Scheduling Support for Transactional Memory Contention Management

Walther Maldonado
Patrick Marlier
Pascal Felber
Univ. of Neuchâtel
Switzerland
first.last@unine.ch

Adi Suissa
Danny Hendler
Ben Gurion Univ.
Israel
adisuis@cs.bgu.ac.il
hendlerd@cs.bgu.ac.il

Alexandra Fedorova
Simon Fraser Univ.
Canada
fedorova@cs.sfu.ca

Julia L. Lawall
Univ. of Copenhagen
Denmark
julia@diku.dk

Gilles Muller
INRIA / Regal
France
gilles.muller@lip6.fr

Abstract

Transactional Memory (TM) is considered as one of the most promising paradigms for developing concurrent applications. TM has been shown to scale well on multiple cores when the data access pattern behaves "well," i.e., when few conflicts are induced. In contrast, data patterns with frequent write sharing, with long transactions, or when many threads contend for a smaller number of cores, result in numerous conflicts. Until recently, TM implementations had little control of transactional threads, which remained under the supervision of the kernel's transaction-ignorant scheduler. Conflicts are thus traditionally resolved by consulting an STM-level *contention manager*. Consequently, the contention managers of these "conventional" TM implementations suffer from a lack of precision and often fail to ensure reasonable performance in high-contention workloads.

Recently, scheduling-based TM contention-management has been proposed for increasing TM efficiency under high-contention [2, 5, 19]. However, only user-level schedulers have been considered. In this work, we propose, implement and evaluate several novel kernel-level scheduling support mechanisms for TM contention management. We also investigate different strategies for efficient communication between the kernel and the user-level TM library. To the best of our knowledge, our work is the first to investigate kernel-level support for TM contention management.

We have introduced kernel-level TM scheduling support into both the Linux and Solaris kernels. Our experimental evaluation demonstrates that lightweight kernel-level scheduling support significantly reduces the number of aborts while improving transaction throughput on various workloads.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.4.1 [*Operating Systems*]: Process Management

General Terms Algorithms, Performance

Keywords Transactional Memory, Scheduling, Contention Management

1. Introduction

Transactional memory (TM) [11, 18] has emerged as a promising alternative to lock-based programming for developing concurrent applications [1, 12]. In transactional memory, critical sections are expressed as atomic blocks performed as *transactions*. At runtime, these transactions may be executed concurrently based on the optimistic expectation that the set of locations accessed by one transaction will not overlap with the set of locations written by another concurrent transaction. If such a *conflict* does occur, a TM *contention manager* (CM) [10] decides how it should be resolved; typically, one of the conflicting transactions is aborted (the *loser*) and rolled back while the other is allowed to proceed (the *winner*). TM has been shown to scale well on multiple cores when the data access pattern behaves "well," i.e., when few conflicts are induced [1, 12]. In contrast, a data access pattern with frequent writes to shared data will induce numerous aborts. This means that a long running transaction has little chance to make progress and commit successfully unless specific—often ad-hoc—measures are taken.

Our goal is to address the problem of *repeated aborts* in software transactional memory (STM). In some cases, to avoid repeated aborts, it is sufficient to follow "good" practices, such as using short transactions in cases where the rate of conflicts is likely to be high. When this is not possible, the problem of repeated aborts is usually addressed via an application-level contention manager, which reacts when a conflict is detected, by aborting one of the conflicting transactions or by waiting for the conflict to be possibly resolved. Exponentially increasing delays can furthermore be inserted after an abort to reduce the probability of a recurrence of the conflict. These approaches, however, have been found to often provide poor performance with many workloads commonly used to evaluate STMs [15, 16]. They suffer from: (i) too many aborts, e.g., when a long running transaction conflicts with shorter transactions; (ii) lack of precision, since an aborted thread may wait too long after the commit of the conflicting transaction to restart its own transaction; and (iii) unpredictable benefits, as delaying the restart of a long transaction does not necessarily increase its chance of success, unless all other conflicting transactions have completed or are delayed even longer. These problems are particularly acute when there are more threads than cores, as can be desirable for the execution of server-type applications where threads can block in non-transactional code. In this case, a transaction that repeatedly aborts prevents other useful work from being performed on the same CPU.

The problem of conflict management is essentially a problem of controlling when transactions are executed, such that a transaction is not executed at the same time as some other conflicting transaction, and is thus essentially a *scheduling problem*. This observation has led researchers to consider various user-level scheduling policies performing some variant of *serialization*, in which the thread running a loser transaction is moved to a wait queue until the winner transaction completes [2, 5, 19]. Such approaches, however, incur a high overhead for short transactions, because they replicate at the user level kernel-level scheduling operations and abstractions and because their implementation introduces additional user-kernel context switches. Furthermore, they do not address the problem of context switches during the execution of a transaction, which increase the window of vulnerability in which a conflict can occur.

In this paper, we propose, implement and evaluate several novel user and kernel-level scheduling support mechanisms for TM contention management and analyze their relative benefits, depending on the number of threads and cores, the duration of transactions, and the degree of contention. Novel features of the algorithms considered include the use of a shared memory segment to provide lightweight communication between the user-level STM library and the kernel-level scheduler, and "soft" forms of serialization in which the loser thread is not prevented from executing, but only has its priority reduced. Finally, we propose a new contention management strategy that is based on extending the time slice of a thread running a transaction, to reduce its window of vulnerability. This strategy is orthogonal to serialization and can be combined with any of the proposed serialization algorithms. With respect to these new features, we find that the use of shared memory allows defining a serialization strategy that is efficient for short transactions with high contention, that soft serialization is beneficial for transactions that may be nondeterministic, and that time slice extension can improve scalability for some contention management strategies.

The contributions of this paper are as follows:

- We propose novel approaches for improving the performance of software transactional memory by modifying the OS scheduling policy. Our work is the first to implement and evaluate TM contention management support in the kernel.
- We study the relative efficiency of system calls and shared memory as a means of allowing the STM to interact with the kernel.
- We demonstrate the benefits of a simple yield as a contention management strategy, particularly in the presence of low contention and transactions with a duration of less than one time slice.
- We have created a reference implementation of our approach in the Linux kernel. Our modifications to the kernel are small and highly localized. We have subsequently validated the general applicability of our approach by porting and testing it on Solaris.
- We have evaluated our approach on both micro- and macro-benchmarks using a lock-based STM library. Results show that our approach is effective in situations where an application-level contention manager cannot provide satisfactory performance.

The rest of this paper is organized as follows. Section 2 further describes software transactional memory and the problem of contention management. In Section 3, we consider four approaches to serialization, and analyze their tradeoffs. In Section 4, we propose a complementary contention management strategy, based on time-slice extension, to close the window of vulnerability introduced by thread preemption at the kernel level. Section 5 evaluates these strategies on a range of benchmarks. Finally Section 6 presents related work and Section 7 concludes.

2. Background

In this section, we briefly describe the main principles of software transactional memory and highlight the difficulties that conventional contention managers have in accurately dealing with conflicts.

2.1 Software Transactional Memory

Software transactional memory (STM) [17] is a lightweight alternative to locks for synchronizing threads in concurrent applications. STM allows developers to combine sequences of concurrent operations into atomic transactions that execute optimistically and, upon conflict, automatically roll back and restart their execution. This approach promises a great reduction in the complexity of both programming and verification, by making parts of the code appear to be sequential without the need to program fine-grained locks.

There exist several types of STM designs [12], which mainly differ in their liveness properties and the granularity of conflict detection (e.g., memory word, object), as well as the various implementation choices they follow. In this paper, we use the TinySTM [8] library, which belongs to the class of *word-based* and *lock-based* STMs, which are widely considered to be among the most efficient and general purpose [4, 7]. In this class of STMs, conflict detection is performed at the level of individual machine words, and the implementation uses revokable locks to protect memory from conflicting accesses. This class also notably includes Ennals' STM [7], McRT-STM [14], and TL2 [4]. A distinguishing feature of TinySTM is that it provides by default *eager write conflict detection and invisible reads*, i.e., write-read and write-write conflicts are checked for when a memory location is accessed, while read-write conflicts are only checked for at commit time.

