employ the random initial delays and frame-based approach to execute transactions. They are evaluated using four widely used benchmarks: sorted link list [6], red-black tree [6], skip list [31], and vacation benchmark from STAMP suite [32].

We present some initial evaluation results showing that window-based contention managers have a very reasonable performance throughput in different TM benchmarks comparing to the other contention managers experimented so far in practice. The performance comparison of window-based contention manager variants is done through experiments using following three widely known contention managers available in the literature: (i) *Polka* [20], the published best contention manager (although it has no provable properties); (ii) *Greedy* [17], the first contention manager with provable theoretical and practical performance properties for one-shot scheduling problem; and (iii) *Priority* [20], a simple priority based contention manager. The final conclusion from the evaluation results is that window-based contention managers achieve comparable performance with *Polka*, and outperform *Greedy* and *Priority* in most of the benchmarks used in the experiments, sometimes by significant margins. The evaluation results confirm the benefits of window-based

contention managers in practical performance throughput and other transactional metrics such as aborts per commit, execution time overhead, etc., along with their non-trivial provable properties. This is a very significant step toward designing transactional memory schedulers that cope up with the size and complexity of the number of cores increases in the multi-core architectures.

D. Related Work

In 2003, Herlihy, Luchangco, Moir, and Scherer III [6] proposed Dynamic STM (DSTM) for dynamic-sized data structures. They gave experimental results in DSTM using *Polite*, *Aggressive*, and *Simple Locking* contention management mechanisms on *IntSetSimple*, *IntSetRelease*, and red-black tree benchmarks, and conclude that choice of a contention management algorithm can significantly affect the transaction throughput and some contention manager which has good performance at some benchmark may not achieve the same performance result at other benchmarks. Scherer III and Scott, in [20], proposed and analyzed different contention management policies considering visible and invisible versions of read accesses, and different benchmarks that vary in complexity, level of contention, and mix of reads and writes. Their analysis of throughput results reveals that choice of a contention manager is crucial for the performance throughput in different benchmarks. They concluded that *Polka* generally gives good performance in most of the benchmarks even though it has no provable properties.

The first step towards developing contention managers which exhibit good practical performance along with the non-trivial provable properties is due to Guerraoui et al. [17]. The *Greedy* contention manager proposed by them is the first contention manager which decides in favor of old transactions using timestamps and achieves $O(s^2)$ competitive ratio in comparison to the optimal off-line schedulers for M concurrent transactions that share s resources, and has promising empirical performance at the same time. They experimented *Greedy* contention manager in DSTM using link list and red-black tree benchmarks and concluded that it achieves similar performance in comparison to other contention managers like *Polka* and *Aggressive* along with its provable properties. Attiya et al. [16] improved the competitive ratio of *Greedy* to $O(s)$ which is a significant improvement over the previous theoretical result but no experimental results have been given. While previous studies showed that *Polka* [20] and *SizeMatters* [22] exhibit good overall performance for variety of benchmarks, Schneider and Wattenhofer's work [19] showed that they may perform exponentially worse than their *RandomizedRounds* algorithm from a worst-case perspective.

Other recent works are *Serializer* [9], *Shrink* [21], *Adaptive Transaction Scheduling (ATS)* [25], *Steal-On-Abort* [24], and *Bimodal* [26] schedulers. Among them, the *Bimodal* is proven to be $O(s)$ -competitive while rest are at

¹In an eager conflict management implementation, a contention manager is called right away when a transaction discovers a conflict.

least $O(M)$ -competitive in the worst case. One very recent work in the direction similar to bimodal workload model is the balanced workload model² studied by Sharma and Busch [27]. Their first algorithm *Clairvoyant* is $O(\sqrt{s})$ -competitive and second algorithm *Non-Clairvoyant* is $O(\sqrt{s} \cdot \log M)$ -competitive. They also prove the lower bound of $\Omega(\sqrt{s})$ in the competitive ratio for any algorithm that works on balanced workload model.

E. Outline of Paper

The rest of the paper is organized as follows: In Section II, we discuss window-based contention managers and their provable properties. We present evaluation results in Section III. Section IV concludes the paper with some discussions.

II. WINDOW-BASED CONTENTION MANAGEMENT

A. Execution Window Model and Definitions

We consider M threads $\mathcal{P} = \{P_1, \dots, P_M\}$ and a model that is based on a $M \times N$ execution window W consisting of a set of transactions $\mathcal{T}(W) = \{(T_{11}, \dots, T_{1N}), (T_{21}, \dots, T_{2N}), \dots, (T_{M1}, \dots, T_{MN})\}$, where each thread P_i issues N transactions T_{i1}, \dots, T_{iN} in sequence, so that T_{ij} is issued as soon as $T_{i(j-1)}$ has committed. If $N = 1$ then this is similar to the one-shot TM model, that uses one transaction per thread. Each transaction T_{ij} has execution time duration $\tau > 0$. Transactions share a set of s shared resources $\mathcal{R} = \{R_1, \dots, R_s\}$. Each transaction is a sequence of actions that is either a read or write operation to some shared resource R . Concurrent write-write actions or read-write actions to shared objects by two or more transactions cause conflicts between them. If a transactions conflicts then it either aborts, or it may commit and force to abort all other conflicting transactions. In greedy contention management algorithms, if a transaction aborts it then immediately restarts and attempts to commit again.

