Transaction Activation scheduling Support for Transactional Memory

Walther Maldonado, Patrick Marlier, Pascal Felber, Julia Lawall, Gilles Muller

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Transaction Activation Scheduling Support for TransactionalMemory

Walther Maldonado —
Patrick Marlier —
Pascal Felber — Julia Lawall — Gilles Muller

N° 6807

Janvier 2009

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6807--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Transaction Activation Scheduling Support for TransactionalMemory

Walther Maldonado*,
Patrick Marlier* ,
Pascal Felber * , Julia Lawall† , Gilles Muller‡

Thème COM — Systèmes communicants
Équipe-Projet Regal

Rapport de recherche n° 6807 — Janvier 2009 — 25 pages

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, produce numerous aborts. These problems are traditionally addressed by application-level contention managers, but they suffer from a lack of precision and provide unpredictable benefits on many workloads.

In this paper, we propose a system approach where the scheduler tries to avoid aborts by preventing conflicting transactions from running simultaneously. We use a combination of several techniques to help reduce the odds of conflicts, by (1) avoiding preempting threads running a transaction until the transaction completes, (2) keeping track of conflicts and delaying the restart of a transaction until conflicting transactions have committed, and (3) keeping track of conflicts and only allowing a thread with conflicts to run at low priority.

Our approach has been implemented in Linux for Software Transactional Memory (STM) using a shared memory segment to allow fast communication between the STM library and the scheduler. It only requires small and contained modifications to the operating system. Experimental evaluation demonstrates that our approach significantly reduces the number of aborts while improving transaction throughput on various workloads.

Key-words: Software transactional memory, TinySTM, thread scheduling

A previous version of this document appeared as research report UniNE-8/11/2008

* University of Neuchâtel, Switzerland
† DIKU, University of Copenhagen, Denmark
‡ EMN–INRIA/Regal

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Transaction Activation Scheduling Support for TransactionalMemory

Résumé : La mémoire transactionelle (TM) est actuellement considérée comme l'un des paradigmes les plus prometteurs pour le développement d'applications concurrentes. Les études dans ce domaine ont montrées que la performance était proportionnelle au nombre de cœurs lorsque le canevas des accès mémoire possède un bon profil, c.à.d que peu de conflits sont engendrés entre les transactions. En revanche, lorsque les accès aux données induisent du partage fréquent en écriture, de longues transactions ou lorsque de nombreux fils de contrôle sont en compétition pour s'exécuter sur un nombre de cœurs inférieurs, il en résulte de nombreux avortements. Ces problèmes sont traditionellement traités par des gestionnaires de contention au niveau applicatif. Toutefois, les solutions existantes souffrent d'un manque de précision et entraînent des gains non prévisibles sur de nombreuses charges applicatives.

Dans cet article, nous proposons une approche système où l'ordonnanceur essaie de prévenir les avortements en évitant de faire s'exécuter simultanément des transactions conflictuelles. Plus précisément, nous utilisons une combinaison des techniques suivantes : (i) éviter de suspendre une thread tournant une transaction jusqu'au commit de celle-ci, (ii) mémoriser les conflits et retarder le redémarrage d'une transaction jusqu'à ce que les transactions conflictuelles aient commitées, (iii) mémoriser les conflits et permettre aux fils tournant des transactions en conflit de s'exécuter avec une priorité basse.

Notre approche a été mise en œuvre dans le contexte d'une mémoire transactionelle logicielle (STM) sur Linux en utilisant un segment de mémoire partagéee pour permettre des communications rapide entre l'ordonnanceur et la STM. Cette approche requiert peu de modifications dans le noyau de Linux. Nos évaluations montrent que cette approche réduit de manière significative le nombre d'avortements et qu'elle améliore le débit en transactions sur des charges applicatives variées.

Mots-clés : Mémoire transactionelle logicielle, TinySTM, ordonnancement noyau

1 Introduction

Transactional Memory (TM) is considered as one of the most promising paradigms for developing concurrent applications by exploiting in a simple and efficient manner the power of new multi-core processors [1, 9]. The basic idea is to protect, using lightweight transactions, blocks of instructions that access shared data. These transactions execute optimistically on the assumption that there is no concurrent operation on the same data; upon conflict, a transaction might implicitly abort and restart its execution. TM thus provides a simple and safe abstraction for using speculative execution in multi-threaded 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 [1, 9]. In contrast, a data pattern with frequent write sharing 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.