2.2 Contention Management

STM relies on the hypothesis that conflicts are unlikely and thus, in most cases, transactions can commit. Therefore, the ability of STM to scale directly depends on the workload. Two scenarios can negatively affect performance. First, transactions that conflict frequently will trigger many aborts, sometimes even creating a livelock situation. This is particularly the case with long running transactions. Second, when the number of threads exceeds the number of cores, threads can be preempted while executing transactions. This increases the transaction duration by one or more scheduling time slices, which is often several orders of magnitude more than the actual computation time of the transaction (see Section 5), thus drastically increasing the risk of conflicts. In both situations, transaction throughput (i.e., commit rate) will decrease.

An important challenge is to be able to handle such scenarios gracefully. This task is traditionally addressed by the contention manager (CM), which determines based on various policies whether conflicting transactions should abort, wait, or proceed. Comparative studies of contention managers [15, 16] show that different strategies work better for different benchmark applications, but no single manager performs best on all workloads. In lock-based STMs (e.g., TL2 [4] and TinySTM [8]), the typical strategy is to abort the thread that discovers the conflict. If a transaction aborts multiple times, it may wait for a random, exponentially increasing, delay before restarting, to give conflicting transactions a chance to complete. This *exponential backoff* strategy practically avoids livelocks, but often does not give good performance. Obstruction-free designs [10] usually provide more flexibility in the actions that CMs can take, by letting transactions not only adjust their own behaviors but also abort others, yet this flexibility comes with an extra cost.

To evaluate the impact of conflicts with many concurrent transactions and the effectiveness of conventional contention management, we have run a set of benchmarks from the STAMP suite [3] on a 16-core machine with two STM implementations (see Section 5 for details on our experimental setup): TL2 and TinySTM, with and without exponential backoff CM. TinySTM can either use encounter-time locking (ETL), or commit-time locking (CTL), which make a write visible to other transactions immediately or only at commit time, respectively.[1]

In the graphs in Figure 1, we observe that: (i) performance degrades significantly on most STAMP benchmarks once there are more threads than cores, except for `ssca2` that has short transactions and very few conflicts (see Section 5 and Table VI in [3]); (ii) degradation starts even *before* reaching 16 threads on two benchmarks (kmeans and intruder); (iii) degradation is generally slightly less significant using commit-time locking (TL2 and TinySTM-CTL); (iv) the backoff contention manager can be helpful (intruder) or counterproductive (genome), but it does not affect most benchmarks.

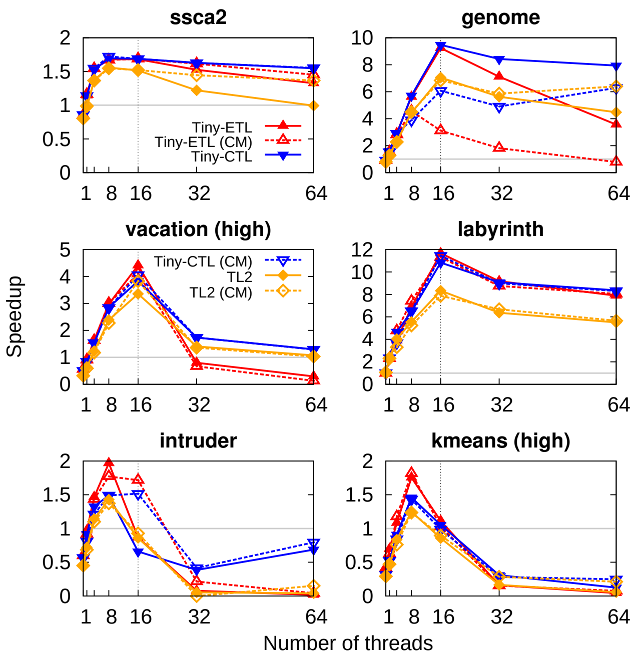

Figure 1. Performance of TinySTM (ETL and CTL) and TL2 on STAMP benchmarks (speedup relative to single-threaded execution).

There are several reasons why a conventional contention manager may be inadequate for scheduling transactions. Notably, it lacks precision when delaying the restart: a sleep of even a few microseconds is likely much longer than the duration of the winner transactions, implying that cycles are wasted. In addition, at the user level it is not possible to control when threads are preempted: if the OS scheduler preempts a thread in the middle of an active transaction, the risk of conflict with concurrent transactions increases.

3. Serialization

To address the deficiencies of conventional contention managers, attention has turned to *serialization*. Serialization is a contention-management strategy that prevents the tread running the loser transaction from executing until the winner transaction has completed. The rationale behind serialization is the following: once a pair of transactions conflict, they are likely to conflict again if allowed to execute concurrently. It follows that the execution of a loser transaction concurrently with a transaction with which it conflicted before is likely to waste CPU cycles.

Algorithm 1 shows the basic outline of an implementation of serialization. On starting a transaction, the transaction is mapped to the current thread. When a transaction detects a conflict it aborts, and is then made to wait for the completion of the winner. Finally, when a transaction commits, it releases any waiting transactions. The critical aspect of this algorithm is how to implement Wait and ReleaseAll, in particular whether waiting should be implemented entirely at the user level, or by blocking the loser thread, which requires interaction with the kernel. The optimal implementation depends on the relationship between the number of threads and the number of cores, the duration of transactions, and the level of contention. Previous work has considered purely user-level strategies. We propose strategies that involve the kernel, as well.

Algorithm 1: Serialization implementation outline

```
// Start transaction $tx$
1 upon Start($tx$)
2     $tx.thr \leftarrow$ CurrentThread()

// Conflict between $tx$ and $tx'$
3 upon Conflict($tx$, $tx'$)
4     Abort($tx$)
5     Wait($tx.thr$, $tx'.thr.wait$ )                // Serialize after winner

// Commit transaction $tx$
6 upon Commit($tx$)
7     ReleaseAll($tx.thr.wait$ )
```

User-level implementation Executing a transaction when it is known that the transaction is likely to encounter a conflict uses CPU resources unnecessarily and may provoke conflicts with other transactions. If the number of threads is less than or equal to the number of cores, however, using CPU resources unnecessarily does not affect throughput. In this case, serialization may be implemented by causing the loser transaction to wait on a spinlock that is held by the winner, thus preventing the loser from performing operations that may provoke further conflicts. This is indeed a contention management strategy that is already provided by TinySTM (`CM_DELAY`), which we refer to as Ser-u (spin).[2] When the winner transaction commits, it releases the spinlock, allowing the waiting loser transactions to restart immediately. If there are more threads than cores, however, the use of spinlocks drastically reduces throughput, as the loser transaction monopolizes the CPU for its entire time slice, without performing any useful computation.

Kernel-level implementation An alternative to having the loser thread wait on a spinlock at the user level is to block the thread, thus allowing other threads to access the CPU. This approach is indeed essential to achieving adequate parallelism when there are more threads than cores. Blocking a thread, however, requires removing it from the kernel scheduler's ready queue, which implies communication between the user and the kernel level. To achieve the full benefit of serialization, this communication must be efficient.

Traditionally, the user level communicates with the kernel via system calls. Following this strategy, which we refer to as Ser-u (cond), Wait is implemented using a system call that causes the thread to be blocked on a condition variable associated with the winner transaction, and ReleaseAll is implemented by a system call that releases all the threads blocked on the thread's condition variable. Locks are used to ensure that the winner transaction has not yet committed before the loser waits. System calls, however, have

[1] ETL is the default. CTL typically reduces the risk of conflicts during transaction execution but increases the likelihood of an abort at validation time [8].

[2] "u" refers to user-level. In the sequel, we use "k" to refer to kernel-level.

a significant overhead, which adds an extra cost to every COMMIT. This overhead may be unacceptable for short transactions, especially when there are few conflicts, in which case the system call performed by COMMIT is likely not to find any blocked threads.