Let $\mathcal{R}(T_i)$ denote the set of resources used by transaction T_i . We can write $\mathcal{R}(T_i) = \mathcal{R}_w(T_i) \cup \mathcal{R}_r(T_i)$, where $\mathcal{R}_w(T_i)$ are the resources which are to be written by T_i and $\mathcal{R}_r(T_i)$ are the resources to be read by T_i . Two transactions T_i and T_j *conflict* if at least one of them writes on a common resource, that is, there is a resource R such that $R \in (\mathcal{R}_w(T_i) \cap \mathcal{R}(T_j)) \cup (\mathcal{R}(T_i) \cap \mathcal{R}_w(T_j))$. In other words, we can say that R causes the conflict. The *conflict graph* $G = (\mathcal{T}, E)$ has nodes the transactions, and $(T_i, T_j) \in E$ for any two transactions T_i, T_j that conflict. The conflict graph can be used to obtain a simple greedy schedule of transactions by computing the vertex coloring of it and committing all transactions of same color simultaneously.

B. Window-Based Contention Management Algorithms

We present and analyze, in [36], a family of window-based contention management algorithms. We briefly describe them here before presenting the experimental eval-

²In balanced workloads, if a transaction is writing, the number of write operations it performs is a constant fraction of its total reads and writes.

uation in Section III. We assume that parameters M and N are known to the algorithms and also each thread P_i knows C_i , which denotes the maximum number of transactions that any transaction in P_i conflicts with; namely the maximum degree of a node in conflict graph, and $C = \max_i C_i$. We divide time steps into *frames* which are the time period of durations depend on the nature of the algorithm. Each thread P_i is assigned an initial random time period consisting of q_i frames, where q_i is chosen randomly from the range $[0, \alpha_i - 1]$, $\alpha_i = C_i / \ln(MN)$. Each transaction has two priorities: *low* or *high*. Transaction T_{ij} is initially in low priority. Transaction T_{ij} switches to high priority in the first time step of frame $F_{ij} = q_i + (j - 1)$ (this is the assigned frame for T_{ij}) and remains in high priority thereafter until it commits.

1) *Offline Algorithm*: This algorithm is a basic greedy algorithm which uses the conflict graph explicitly to resolve conflicts of transactions. It is called offline because of its use of dynamic conflict graph explicitly to resolve the conflicts between transactions at each step of execution. Each transaction's priorities are used to resolve conflicts. A high priority transaction may only be aborted by another high priority transaction. A low priority transaction is always aborted if it conflicts with a high priority transaction. In this algorithm the frames are of size $\Theta(\ln(MN))$ time steps.

The intuition behind the algorithm is as follows: consider a thread P_i and its first transaction in the window T_{i1} . According to the algorithm, T_{i1} becomes high priority in the beginning of frame F_{i1} . Because q_i is chosen at random among $C_i / \ln(MN)$ positions it is expected that T_{i1} will conflict with at most $O(\ln(MN))$ transactions in its assigned frame F_{i1} which become simultaneously high priority in F_{i1} . Since a time frame contains $\Phi = \Theta(\ln(MN))$ time steps, transaction T_{i1} and all its high priority conflicting transactions will be able to commit by the end of time frame F_{i1} , using the conflict graph. The initial randomization period of $q_i \cdot \Phi$ frames will have the same effect to the remaining transactions of the thread P_i , which will also commit within their assigned frames.

2) *Online Algorithm*: This algorithm is online in the sense that it does not depend on knowing the conflict graph to resolve conflicts. This algorithm is similar to *Offline* with the difference that for the conflict resolution it uses as a subroutine a variation of *RandomizedRounds* [19]. The frames are of size $\Theta(\ln^2(MN))$ time steps and two different priorities are associated with each transaction under this algorithm. The pair of priorities for a transaction is given as a vector $\langle \pi^{(1)}, \pi^{(2)} \rangle$, where $\pi^{(1)}$ represents the Boolean priority value *low* or *high* (with respective values 1 and 0) as described in *Offline* algorithm, and $\pi^{(2)} \in [1, M]$ represents the random priorities used in *RandomizedRounds* algorithm. The conflicts are resolved in lexicographic order based on the priority vectors, so that vectors with lower lexicographic order have higher priority.