In the context of Software Transactional Memory (STM), the traditional approach to solve this problem is to use an application-level *contention manager*. The role of the contention manager is to react when a conflict is detected, by aborting one of the conflicting transactions or by waiting (using a sleep) for the conflict to be possibly resolved. Additionally, exponentially increasing delays can be inserted after an abort so as to reduce the probability of a further conflict.

There are several problems with these approaches: (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; (iii) unpredictable benefits, as delaying the restart of a long transaction does not necessarily increase its chance of success, unless all other conflicting transactions are also delayed. Therefore, "blind" conflict resolution techniques usually provide poor performance with many workloads commonly used to evaluate STMs [12, 13].

In this paper, we consider word-based STMs that use revokable locks, such as McRT-STM [11], TL2 [4] and TinySTM [7], because of their flexibility (they allows transactional accesses to arbitrary memory addresses) and their high efficiency [6]. We propose an approach where the scheduler tries to avoid conflicts by preventing conflicting transactions from running simultaneously. We use a combination of three strategies to help reduce the odds of conflicts: First, strategy S1 tries to avoid suspending threads running a transaction until the transaction completes (commits or aborts), thus reducing the "window of vulnerability." Second, strategy S2 keeps track of recent conflicts and marks threads that have just aborted so that the scheduler will not elect them until the conflicting transactions have committed. Finally, strategy S3 considers a variant of the previous strategy where a thread that has detected a conflict is allowed to run, but at low priority. While strategies S2 and S3 address problems that occur at the level of a single multithread application, S1 tackles an issue that arises as soon as two threads run on the same core, even if they belong to independent applications. This advocates for an implementation within the OS kernel because it is aware of and thus can adjust all scheduling decisions.

The potential problem with such a kernel approach is that transactions must communicate with the operating system to signal transactional events (start, commit, abort). For transactions, which are intended to be lightweight, the use of specialized system calls to implement this communication would induce

an unacceptable overhead. We propose a new solution based on the use of a memory segment shared between the STM library and the scheduler. The STM writes information into the shared memory segment, which is then used at scheduling time to take appropriate decisions.

The contributions of the paper are as follows:

- We propose novel approaches for improving the performance of software transactional memory by modifying the OS scheduler policies. We demonstrate that an OS implementation reduces the window of vulnerability even when multiple applications share the same core. To our knowledge, the impact of running multiple applications is a critical issue that has been overlooked until now.
- We have implemented our approach in the Linux kernel. Our modifications to the kernel are small and contained.
- We have evaluated our approach on both micro- and macro-benchmarks. 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 propose three system-level contention management strategies at an intuitive level, while Section 4 discusses the details of their implementation. 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 application-level contention managers have in accurately dealing with conflicts.

2.1 Software Transactional Memory

Software transactional memory (STM) [14] 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.

A typical API exported by an STM contains functions to start a transaction, access shared data for reading or writing, try to commit the transaction or force it to roll-back. In TinySTM [7], for instance, these functions are respectively called `stm_start`, `stm_load`, `stm_store`, `stm_commit`, and `stm_abort`. Additional hooks are provided to react when particular events happen, such as commit, abort, conflict, or restart. A transaction is bound to a single thread, and each thread can execute a sequence of transactions.

There exist several types of STM designs [9], which mainly differ in their liveness properties (e.g., blocking, obstruction-free, lock-free) and the granularity of conflict detection (e.g., memory word, object), as well as the various implementation choices they follow. In this paper, we consider a popular class of STMs, widely considered to be among the most efficient and general purpose, which follow a *word-based* and *lock-based* design (i.e., 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 notably includes Ennals' STM [6], McRT-STM [11], TL2 [4], and TinySTM [7].

A key to the performance of lock-based STM designs is the limited synchronization between the thread and the indirection-free access to shared data. In contrast, obstruction-free designs typically require extra indirections for every access and are often significantly slower than their lock-based counterparts (e.g., see [6]).

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 considered. 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 leads to longer transaction execution times and a higher 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), a STM component responsible for resolving conflicts between a pair of transactions at runtime. The CM determines based on various policies whether conflicting transactions should abort, wait, or proceed. Comparative studies of contention managers [12, 13] show that different strategies work better for different benchmark applications, but no single manager performs best on all workloads.