To enable communication between the transaction operations and the kernel without resorting to system calls, we have developed a novel architecture based on a memory region shared between the STM library and the scheduler, as shown in Figure 2. The shared memory region contains a table of MAXTHREADS elements, each being a structure describing STM information for a given thread. We also augment the OS thread structure with a pointer to the respective entry of the aforementioned table. The initial setup and communication between the application and the kernel is done via a special device file `/dev/stm`, through which, via I/O control and memory mapping calls, the table of elements is allocated by the main application thread. Child threads subsequently request to be linked to the next available entry in the array, as part of their STM initialization process. Using this strategy, to interact with the kernel, the application simply fills in data in the shared structure, avoiding the cost of a direct user-kernel interaction.

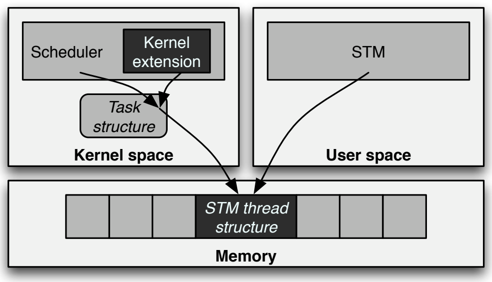

Figure 2. Layout of the interaction between the Kernel and application.

The shared memory-based implementation of serialization, SER-k, is implemented in two parts, at the STM and at the kernel level. Algorithm 2 shows the code executed by the STM at the user level that informs the kernel level that the start of a transaction, a conflict, or a commit has occurred. The WAIT and RELEASEALL actions are then shifted to the kernel scheduler, as shown in Algorithm 3. The kernel scheduler is chosen as the place to invoke this code because it is executed relatively frequently and because it can immediately react to any changes in thread state induced by the contention management code. For each thread, the scheduler first checks whether the thread has been configured to run transactions (line 11), and then whether it has committed a transaction (line 12) or detected a conflict (line 18). Because the STM-level COMMIT code does not perform any action, *e.g.,* yield, that would cause an invocation of the kernel scheduler, a thread may have both committed one or more transactions and detected a conflict during the time that elapses before the scheduler is invoked. The SCHEDULE function thus tests on line 15 that the stored conflict transaction is different from the current transaction of the committing thread, if any, so that only threads in conflict with completed transactions are returned to the ready queue. Next, if the thread is in conflict (lines 19–21), it is moved to the wait queue of the conflicting thread (line 21). This is only done if the latter thread is still in the OS's ready queue (line 20), *i.e.*, it has not concurrently detected a conflict, to avoid introducing cycles that could cause deadlock.

Using the shared memory approach, COMMIT now only performs two assignments, and no user-kernel interactions. Thus, there is essentially no overhead on COMMIT, even when there are many short transactions but few conflicts. Furthermore, because the kernel schedule function has access to all threads, it can perform all of the commit and conflict handling on only one core at a time, thus reducing the amount of locking required. Nevertheless, this approach can incur a substantial latency on awakening the blocked loser threads, as the kernel only polls the shared memory structure when the schedule function is invoked. A loser transaction can thus be unnecessarily blocked for a period of time that may be as long as a thread's time slice, which may be much longer than the duration of the transaction's computation itself.

Algorithm 2: SER-k: STM-level code

```
1   struct $stm\_thread$
2       $tx$ : transaction_id
3       $conflict\_thr$ : thread
4       $conflict\_tx$ : transaction_id
5       $commit$ : bool

    // Start transaction $tx$
6   upon START($tx$)
7       $tx.thr \leftarrow$ STMSHAREDSTRUCTPTR()
8       $tx.thr.tx \leftarrow tx$

    // Conflict between $tx$ and $tx'$
9   upon CONFLICT($tx$, $tx'$)
10      $tx.thr.conflict\_tx \leftarrow tx'$
11      $tx.thr.conflict\_thr \leftarrow tx'.thr$
12      ABORT($tx$)
13      YIELD($tx.thr$)

    // Commit transaction $tx$
14  upon COMMIT($tx$)
15      $tx.thr.tx \leftarrow$ null
16      $tx.thr.commit \leftarrow$ true
```

Algorithm 3: SER-k: Kernel-level code

```
1   struct $thread\_extension$
2       $sh\_shm$ : struct $stm\_thread$
3       $wait$ : waitqueue

    // Kernel initialization of shared data structures
4   upon KERNELSTMINIT($t$)
5       $t.sh\_stm \leftarrow$ GETNEXTFREESTMSTRUCT()
6       $tx.sh\_stm.thr.conflict\_thr \leftarrow$ null
7       $tx.sh\_stm.thr.commit \leftarrow$ false
8       $t.wait \leftarrow$ INITQUEUE()
    // Thread election, $\mathcal{Q}$ is the OS ordered ready queue
9   upon SCHEDULE($\mathcal{Q}$)
10      foreach thread $t$ do
11          if $t.sh\_stm$ then
                // ReleaseAll on Commit
12              if $t.sh\_stm.commit$ then
13                  $t.sh\_stm.commit \leftarrow$ false
14                  foreach thread $t'$ in $t.wait$ do
                        // Unblock threads conflicting with committed
                        transactions
15                      if $t'.sh\_stm.conflict\_tx \neq t.sh\_stm.tx$ then
16                          MOVETOREADYQUEUE($t'$)

                // Wait on Conflict
17              $t' \leftarrow t.sh\_stm.conflict\_thr$
18              if $t' \neq$ null then
19                  $t.sh\_stm.conflict\_thr \leftarrow$ null
20                  if ONREADYQUEUE($t'$) then
21                      MOVETOWAITQUEUE($t$, $t'.wait$)

22      ELECT(FIRST($\mathcal{Q}$))
```

The main source of overhead in both of the above blocking-based strategies is due to the need to interact with the kernel to restore the normal execution of the loser threads on each Commit. A third possible solution is then to release the CPU in a way that does not require explicit unblocking. For this, we implement Wait using the `yield` system call and ReleaseAll as a no-op, as shown in Algorithm 4. Yield instructs the scheduler to elect a new thread, while considering that the yielded thread has the lowest possible priority. If there are more threads than cores, the yielding thread will be blocked for at least one time slice. This strategy is thus beneficial when the winner transaction is short, implying that this amount of time is sufficient to allow it to complete.

Algorithm 4: Yield (system-call based)

```
   // Start transaction $tx$
1  upon Start($tx$)
2      $tx.thr$ ← CurrentThread()

   // Conflict between $tx$ and $tx'$
3  upon Conflict($tx$, $tx'$)
4      Abort($tx$)
5      Yield($tx.thr$ )

   // Commit transaction $tx$
6  upon Commit($tx$)
       // No action
```

Non-determinism Serialization relies on the property that, once a transaction has conflicted with another transaction, it is likely to continue to do so. If a transaction is non-deterministic, however, then this property may not hold. Non-determinism occurs when a transaction takes a different code path when it is restarted relative to its previous execution. This can occur, for example, because the values that are used for making branching decisions were changed between the two incarnations of the transaction. Blocking the loser is too drastic in this case. As the future behavior of a possibly non-deterministic transaction is impossible to predict, we propose an alternative strategy, SoftSer, that reduces the priority of the loser, rather than blocking it completely. With a scheduler such as that of Linux, the loser will then only be elected if there are no other higher-priority ready threads on the current core.

The implementation of SoftSer, following a system-call based approach, is shown in Algorithm 5. Because a thread in conflict remains able to run while in a conflict state, it may encounter a conflict with a given thread multiple times or it may encounter conflicts with multiple threads. To manage information about multiple conflicts, the implementation of SoftSer uses a two-dimensional boolean array $\mathcal{C}$, where $\mathcal{C}[i][j]$ is true if and only if thread i has detected a conflict with thread j, and a conflict counter for each thread that indicates the number of transactions with which it is currently in conflict. When thread i detects a conflict with thread j, Conflict sets $\mathcal{C}[i][j]$ to true and increments thread i's conflict counter (lines 8–9). In Commit, the column $\mathcal{C}[t][*]$ for the thread t performing the commit is cleared, to indicate that other threads are no longer in conflict with the current one, and the conflict counter for each such thread is decremented. The row $\mathcal{C}[*][t]$ is also cleared, since the thread is no longer in a transaction. When a thread is no longer in conflict with any transaction, as indicated by its conflict counter reaching 0, its priority is returned to normal.