When a transaction T is issued, it starts to execute

immediately in low priority ($\pi^{(1)} = 1$) until the respective randomly chosen time frame F starts where it switches to high priority ($\pi^{(1)} = 0$). Once in high priority, the field $\pi^{(2)}$ will be used to resolve conflicts with other high priority transactions. A transaction chooses a discrete number $\pi^{(2)}$ uniformly at random in the interval $[1, M]$ on start of the frame F_{ij} , and after every abort. In case of a conflict of T with another high priority transaction K which has higher $\pi^{(2)}$ value than T , then T proceeds and K aborts. The procedure $abort(T, K)$ aborts transaction K .

3) *Adaptive Algorithm*: The algorithms Offline and Online have a limitation that they need to know C_i . This algorithm removes this limitation by allowing each thread to guess the individual values of C_i . The simple adaptive algorithm can start by allowing each thread P_i to start with assuming $C_i = 1$. Based on the current estimate C_i , the thread attempts to execute Online algorithm, for each of its transactions inside the window. Now, if the choice of C_i is correct then each transaction of the thread in the window W of the thread P_i should commit by the end of the respective assigned frame that it becomes high priority. Thus, all transactions of thread P_i should commit within the time estimate of Online algorithm. However, if thread P_i is unable to commit one of its transactions within its assigned frame (we call this a *bad event*), then thread P_i will assume that the choice of C_i is incorrect, and will start over again with the remaining transactions assuming $C_i' = 2 \cdot C_i$. Eventually thread P_i will guess the right value of C_i for the window W , and all its transactions will commit within their respective time frames. It is easy to see that the correct choice of C_i will be reached by a thread P_i within $\log C_i$ iterations. We discuss other variants of Adaptive used in experiments in Section III-A.

C. Formal Performance Properties of the algorithms

The contention measure C within the window is proposed to allow more precise statements about the worst-case complexity bound of any contention management algorithm. Based on C we prove, in [36], the following theorems for the first algorithm Offline:

Theorem 2.1 (Makespan of Offline): Algorithm Offline produces a schedule of length $O(\tau \cdot (C + N \cdot \log(MN)))$ with probability at least $1 - (MN)^{-1}$, where τ is the execution time duration of any transaction inside the window.

Theorem 2.2 (Competitive Ratio of Offline): The makespan of the schedule produced by Algorithm Offline has competitive ratio $O(s + \log(MN))$ with probability at least $1 - (MN)^{-1}$ for any choice of C , where s is the number of shared resources.

For the second algorithm Online we prove the following theorems in [36]:

Theorem 2.3 (Makespan of Online): Algorithm Online produces a schedule of length $O(\tau \cdot (C \cdot \log(MN) + N \cdot \log^2(MN)))$ with probability at least $1 - 2 \cdot (MN)^{-1}$.

Theorem 2.4 (Competitive Ratio of Online): The makespan of the schedule produced by Algorithm Online has competitive ratio $O(s \cdot \log(MN) + \log^2(MN))$ with probability at least $1 - 2 \cdot (MN)^{-1}$ for any choice of C .

The makespan produced by Online is only a factor of $O(\log(MN))$ worse in comparison to Offline but the benefit is that it does not need to know the conflict graph of the transactions to resolve the conflicts and resolves the conflicts based on random priorities. The third algorithm Adaptive is the adaptive version of the previous algorithms which achieves similar worst-case performance and adaptively guesses the value of the contention measure C .

An advantage of our algorithms is that if the conflicts inside the window are bounded by $C \leq N \cdot \log(MN)$ then the upper bounds we have obtained are within poly-logarithmic factors from optimal, since N is a trivial lower bound in execution time. This is an improvement over the trivial approach of using N one-shot executions.

III. EXPERIMENTAL EVALUATION

The experimental evaluation aims to investigate the performance benefits of the window-based contention management algorithms by executing several benchmarks using different contention configurations (ranging from low contention to high contention). The platform used to execute benchmarks is an Intel i7 (4-core) Processor 2.66 GHz system with 8GB RAM, running Windows 7 Enterprise, and using Java 1.6.0_14. We perform our experiments in DSTM2 [7], an eager conflict management STM implementation, using the default shadow factory and visible reads. DSTM2, like other STMs (e.g., TL2 [8], RSTM [37], TinySTM [38]), creates a number of threads that concurrently execute transactions. Experiments are executed with $M = 1, 2, 4, 8, 16$, and 32 threads, $N = 50$, and $\alpha_i = C_i / \log(MN)$ at most N . We run the experiments for 10 seconds and the data plotted are the average of 6 experiments.

The benchmarks used to evaluate our window-based algorithms are link list [6], red-black tree [6], skip list [31], and vacation benchmark from the STAMP suite [32]. Hereafter, we refer them as List, RBTree, SkipList, and Vacation, respectively for clarity and conciseness. The benchmarks are configured to generate large amounts of transactional conflicts (i.e., high contention scenarios) that facilitate us to evaluate the algorithms we proposed in this paper. The evaluation of our algorithms has been done by changing the amounts of transactional conflicts from low contention scenario to high contention scenario on the benchmarks. Here we briefly describe each benchmark used in the experiments.