In lock-based STM, 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.

To evaluate the impact of conflicts with many concurrent transactions and the effectiveness of application-level contention management, we have run a set of benchmarks from the STAMP suite [3] on an 8-core machine with two STM implementations (see Section 5 for details on our experimental setup): TL2 (with and without exponential backoff CM), and TinySTM. The latter implementation can either use encouter-time locking (ETL), or commit-time locking (CTL), depending on whether a write is visible to other transactions immediately or only at commit time, respectively.[1]

In the graphs in Figure 2.2, we observe that: (i) performance degrades significantly on most STAMP benchmarks once there are more threads than cores,

[1] CTL typically reduces the risk of conflicts during transaction execution but increases the likelihood of an abort at validation time [7].

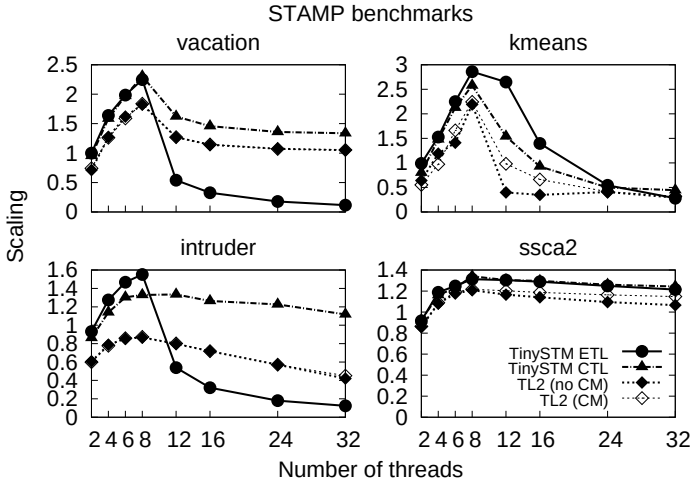

Figure 1: Performance of TinySTM (ETL and CTL) and TL2 on STAMP benchmarks.

except for `ssca2` that has short transactions and very few conflicts (see Table VI in [3]); (ii) degradation is generally less important using commit-time locking (TL2 and TinySTM-CTL), except for `kmeans` where we observe the opposite trend; and (iii) the backoff contention manager does not help.

There are several reasons why the application-level contention manager is inadequate. First, it typically lacks precision when delaying the restart: if the delay is too short the same conflict might be encountered again, if it is too long cycles might be wasted. Second, it takes local actions after a pairwise conflict has been detected and cannot easily make informed decisions based on current and past conflicts, when restarting a transaction. Third, it does not control when threads are preempted: if preemption occurs in the middle of an active transaction, the risk of conflict with concurrent transactions increases.

3 A System Approach to Contention Management

In this section, we present a new approach to contention management that relies on cooperation between the STM library and the OS scheduler. The basic idea is to enable the scheduler to take appropriate decisions in thread election by keeping it aware of transaction progress and conflicts. Our goal is to prevent wasting CPU resources by reducing the number of transaction aborts. We provide a high-level description of three scheduling strategies that explore variations around this idea. The presentation of these strategies is intended to be as clear as possible, so implementation issues are deferred to the next section.

We have developed three scheduling strategies (S_1–S_3) that fall into two complementary categories: (i) S_1 gives priority to threads currently running a transaction, and (ii) S_2 and S_3 avoid running a thread that will execute a

transaction that is likely to abort in the future. We now describe these strategies in detail.

3.1 Strategy S_1 — Avoiding transaction suspension

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. Strategy S_1 minimizes this phenomenon by deferring the preemption of a thread that is running a transaction. In order to prevent a busy transactional thread from monopolizing a processor, a counter is associated to 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.