This system-call based implementation of SoftSer is converted to a shared-memory implementation as done for Ser. In particular, the STM-level code (Algorithm 2) is identical.

Assessment Table 1 summarizes the tradeoffs between the various serialization strategies. Different strategies are beneficial in different contexts, depending on the relationship between the number of threads and the number of cores, the duration of transactions, and the degree of contention. − indicates a particular disadvantage for a given strategy, while + indicates a particular advantage. Spinlocks only work well when there are fewer threads than cores. Serialization with condition variables may incur too much overhead for very short transactions. Shared memory-based serialization may incur latency, but this is mitigated when there are many aborts. Finally, yielding is most effective for transactions shorter than a time slice.

Algorithm 5: SoftSer (system-call based)

```
1  $\mathcal{C}[*][*]$ ← false                       // Conflict matrix, initialized to false
   // Start transaction $tx$
2  upon Start($tx$)
3      $tx.thr$ ← CurrentThread()
4      $tx.thr.conflict\_count$ ← 0

   // Conflict between $tx$ and $tx'$
5  upon Conflict($tx$, $tx'$)
6      Abort($tx$)
7      if $\neg\mathcal{C}[tx.thr][tx'.thr]$ then
8          $\mathcal{C}[tx.thr][tx'.thr]$ ← true
9          $tx.thr.conflict\_count$ ← $tx.thr.conflict\_count + 1$
10         ChangePrio($tx.thr$, low)

   // Commit transaction $tx$
11 upon Commit($tx$)
12     foreach thread $t$ do                          // Clear column
13         if $\mathcal{C}[t][tx.thr]$ then
14             $\mathcal{C}[t][tx.thr]$ ← false
15             $t.conflict\_count$ ← $t.conflict\_count - 1$
16             if $t.conflict\_count = 0$ then
                   // Reset priority if no more conflicts
17                 ChangePrio($t$, normal)
18         $\mathcal{C}[tx.thr][t]$ ← false                  // Clear row
19     ChangePrio($tx.thr$, normal)                  // Reset priority

20 upon Schedule($\mathcal{Q}$)
21     Elect(First($\mathcal{Q}$))
```

	# threads ≤ # cores	# threads > # cores		
		Short transactions		Long transactions
		Few conflicts	Many conflicts	
Ser-u (spin)	+	−	−	−
Ser-u (cond)		−	−	
Ser-k		−	+	
Yield		+	+	−

Table 1. Comparison of the benefits of the serialization strategies

4. Time-Slice Extension

Serialization addresses the problem of repeated aborts, by preventing a thread from executing if it is likely to conflict again. An orthogonal approach is to prevent aborts in the first place, by ensuring that each transaction can commit as quickly as possible. In an operating system such as Linux with a priority-based round-robin scheduler, a thread is suspended when its time slice has expired or when a thread with a higher priority is unblocked. If the thread is currently running a transaction, being suspended in this manner dramatically increases the probability for another transaction to create a conflict, by modifying some shared data, or for it to create a conflict with other transactions, by hiding its own modifications to shared data over one or more time slices.

Strategy Ext minimizes this phenomenon by deferring the preemption of a thread that is running a transaction. In order to prevent a thread running a very long transaction from monopolizing a processor, a counter is associated with the thread, meaning it only gets a maximum number N of such "extensions" before it is actually suspended. Finally, if a thread has benefited from an extension, it yields after the next commit, again to ensure that it does not monopolize the processor. Strategy EXT only has an impact on each thread in isolation, rather than on the interaction between winner and loser threads. Thus, any of the serialization strategies can be augmented to use time-slice extension.

A shared-memory based implementation Because Linux does not provide a time-slice extension system call,[3] we consider only a shared-memory based implementation. As shown in Algorithm 6, this implementation sets a flag indicating that the thread is in a transaction in START (line 8) and clears it in COMMIT (line 11). On each invocation of the scheduler, which represents a potential preemption point, the scheduler checks whether the current thread is running a transaction (line 18) and if so whether the number of extensions received for the current transaction is less than the limit N. If both conditions are satisfied, the scheduler increments the extension counter, performs some Linux-specific bookkeeping operations, and then returns (lines 20-21), thus preventing the thread from being preempted.

Algorithm 6: EXT (shared-memory)

```
   // Thread structure in shared memory
1  struct $stm\_thread$
2      $tx$ : transaction_id
3      $extensions$ : int
   // Init of the thread before using the STM
4  upon KERNELSTMINIT($t$)
5      $t.sh\_stm \leftarrow$ GETNEXTFREESTMSTRUCT()
   // Start transaction $tx$
6  upon START($tx$)
7      $tx.thr \leftarrow$ STMSHAREDSTRUCTPTR()
8      $tx.thr.tx \leftarrow tx$
9      $tx.thr.extensions \leftarrow 0$
   // Commit transaction $tx$
10 upon COMMIT($tx$)
11     $tx.thr.tx \leftarrow$ null
12     if $tx.thr.extensions > 0$ then
13         YIELD($tx.thr$)
   // Extension of OS thread structure
14 struct $thread\_extension$
15     $sh\_shm$ : struct $stm\_thread$
   // Thread election, $\mathcal{Q}$ is the OS ordered ready queue
16 upon SCHEDULE($\mathcal{Q}$)
17     $t \leftarrow$ CURRENTTHREAD()
       // Check that the thread uses the STM and is in a transaction
18     if $t.sh\_stm \neq$ null $\wedge$ $t.sh\_stm.tx \neq$ null then
19         if $t.sh\_stm.extensions < N$ then
20             $t.sh\_stm.extensions \leftarrow t.sh\_stm.extensions + 1$
21             return                    // The current thread keeps running
22     ELECT(FIRST($\mathcal{Q}$))
```

5. Evaluation

The goal of our experimental evaluation is to study the relative performance of the various scheduling algorithms we have proposed. In addition to these algorithms, we consider for comparison CM_SUICIDE, which we refer to as *basic*, a minimal contention manager provided by TinySTM that just restarts a transaction on abort.

[3] Solaris does provide such a system call, but with a somewhat different semantics.

Our tests have been carried out on an AMD Opteron server with four 2.3 GHz quad-core CPUs (16 cores in total) and 8GB RAM running Linux 2.6.30. Our implementation extends the default Linux scheduler (CFS), which works with per-CPU task queues. We use two types of benchmarks: synthetic and realistic. In our benchmarks, we focus on throughput (commit rate), as would be relevant to a server application that receives a stream of continuous requests.

5.1 Synthetic benchmarks

We consider three categories of synthetic benchmarks: those with very short transactions (skip list and red-black tree), those with medium-length transactions (linked list), and those with very long transactions (STMBench7). The transaction durations are shown in Figure 3.[4] These results were obtained in single threaded mode, where there is no contention on data access. The benchmarks with very short transactions represent the worst case in terms of any overhead that is added by contention management. The benchmarks with very long transactions represent the worst case in terms of the number of aborts per transaction.

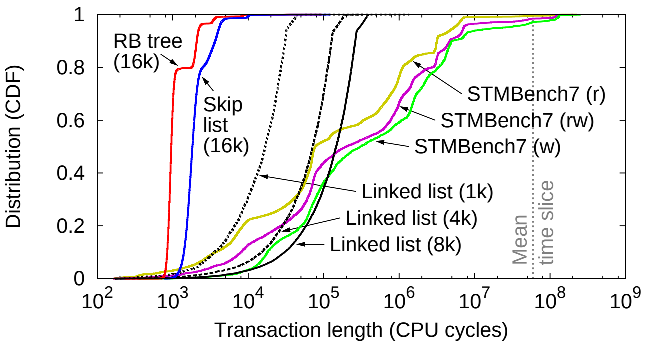

Figure 3. Transaction lengths for the synthetic benchmarks.