The List and RBTree benchmarks transactionally insert and remove random numbers into a sorted linked list and a tree, respectively. The SkipList is a benchmark that stores a sorted list of items, using a hierarchy of linked lists that connect increasingly sparse subsequences of the items. The insertion and removal of an item in the SkipList is also done

transactionally. List, RBTree, and SkipList are configured to perform randomly selected insertion and deletion of transactions with equal probability. Vacation is a benchmark from the STAMP suite which simulates a travel booking database with three tables to hold bookings for flights, hotels, and cars. Each transaction simulates a customer making several bookings, and thus several modifications to the database. High contention scenario is achieved by configuring Vacation to execute many transactions which perform large number of modifications to the travel booking database.

A. Algorithm Variants Used in Experiments

We now briefly describe the window-based algorithm variants used in the experiments (see Fig. 2 for their performance throughput). We did not use Offline algorithm in evaluation because it resolves conflicts based on the conflict graph, which requires global knowledge.

- **Online:** is the same algorithm described in Section II.
- **Online-Dynamic:** is the improved version of Online algorithm where frames are dynamically contracted or expanded based on the amount of contention inside the frame (see Section III-B for details). Both of the algorithms Online and Online-Dynamic assume the contention measure C_i for each thread P_i to determine the random initial delay at the beginning of the window.
- **Adaptive:** is same as the one described in Section II.
- **Adaptive-Improved:** is the variant of Adaptive algorithm where the new contention measure value C'_i is calculated based on the contention intensity (CI) calculation similar to [25].
- **Adaptive-Improved-Dynamic:** is the variant of Adaptive-Improved where frames are dynamically contracted or expanded similar to Online-Dynamic.

The performance of our window-based algorithms is compared through experiments with the following contention managers (see Fig. 3). We briefly describe them here (see [17, 20] for the detailed description):

- **Polka:** combines Karma and Backoff by giving the enemy transaction exponentially increasing amounts of time to commit, for a number of iterations equal to the difference in the transactions' priorities, before aborting the enemy transaction [20].
- **Greedy:** aborts the younger transaction between the two conflicting transactions based on static timestamps, unless the older transaction is suspended or waiting [17].
- **Priority:** is a static priority-based manager, where the priority of a transaction is its start time, that aborts lower priority transactions during conflicts [20].

B. Throughput Results

The throughput results of these algorithms in List, RBTree, SkipList, and Vacation benchmarks are given in Fig. 2.

Performance variance is generally minimal between the two best performing strategies Online-Dynamic and Adaptive-Improved-Dynamic.

Window-based contention managers use the randomized time period at the beginning of the window and the frames of fixed sizes for the transactions which are in high priority at the same frame to execute and commit. Due to the randomized interval, the probability of conflict among transactions that are in high priority at particular frame is very low. As a result, they may finish execution and commit sufficiently before the end of the frame. The remaining time between the last transaction in the frame commit and the end of the frame is wasted in the algorithms described in Section II. To utilize that time we devised Online-Dynamic and Adaptive-Improved-Dynamic which work based on dynamic contraction of the frames, i.e., as soon as last transaction inside a particular frame finishes, we start the new frame. This helps in reducing the overhead imposed by random delay in the beginning and the size of the frames. As shown in Fig. 2, dynamic variants always perform better in comparison to their static variants Online and Adaptive-Improved. Similarly, if all the transactions inside a particular frame do not commit until the end of the frame, we can expand the frame till all the transactions commit. The basic expansion of the frame can be obtained by adding an extra frame. As window-based contention managers obey *pending commit* property [17], even if all the transactions conflicts with each other and they need to be serialized, all transactions, with very high probability, finish by the end of the frame. Thus, frame expansion is generally not needed.

The throughput comparison of our algorithms with Polka, Greedy, and Priority in List, RBTree, SkipList, and Vacation benchmarks is given in Fig. 3. We compare the performance of window-based algorithms with Polka because it is the published best contention manager for most of the transactional memory benchmarks (although it has no provable theoretical bounds). Similarly, we compare with Greedy because it is the first contention manager that exhibits non-trivial provable worst-case guarantees along with good empirical performance. We are specially interested to the comparison results with Greedy because of its both theoretical and practical performances. Priority is the simplest contention manager for comparison which decides to abort the transaction based on priority comparison. Curious observation of initial experimental results shows that our contention managers always improve throughput over Greedy in List, RBTree, and Vacation (see top left, top right, and bottom right of Fig. 3), sometimes by significant margins. The performance improvement is generally 2–4 fold in List, 2–3 fold in RBTree, and 2 fold in Vacation than Greedy. The throughput results are comparable to Polka in all the benchmarks, except Vacation where window variants outperform by 1.5–2×. In SkipList, the performance is little

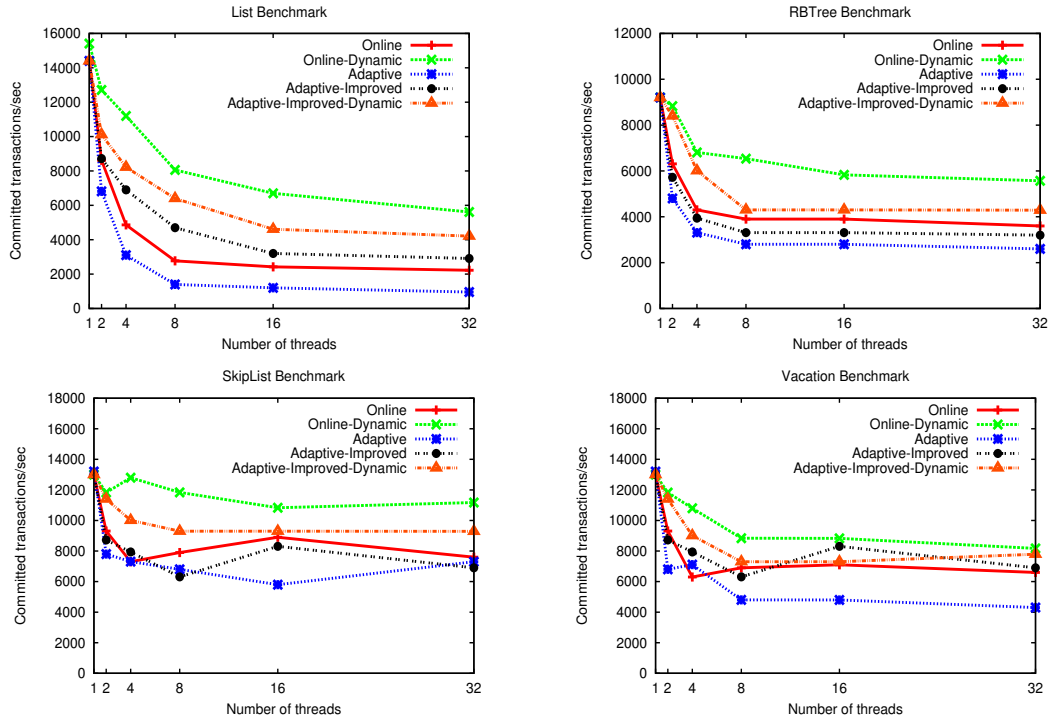


Figure 2: Performance throughput results comparison of different window-based algorithm variants in List, RBTree, SkipList, and Vacation benchmarks

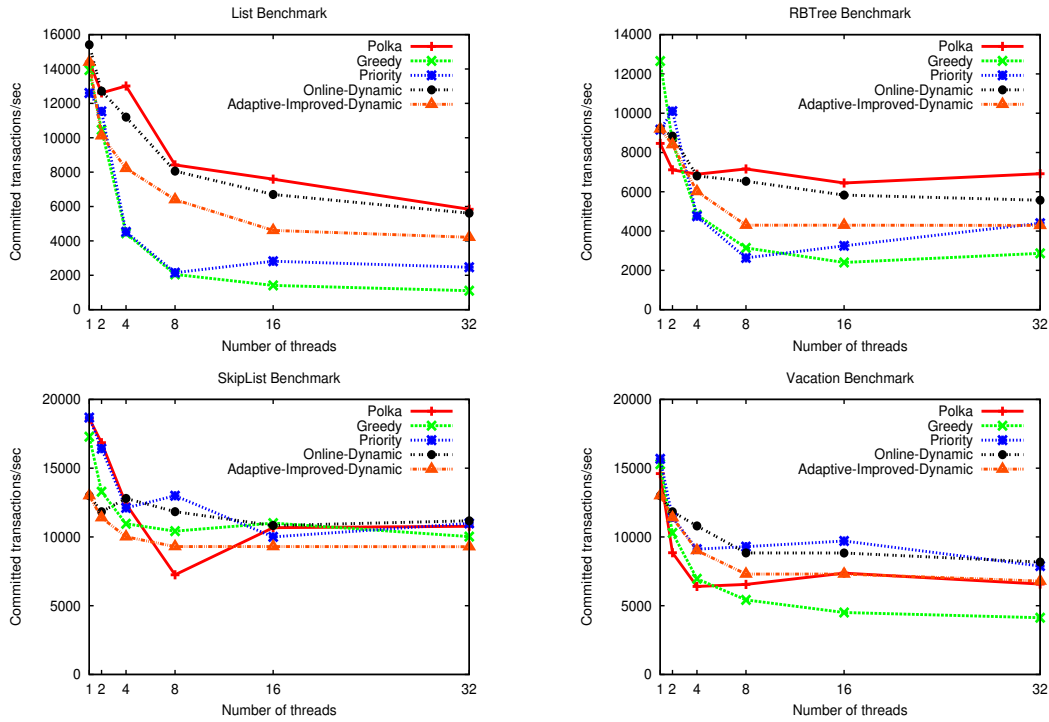


Figure 3: Performance throughput results comparison between best performing window algorithms and Polka, Greedy, and Priority contention managers in List, RBTree, SkipList, and Vacation benchmarks