The skip list (SL), red-black (RB) tree, and linked list (LL) benchmarks all manipulate a set of integers. An execution consists of both read transactions, which determine whether an element is in the set, and update transactions, which either add or remove an element (reads are also done to find the position of the element to add or remove). The set is initially populated with a given number of elements and its size is maintained constant by alternating insertions and removals. SL and RB use data structures designed to make it possible to access any element by traversing only a few other elements, and thus exhibit a high rate of potential parallelism. For LL, accessing an element requires traversing all previous elements, implying that any write to a previous element that occurs before a transaction completes causes a conflict. This benchmark thus has less potential parallelism. Finally, STMBench7 performs read and write accesses on a large graph of objects [9]. The benchmark allows varying the ratio of reads to writes, leading to read dominated, write dominated, or mixed workloads. STMBench7 furthermore includes nondeterministic transactions.

For each benchmark, Table 2 summarizes the transaction length (average μ and standard deviation σ), the average number of reads and writes per transaction, and the amount of contention (percentage of transactions that abort at least once on an execution with no scheduling algorithm activated for 2, 8 and 16 threads on a 16-core

[4] Figure 3 presents the duration of the transaction in cumulative distribution frequency (CDF) format. In this format, each point represents the percentage of transactions that have a duration less than the value indicated on the x-axis. For example, a vertical line from the bottom to the top of the graph, as in the case of RB tree, indicates that most transactions have the duration indicated on the x-axis at the point of the increase. A gradual increase, as in the case of STMBench, indicates that there are transactions of many different durations.

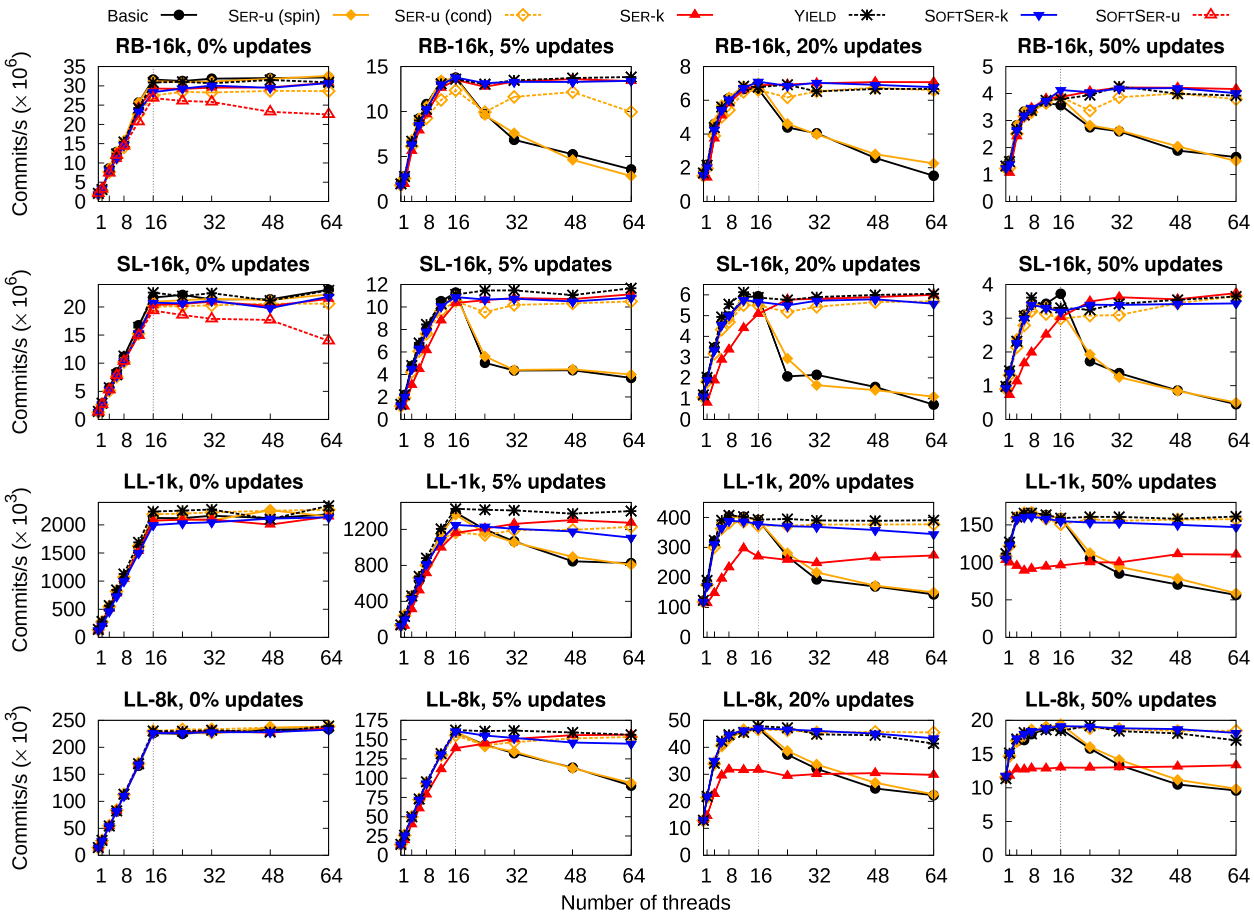

Figure 4. Commit rate of the red-black tree, skip list, and linked list benchmarks.

server). For LL, where we vary the size of the set (1k, 8k) and the update rate (only numbers for 20% are shown), the transaction length and number of reads increase roughly linearly in the size of the list. The number of writes for a given update rate is constant. For small sizes and update rates, the contention increases roughly linearly with the update rate, but reaches a maximum at around 20% due to the constraint of the number of available processors.

Figure 4 shows the commit rate of the LL, SL, and RB benchmarks with different update rates. Experiments were run with up to 64 threads, i.e., four times the number of available cores, to evaluate the benefits of our scheduling algorithms when contention increases and threads have to share CPUs. Indeed, given the small size of the transactions and the limited contention, there is not much room for improvement when the number of threads does not exceed the number of cores.

Application	Tx length (cycles)		Reads	Writes	Contention (%)		
	μ	σ	μ	μ	@2	@8	@16
`rb-16k-20%`	1,258	1.1e3	31.6	1.8	0.01	0.06	0.11
`sl-16k-20%`	2,204	1.4e3	58.2	0.9	0.03	0.25	0.44
`ll-1k-20%`	17,298	1.1e4	1,034	0.4	6.38	22.08	23.48
`ll-8k-20%`	149,606	9.0e4	8,168	0.4	6.37	21.02	22.33
`sb7-r`	1,530,040	8.1e6	1,150	17.1	2.10	10.67	15.46
`sb7-rw`	3,564,994	1.5e7	869.8	65.5	4.32	16.73	20.10
`sb7-w`	5,626,921	2.1e7	483.7	135.6	5.15	16.63	20.07

Table 2. Workload characteristics for the synthetic benchmarks.

We first analyze the minimal overhead of contention management. To this end, we have run experiments with just read-only transactions, implying that there are no collisions and no expected benefit from the scheduling algorithms. The ideal behavior is that throughput rises linearly as the number of threads increases until the number of threads reaches the number of processors, at which point throughput levels off. This pattern is exhibited by all of the algorithms except SOFTSER-u, and indeed all of these algorithms have essentially the same throughput as basic, which has the least overhead. SER-u (cond), SER-k, and SOFTSER-k sometimes has slightly lower throughput, as they all perform some extra operations on commit. Unlike the other algorithms, SOFTSER-u has a high overhead for the SL and RB benchmarks, which have very short transactions. This overhead comes from the traversal of the conflict array $\mathcal{C}$ required on every commit. In SOFTSER-k, this overhead is much reduced, because such traversals are handled by the kernel once per call to the scheduler function. With such short transactions, and no conflicts, there are many more commits than calls to the scheduler. Because of this high overhead, we omit SOFTSER-u from our other tests.