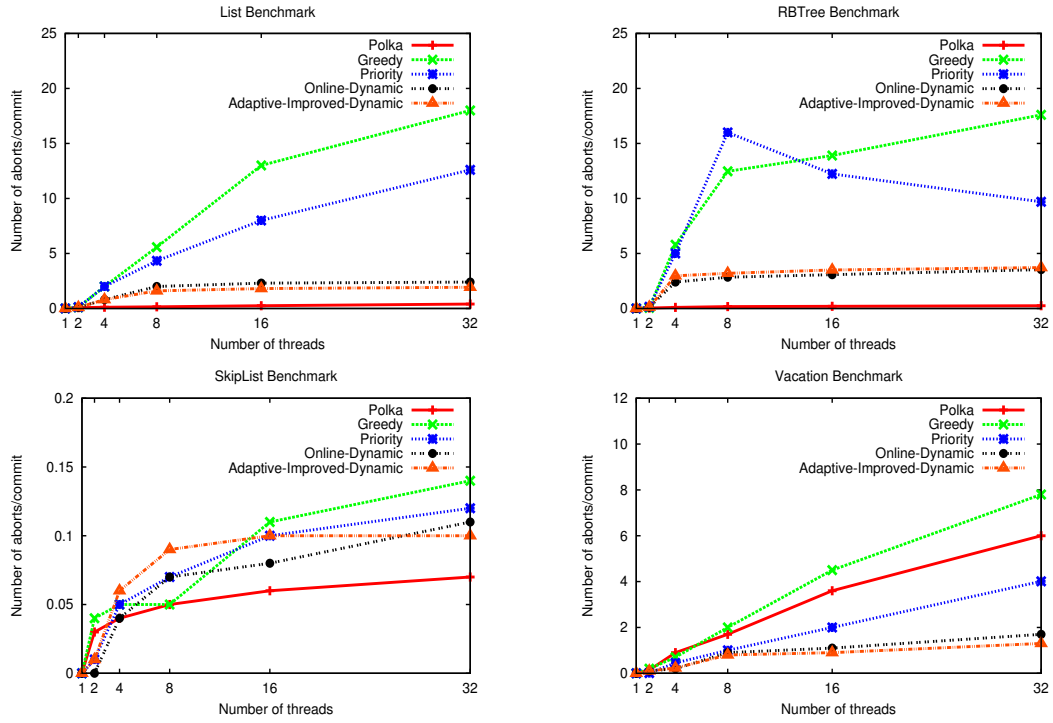


Figure 4: Aborts per commit results comparison between window-based algorithm variants and Polka, Greedy, and Priority contention managers in List, RBTREE, SkipList, and Vacation benchmarks

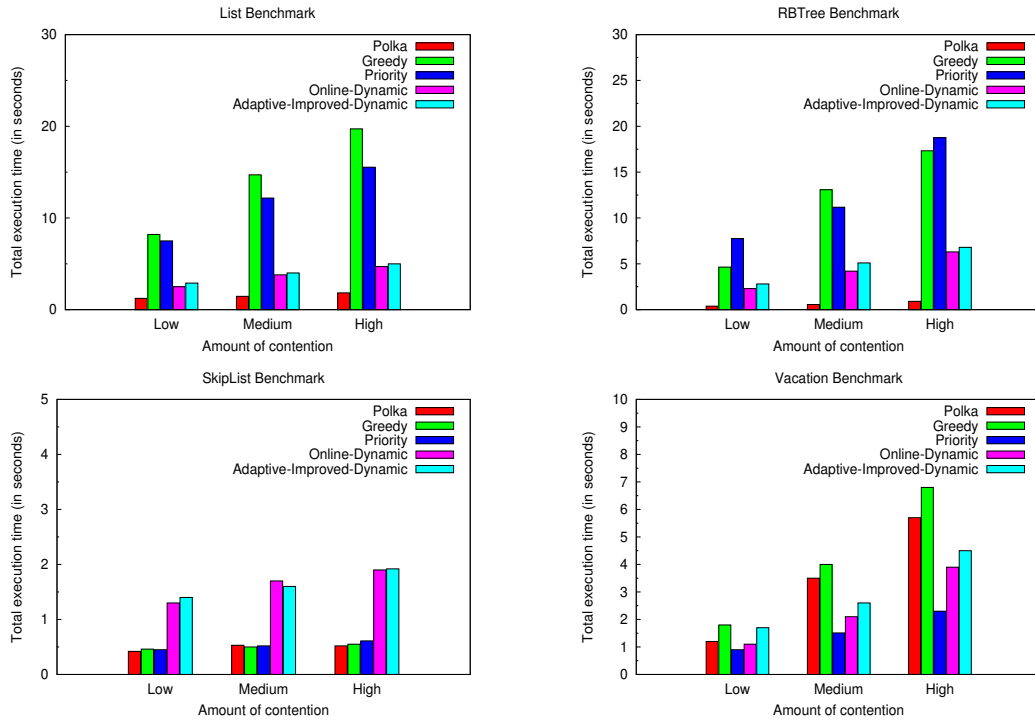


Figure 5: Comparison of total time needed by window-based algorithm variants and Polka, Greedy, and Priority contention managers to commit 20000 transactions in List, RBTREE, SkipList, and Vacation benchmarks using 32 threads

bit worse but comparable (see bottom left of Fig. 3).