Next, we analyze the benefits of our algorithms when increasing the percentage of updates, and hence the level of contention. When there are fewer threads than cores, adding more threads leads to a declining increase in throughput, most notably for 50% updates, where the maximum throughput is often reached at 4 or 8 threads. For more threads than cores, throughput typically holds steady for the blocking and yielding algorithms, but drops off significantly for the purely user-level algorithms, basic and SER-u (spin), in which an aborting transaction never releases the processor. For RB, which has the shortest transactions, SER-u (cond) has also has some drop off in throughput, due to the high overhead for managing condition variables as compared to the transaction length. Overall, SER-k has lower throughput than the other blocking and yielding algorithms. This is because of the latency of waking up blocked threads after a commit. Nevertheless, for RB and SL, which have very short transactions, when there are many threads, and thus many conflicts, there are frequent calls to the scheduler and SER-k has throughput comparable to the others. Finally, YIELD often gives the best throughput for low rates of contention (5% updates), reflecting the fact that, unlike the other serializing algorithms, it has no overhead on commit.

Figure 5 shows the commit rate when using time-slice extension, limited for space reasons to combinations with SER-k and YIELD and 20% updates. As previously noted, SER-k already gives good performance when there are short transactions and frequent conflicts, and thus time slice extension combined with SER-k gives little added benefit for RB and SL. For LL, however, where the performance of SER-k degrades when there are more threads than cores, the addition of time-slice extension restores scalability, leading to throughput that is the same as the other serializing and yielding algorithms in the 1K case. YIELD, on the other hand, already has among the best throughputs on all three applications, and thus time slice extension does not give a significant improvement in this case.

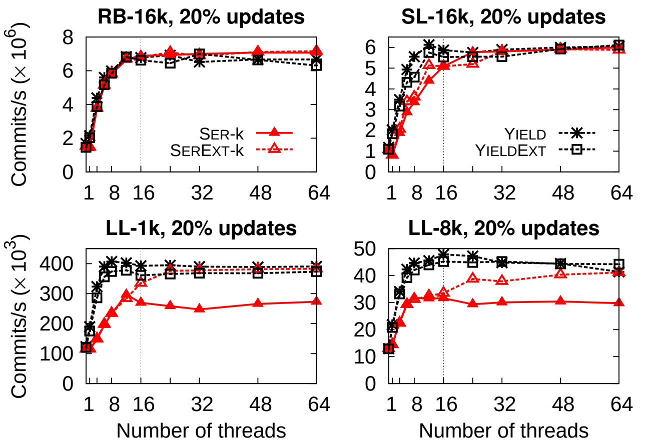

Figure 5. Commit rate of red-black tree, skip list, and linked list benchmarks when using time-slice extension.

We now study the performance of our algorithms with STMBench7 on three workloads that differ in the percentage of writes performed by the transactions (Figure 6). This benchmark has longer transactions, as well as nondeterministic transactions. In the read-dominated workload, SER-u (spin) gives the best throughput when there are fewer threads than cores, because it has low overhead, but unlike basic and YIELD does block a thread that encounters a conflict. YIELD is notably ineffective when the number of threads exceeds the number of cores because the yield duration is often not sufficient to allow the winner transaction to complete. For more threads than cores, for the read-dominated workload, SER-u (cond) gives the best throughput, as the cost of a system call is negligible as compared to the duration of a transaction. On the other hand, in the write-dominated workload, the presence of nondeterministic transactions has a more significant impact, and the blocking serialization algorithms, which can block a transaction that might be able to commit due to taking an alternate execution path, all give significantly reduced performance. In this case, YIELD and SOFTSER-k typically give the best throughput, as they allow a nondeterministic transaction to continue executing, although at lower priority.

The right side of Figure 6 shows the effect of combining time-slice extension with SER-k, YIELD and SOFTSER-k, named SEREXT-k, YIELDEXT, and SOFTSEREXT-k, respectively. For read-dominated workloads, when the number of threads is less than or equal to the number of cores, the combination of time-slice extension with YIELD and SOFTSER-k gives a significant benefit. These combinations, however, give less benefit for more threads than cores, or when there are more writes. On the other hand, combining time slice extension with SER-k gives the greatest benefit for high contention, *i.e.*, when there are 64 threads, in all of the workloads.

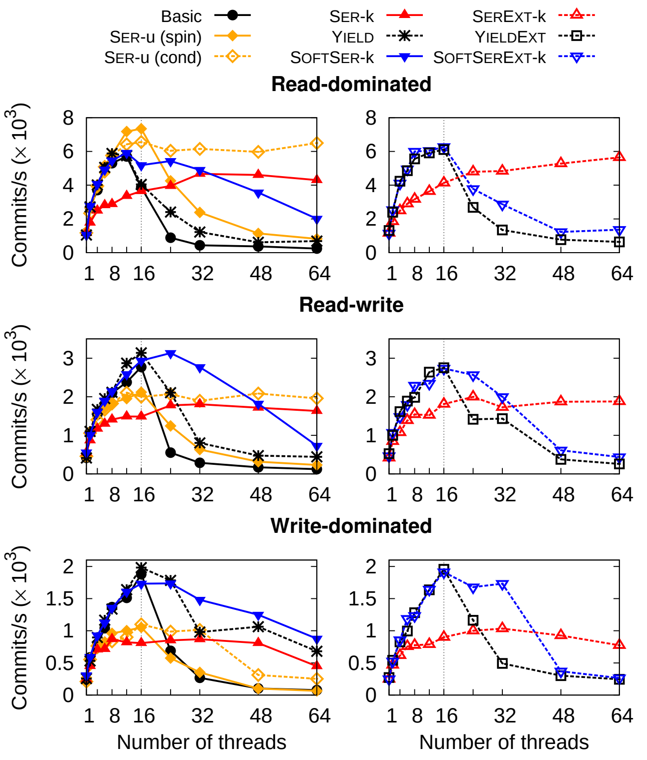

Figure 6. Commit rates for STMBench7.

We also show that the new scheduling algorithms dramatically reduce the rate of aborts. Figure 7 presents the abort rates corresponding to the experiments shown in Figure 6 (data for other algorithms and other benchmarks are omitted due to space limitations, but point to similar trends). We observe that all algorithms reduce the abort rate significantly relative to the basic contention management scheme; indeed, the serializing algorithms reduce the abort rates by several orders of magnitude. Although YIELD and SOFTSER-k (with and without time-slice extension) reduce aborts to a smaller extent than the serializing algorithms, they do not necessarily perform worse: an abort rate that is very small may be indicative of an overly strict serialization, thus reducing the abort rate should not be the sole goal of the scheduling algorithm.

5.2 Realistic applications

We now consider a collection of realistic applications that are not trivially parallelizable without synchronization and can thus benefit from transactional memory's optimistic concurrency. These applications are taken from the STAMP [3] benchmark suite, which is the most widely used STM benchmark. `bayes` uses a hill-climbing algorithm that combines local and global search to learn the structure of Bayesian networks from observed data; `genome` matches a large number of DNA segments to reconstruct the original source genome; `intruder` emulates a signature-based network intrusion detection system; `kmeans` partitions objects in a multi-dimensional space into