C. Aborts per Commit Results

Aborts per commit is the ratio of number of aborts to the number of commits of transactions. It is another metric used to measure the efficiency of the contention manager in utilizing the computing resources. The higher aborts per commit ratio signifies the waste of computing resources due to the aborted transactions. Fig. 4 shows aborts per commit results. The results indicate that best performing window-based variants reduce the number of aborts per commit in List, RBTree, and Vacation significantly in comparison to Greedy and Priority (2–10× less) and the number of aborts are comparable to Polka (only 1–3× more) in all benchmarks except Vacation where window-based variants outperform by 2–4×. The aborts per commit results are comparable for all strategies due to low conflict probability in SkipList in comparison to other benchmarks.

D. Execution Window Overhead Results

We measure the overhead of window model by allowing the window-based algorithms to execute 20000 randomly generated transactions in each benchmarks and measure the total time needed to commit all of them. We perform the experiments using 32 threads in three different contention scenarios (i.e., amount of contention): (i) *Low contention* – each transaction needs to perform only 20% update operations; (ii) *Medium contention* – each transaction needs to perform 60% update operations, hence medium amount of contention; and (iii) *High contention* – each transaction needs to do 100% update operations, hence high contention. That is, increasing percentage of update operations increase the probability of contention among transactions.

Fig. 5 shows the results for the total time needed for different contention management strategies (window-based and others) to commit 20000 randomly generated transactions on List, RBTree, SkipList, and Vacation benchmarks under different amounts of contention using 32 threads. Our best performing algorithms (Online-Dynamic and Adaptive-Improved-Dynamic) always need less time than Greedy and Priority in List and RBTree (see top left and top right of Fig. 5). The time needed in List and RBTree by Online-Dynamic is 100–228% less than Greedy and 175–200% less than Priority in low contention and only 2–3× more than Polka. The similar kind of performance result can also be seen in medium and high contention configuration of List and RBTree benchmarks. In the SkipList (see bottom left of Fig. 5), the overhead is high (2–3× worse) in our algorithm due to initial randomization period and time needed for adaptive guessing of contention (not from the time needed to execute transactions) which is not needed in other contention managers. If we do not consider the aforementioned overheads, window-based algorithms achieve similar time

performance. In Vacation, window-based variants outperform Polka and Greedy but give comparable performance to Priority. In summary, in low-contention scenario, the overhead is visible like in SkipList, but in high-contention scenario, the overhead due to randomization is negligible like in List, RBTree, and Vacation; hence the benefits can be achieved using window-based contention managers.

IV. CONCLUSION

We considered greedy contention managers for transactional memory for $M \times N$ windows of transactions with M threads and N transactions per thread and present and evaluate their variants for contention management in transactional memory using List, RBTree, SkipList, and Vacation benchmarks on a widely used eager conflict management STM implementation called DSTM2. These algorithms are efficient, adaptive, and improve on the worst-case performance of previous results and are the first such results for the execution of sequences of transactions instead of the one-shot scheduling problem considered in the literature. The initial evaluation results confirm the benefits of window-based algorithms in practical performance throughput and other transactional metrics such as aborts per commit, execution time overhead, etc., along with their non-trivial provable properties. Window-based algorithms present new trade-offs in the design and analysis of contention managers, which achieve both good theoretical and empirical performance guarantees.

We are highly encouraged by the results from window model evaluation. In this direction, window-based algorithms can also be evaluated for other performance measures such as *wasted work*, *repeat conflicts*, *average committed transactions duration*, *average response time*, etc. Wasted work metric is the ratio which measures the proportion of execution time spent in executing aborted transactions and it is useful in measuring the cost of aborted transactions in terms of computing resources. Similarly, repeat conflicts measures the amount of time spent in executing aborted transactions. Aborts per commit, wasted work, and repeat conflicts are related and minimizing the one metric automatically improve the performance on other metric. In this sense, they complement each other. However, aborts per commit and repeat conflicts ignore the execution durations of the aborted and committed transactions. Since window model reduced the number of aborts using randomization, which in turn should have reduced the average committed transactions duration and repeat conflicts. At last, the average response time bounds the time spent by individual transaction in the system. We defer the evaluation of window model evaluation on these aforementioned performance measures for future work. We also plan to continue our evaluation in other complex benchmarks from the STAMP suite [32] (such as kmeans, bayes, genome, etc.), and STMBench7 [39].

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their careful reading and useful comments. The first author would also like to thank Jong-Hoon Kim for his very constructive help on running DSTM2 experiments.

REFERENCES

- [1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993, pp. 289–300.
- [2] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun, "Transactional coherence and consistency: Simplifying parallel hardware and software," *IEEE Micro*, vol. 24, pp. 92–103, 2004.
- [3] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *ICDCS*, 2003, pp. 522–529.
- [4] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *ASPLOS-X*. New York, NY, USA: ACM, 2002, pp. 5–17.
- [5] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, "Multiple reservations and the oklahoma update," *IEEE Parallel Distrib. Technol.*, vol. 1, pp. 58–71, November 1993.
- [6] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III, "Software transactional memory for dynamic-sized data structures," in *PODC*, 2003, pp. 92–101.
- [7] M. Herlihy, V. Luchangco, and M. Moir, "A flexible framework for implementing software transactional memory," *SIGPLAN Not.*, vol. 41, no. 10, pp. 253–262, 2006.
- [8] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *DISC*, 2006, vol. 4167, pp. 194–208.
- [9] S. Dolev, D. Hendler, and A. Suissa, "CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory," in *PODC*, 2008, pp. 125–134.
- [10] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, vol. 10, pp. 99–116, 1997.
- [11] T. Harris and K. Fraser, "Language support for lightweight transactions," *SIGPLAN Not.*, vol. 38, no. 11, pp. 388–402, 2003.
- [12] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *PPoPP*, 2005, pp. 48–60.
- [13] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *ASPLOS-XII*. New York, NY, USA: ACM, 2006, pp. 336–346.
- [14] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *PPoP*, 2006, pp. 209–220.
- [15] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An effective hybrid transactional memory system with strong isolation guarantees," in *ISCA*, 2007, pp. 69–80.
- [16] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir, "Transactional contention management as a non-clairvoyant scheduling problem," *Algorithmica*, vol. 57, no. 1, pp. 44–61, 2010.
- [17] R. Guerraoui, M. Herlihy, and B. Pochon, "Toward a theory of transactional contention managers," in *PODC*, 2005, pp. 258–264.
- [18] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon, "Robust Contention Management in Software Transactional Memory," in *SCOO*L, 2005.
- [19] J. Schneider and R. Wattenhofer, "Bounds on contention management algorithms," in *ISAAC*, 2009, vol. 5878, pp. 441–451.
- [20] W. N. Scherer, III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *PODC*, 2005, pp. 240–248.
- [21] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing versus curing: avoiding conflicts in transactional memories," in *PODC*, 2009, pp. 7–16.
- [22] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel, "Metatm/txlinux: Transactional memory for an operating system," *IEEE Micro*, vol. 28, no. 1, pp. 42–51, 2008.
- [23] D. Hasenfratz, J. Schneider, and R. Wattenhofer, "Transactional memory: How to perform load adaptation in a simple and distributed manner," in *HPCS*, 2010, pp. 163–170.
- [24] M. Ansari, M. Lujn, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, "Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering," in *HiPEAC*, 2009, vol. 5409, pp. 4–18.
- [25] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *SPAA*, 2008, pp. 169–178.
- [26] H. Attiya and A. Milani, "Transactional scheduling for read-dominated workloads," in *OPODIS*, 2009, vol. 5923, pp. 3–17.
- [27] G. Sharma and C. Busch, "A competitive analysis for balanced transactional memory workloads," in *OPODIS*, 2010, vol. 6490, pp. 348–363.
- [28] R. Guerraoui, M. Herlihy, and B. Pochon, "Polymorphic contention management," in *DISC*, 2005, vol. 3724, pp. 303–323.
- [29] M. Ansari, C. Kotselidis, M. Lujan, C. Kirkham, and I. Watson, "On the performance of contention managers for complex transactional memory benchmarks," in *ISPDC*, 2009, pp. 83–90.
- [30] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott, "A comprehensive strategy for contention management in software transactional memory," in *PPoPP*, 2009, pp. 141–150.
- [31] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, 1990.
- [32] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IEEE IISWC*, 2008, pp. 35–46.
- [33] R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: a benchmark for software transactional memory," in *EuroSys*. New York, NY, USA: ACM, 2007, pp. 315–324.
- [34] G. Sharma and C. Busch, "Improving the performance competitive ratios of transactional memory contention managers," in *WTTM 2010*, 2010.
- [35] S. Khot, "Improved inapproximability results for maxclique, chromatic number and approximate graph coloring," in *FOCS*, 2001, pp. 600–609.
- [36] G. Sharma, B. Estrade, and C. Busch, "Window-based greedy contention management for transactional memory," in *DISC*, 2010, vol. 6343, pp. 64–78.
- [37] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the overhead of software transactional memory," University of Rochester, Tech. Rep. TR 893, 2006.
- [38] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *PPoPP*, 2008, pp. 237–246.
- [39] R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: a benchmark for software transactional memory," in *EuroSys*, 2007, pp. 315–324.