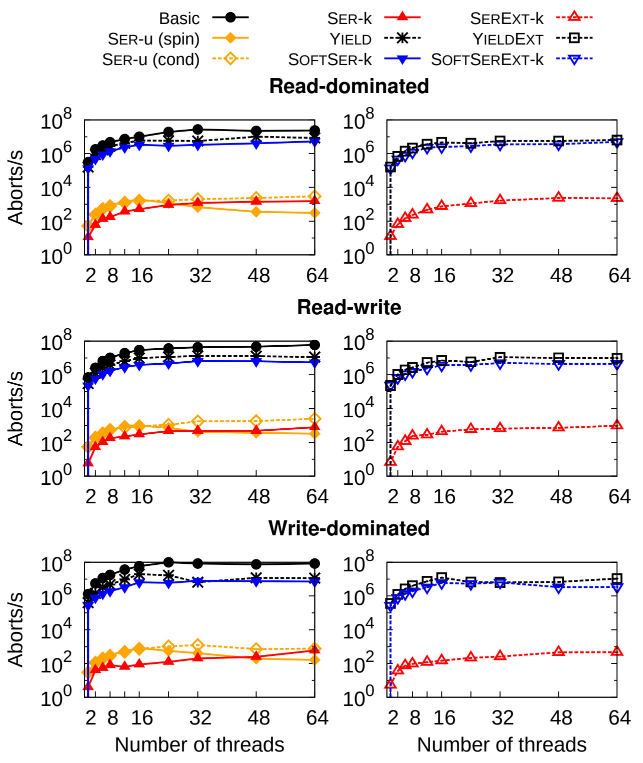

Figure 7. Abort rates for STMBench7.

a given number of clusters; `labyrinth` executes a parallel routing algorithm in a 3-dimensional grid; `ssca2` constructs a graph data structure using adjacency arrays and auxiliary arrays; `vacation` implements an online travel reservation system; `yada` executes a Delaunay mesh refinement algorithm. Additionally, two sets of parameters are recommended by the STAMP developers for `vacation` and `kmeans`, to produce executions with low and high contention. The single-threaded execution time of STAMP applications ranges from a few seconds to several minutes. Figure 8 presents the transaction lengths, and Table 3 summarizes the characteristics of the transactional workloads produced by these applications.

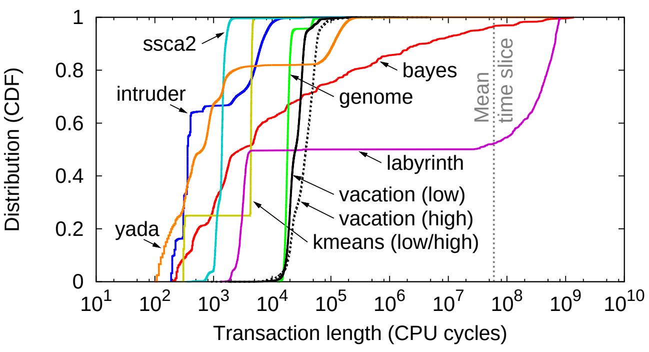

Figure 8. Transaction lengths for the STAMP benchmarks.

The performance of the STAMP benchmarks is shown in Figure 9, as compared to that of the application when run in a single thread, without transactions (1 represents identical performance). Note that for more threads than cores, the performance is overall significantly better than that with the conventional contention managers, as presented in Figure 1.

Application	Tx length (cycles)		Reads	Writes	Contention (%)		
	μ	σ	μ	μ	@2	@8	@16
`ssca2`	1,475	2.3e3	1.0	2.0	3.6e-4	2.6e-3	6.6e-3
`genome`	19,803	9.4e3	30.1	0.03	0.08	0.27	0.41
`vacation-l`	27,039	1.6e4	283.0	5.4	0.02	0.17	0.38
`vacation-h`	39,197	2.6e4	386.7	7.8	0.05	0.35	0.72
`bayes`	14,587,146	9.6e7	28.6	3.2	0.45	2.63	3.95
`yada`	25,664	6.7e5	60.8	18.8	3.31	6.72	6.60
`labyrinth`	207,825,190	2.6e8	180.1	177.0	1.85	6.06	10.56
`intruder`	2,197	3.9e3	23.6	2.7	1.89	23.72	33.93
`kmeans-l`	3,387	2.1e3	25.0	25.0	25.6	31.34	32.40
`kmeans-h`	3,293	1.9e3	25.0	25.0	28.5	45.79	41.33

Table 3. Workload characteristics for the STAMP benchmarks.

`ssca`, `genome`, `vacation` (low and high), and `bayes` all have very low contention. `ssca` has very short transactions, and all algorithms, including basic, maintain the same level of performance as the number of threads increases beyond the number of cores. `genome` has longer transactions and the performance when using basic or SER-u (spin) drops off somewhat after 16 threads, but all of the blocking contention manages maintain good performance. Due to the length of the transactions, YIELD performs little better than basic. `vacation` has even longer transactions, and basic, SER-u (spin), and ultimately YIELD, fall well below the single-threaded case. All of the blocking algorithms drop off in performance after 16 threads, but then level off, maintaining performance well above the single-threaded rate. Finally, `bayes` has very long transactions. Performance is mixed because the workload is quite random [3], and in most cases drops off well before 16 threads.

`yada` and `labyrinth` have moderate contention. `yada` has moderate length transactions, while those of `labyrinth` are quite long. In `yada` the transactions have very large read and write sets, making the management of aborts and the restart process very expensive. The blocking algorithms, SER-u (cond) and SER-k, result in execution that is essentially the same as single-threaded execution. This problem does not occur in `labyrinth`, where all of the algorithms, including basic, show only a slight decline in performance as the number of threads increases beyond the number of cores.

Finally, `intruder`, `kmeans-l`, and `kmeans-h` all have high contention, but short transactions. Basic and SER-u (spin) incur a substantial dropoff in performance already when there are the same number of threads as cores, while the blocking, yielding, and soft serialization algorithms maintain some degree of speedup over the single-threaded case.

5.3 Solaris Implementation

To demonstrate the portability of our techniques we also present results for a subset of the algorithms implemented on the OpenSolaris platform. We have implemented two of the algorithms for OpenSolaris: SOFTSER-u and YIELDExt. The implementation of the former is somewhat different than the Linux version, in that it executes at user level and does not maintain the conflict array $\mathcal{C}$. Instead, a thread that aborts reduces its own priority, and resets it back to normal only when it commits. Due to space limitations, we only show the results for benchmarks LL-1K, LL-8K and RB-16K for 0%, 20% and 50% updates. Our tests were carried out on an AMD Opteron server with four 2.3 GHz quad-core CPUs (16 cores in total) and 32GB RAM running OpenSolaris. As on Linux, the scheduling-based algorithms maintain the throughput as the number of threads exceeds the number of cores. We next look at the results, presented in Figure 10, in detail.

For the scenarios with 0% updates, all algorithms perform roughly similarly. No contention management is needed, since there are no conflicts in this scenario, and the only differences in performance are due to bookkeeping overhead. We do not

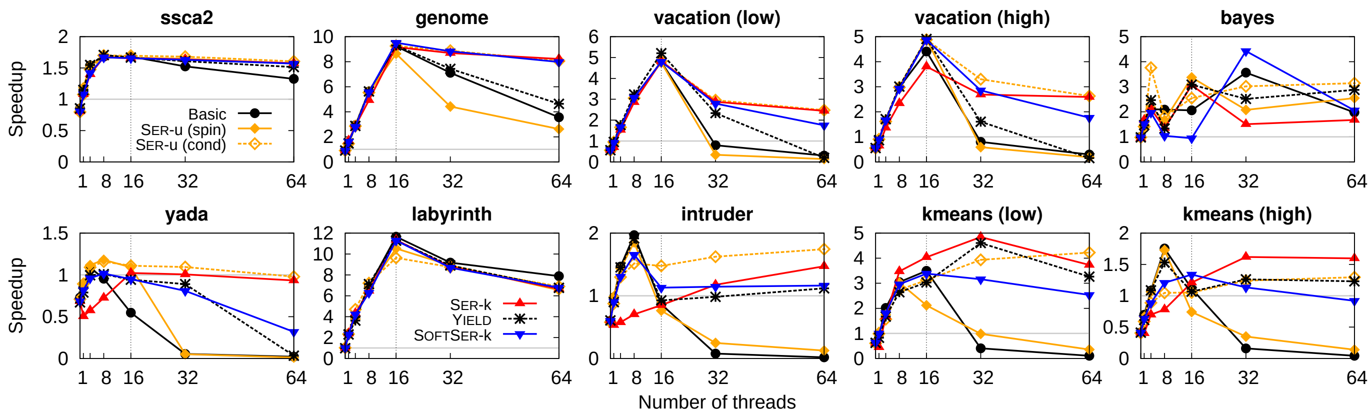

Figure 9. Speedup of the STAMP benchmarks as compared to single-threaded execution.

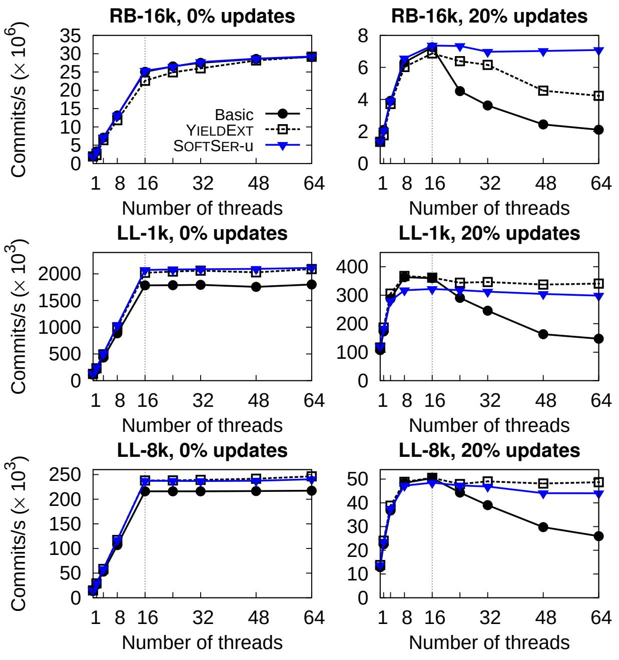

Figure 10. Commit rate of the red-black tree and linked list benchmarks on OpenSolaris.

observe any significant overheads for our contention management algorithms. As on Linux, for the experiments with 20% and 50% updates, the basic contention management scheme is unable to retain high throughput as the number of threads grows beyond the number of cores. The new contention management algorithms, however, enable the application to retain high throughput with only a slight degradation in performance. SOFTSER-u does well on all benchmarks. YIELDEXT does well for the LL benchmark, but slightly worse for RB, where its performance tapers off as the number of threads increases.

These experiments suggest that our techniques are portable across operating systems and that their effectiveness does not hinge on specific implementation details of any particular OS.

6. Related Work

STM implementations must resolve transaction conflicts in order to guarantee progress. Typically this is done by delegating conflict resolution to a *contention manager* [15, 16]. As discussed in Section 2, conventional (non-scheduling) contention managers have significant limitations; notably they lack precision and have very limited control (or no control at all) on the scheduling of transactional threads. Conventional contention managers also take a reactive approach, by only acting once a conflict has been detected. In contrast, our time-slice extension approach can also prevent conflicts from happening by increasing the probability that threads are not preempted in the midst of performing a transaction.

CAR-STM [5] also takes a scheduling-based approach for contention management. It maintains per-core transaction queues, where the transactions in each queue are executed by a dedicated thread in a sequential manner. Upon collision, the loser transaction is enqueued behind the winner transaction. This mechanism is similar to our strategy SER-u, but CAR-STM serializes all loser transactions aborted by the same winner one after the other, which may overly reduce parallelism for many workloads.

Yoo and Lee [19] implemented a simple adaptive user-level scheduler that essentially serializes transactions once a high level of contention is detected. This approach is effective in specific settings where parallelism actually degrades performance. 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. Unlike these approaches, which employ synchronization mechanisms solely at the user level, we also implement and evaluate OS kernel support for STM contention management in order to take advantage of existing kernel scheduling capabilities and reduce synchronization overhead.

TL2's [4] implementation on Solaris uses the *schedctl* mechanism to request short-term preemption deferral during the commit phase.[5] Like our strategy EXT, this reduces the risk that a transaction holding locks is preempted and prevents the progress of others. It does not, however, control the scheduling of an active transaction that has already accessed shared data but did not yet start the commit phase, and for which preemption would also increase the chances of an abort.

Recent work by Dragojevic et al. [6] presented *Shrink*, a user-level transaction scheduler that bases its scheduling decisions on the access patterns of past transactions. Similarly to our strategy

[5] This strategy is not used in TL2's x86 implementation on Linux as it does not support the *schedctl* mechanism.

EXT, *Shrink* can prevent collisions before they occur. *Shrink* uses a serialization mechanism similar to that of Yoo and Lee.

TxLinux [13] is a variant of Linux that exploits hardware transactional memory (HTM) and integrates transactions with the operating system scheduler. It follows different goals and a different approach than our work, by focusing on HTM and experimenting with new ways of achieving synchronization in the kernel for future processors with TM hardware support.

7. Conclusion

In this paper, we have proposed scheduling-based approaches to transactional memory contention management. We have defined and implemented novel serialization and time-slice extension strategies, and have studied their relative performance in the context of workloads that vary in the relationship between the number of threads and the number of cores, the duration of transactions, and the degree of contention. In this, we have particularly focused on the tradeoffs involved in using user-level and kernel-level contention managers.

Our approach has been implemented in Linux, and selected algorithms have been implemented in Solaris. In both implementations, the changes we have made to the kernel require adding a total of around 500 lines of code for all strategies and do not have any impact on the existing scheduling logic. To the best of our knowledge, our work is the first to implement support for STM contention management in the kernel.

Our evaluation establishes that lightweight kernel-level scheduling support enables TM applications to retain high throughput even when the number of threads exceeds the number of cores and under high contention. Moreover, even when the number of threads is smaller than the number of cores, applications benefit from significantly smaller abort rates, while retaining their throughput.

Which scheduling strategy is best depends on the workload and the execution context. When there are few updates and relatively short transactions, a simple yield on conflict is often sufficient. Blocking serialization algorithms (SER-u (cond) or SER-k) often give the best performance for realistic applications, as represented by STAMP. However, SER-k gives the worst performance for moderate-length transactions with more threads than cores due to the incurred latency on restarting blocked transactions; this is exemplified by LL, which shows essentially the same throughput from 1 to 64 threads. Finally, SOFTSER-k gives the best throughput for non-deterministic transactions, as exemplified by StmBench7, because it allows a transaction to continue, at reduced priority, after a conflict.

Dynamically determining the most appropriate scheduling strategy for a given workload seems to be a non-trivial challenge. We believe our current work provides a required foundation for such an endeavor, as the scheduling algorithms we present may serve as building blocks of such a dynamic contention manager. We leave this for future work.

References

[1] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4(10):24–33, 2007.

[2] Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris C. Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *High Performance Embedded Architectures and Compilers, Fourth International Conference (HiPEAC)*, pages 4–18, 2009.

[3] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of IISWC*, September 2008.

[4] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *20th International Symposium on Distributed Computing (DISC)*, pages 194–208, September 2006.

[5] Shlomi Dolev, Danny Hendler, and Adi 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, August 2008.

[6] Aleksandar Dragojevic, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Twenty-Eighth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 7–16, August 2009.

[7] Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Research Cambridge, 2006. IRC-TR-06-052.

[8] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of PPoPP*, pages 237–246, February 2008.

[9] Rachid Guerraoui, Michał Kapałka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. In *Proceedings of the Second European Systems Conference EuroSys 2007*, pages 315–324. ACM, March 2007.

[10] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Twenty-Second ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, July 2003.

[11] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Annual International Symposium on Computer Architecture (ICSA)*, pages 289–300, May 1993.

[12] James Larus and Christos Kozyrakis. Transactional memory. *Communication of the ACM*, 51(7):80–88, July 2008.

[13] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Aditya Bhandari, and Emmett Witchel. TxLinux: Using and managing hardware transactional memory in an operating system. In *Proceedings of SOSP*, pages 87–102, October 2007.

[14] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of PPoPP*, pages 187–197, March 2006.

[15] William N. Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[16] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of PODC*, pages 240–248, July 2005.

[17] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of PODC*, pages 204–213, August 1995.

[18] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[19] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of SPAA*, pages 169–178, June 2008.