Algorithm 1: Strategy S_1

```
   // Start transaction tx
 1 upon START(tx)
 2     tx.thr ← CURRENTTHREAD()
 3     tx.thr.tx ← tx
 4     tx.thr.extensions ← 0

   // Commit transaction tx
 5 upon COMMIT(tx)
 6     tx.thr.tx ← null
 7     if tx.thr.extensions > 0 then
 8         YIELD()

   // Thread election, Q is the OS ordered ready queue
 9 upon SCHEDULE(Q)
10     t ← CURRENTTHREAD()
11     if t.tx ≠ null ∧ t.extensions ≤ N then
12         t.extensions ← t.extensions + 1
13         return                                  // The current thread keeps running
14     ELECT(FIRST(Q))
```

3.2 Strategy S_2 — Preventing restart of an aborted transaction while a conflicting transaction is active

A transaction tx supported by a thread $tx.thr$ may abort due to a conflict with a transaction tx' supported by a thread $tx'.thr$. In this case, we assume that the application is deterministic, so a restart of tx will likely lead to an abort as long as tx' is active. Therefore, Strategy S_2 blocks the thread; it removes the thread from the scheduler's ready queue and moves it to a wait queue associated with the thread with which there is a conflict. When a transaction commits, all of the threads on the wait queue of the supporting thread are returned to the scheduler's ready queue, and thus become re-eligible for election.

Algorithm 2: Strategy S_2

```
   // Start transaction tx
 1 upon START(tx)
 2     tx.thr ← CURRENTTHREAD()

   // Conflict between tx and tx'
 3 upon CONFLICT(tx, tx')
 4     MOVETOWAITQUEUE(tx.thr, tx'.thr.wait )
 5     ABORT(tx)
 6     YIELD(tx.thr)

   // Commit transaction tx
 7 upon COMMIT(tx)
 8     for t in tx.thr.wait do                          // Remove conflicts
 9         MOVETOREADYQUEUE(t)

   // Thread election, Q is the OS ordered ready queue
10 upon SCHEDULE(Q)
11     ELECT(FIRST(Q))
```

3.3 Strategy S_3 — Allowing restart of an aborted transaction at a low priority while a conflicting transaction is active

Strategy S_3 is a variation of S_2 for applications with nondeterministic behavior. In that case, a restarted transaction may not always reproduce past conflicts. Therefore, S_3 allows electing a thread supporting a restarted transaction at a low priority until the conflicting transaction commits. With a scheduler such as that of Linux, such an election will occur only if the OS lacks ready threads and thus processors can be used to potentially waste cycles.

In S_3, a thread in conflict must remain at low priority until all of the threads with which it conflicts have committed the conflicting transactions. 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 be able to efficiently manage information about multiple conflicts, S_3 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. An element of this array is set to true in CONFLICT. 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 row $\mathcal{C}[*][t]$ is cleared as well, since the thread is no longer in a transaction. When a thread is no longer in conflict with any transaction, its priority should be returned to normal. This is indicated by all of the elements in the row associated with the thread being false.

4 Implementation

The naive descriptions of the contention management strategies do not specify how to implement synchronization between multiple processes or communication between the STM and the scheduler. The tasks performed by the transaction operations START, CONFLICT and COMMIT require updating the kernel structures that implement threads. Thus, these tasks must be performed at the kernel level. The transaction operations for strategies S_2 and S_3 furthermore

Algorithm 3: Strategy S_3

```
1  C[*][*] ← false                                   // Conflict matrix, initialized to false
   // Start transaction tx
2  upon START(tx)
3      tx.thr ← CURRENTTHREAD()

   // Conflict between tx and tx'
4  upon CONFLICT(tx, tx')
5      C[tx.thr][tx'.thr] ← true
6      tx.thr.priority ← LOW
7      ABORT(tx)
8      YIELD(tx.thr)

   // Commit transaction tx
9  upon COMMIT(tx)
10     foreach thread t do                           // Clear column
11         if C[t][tx.thr] then
12             C[t][tx.thr] ← false
13             if ∀t' : ¬C[t][t'] then
                   // Reset priority if no more conflicts
14                 t.priority ← NORMAL
15         C[tx.thr][t] ← false                      // Clear row
16     tx.thr.priority ← NORMAL                      // Reset priority

17 upon SCHEDULE(Q)
18     ELECT(FIRST(Q))
```

potentially require manipulating structures such as the wait queues that may be accessed concurrently by multiple threads, and thus require mutual exclusion.

A possible strategy for shifting work from the user level to the kernel level is to introduce specialized system calls. Because START and COMMIT occur frequently, however, using specialized system calls to implement communication between the STM at user level and the kernel to access the thread structures would induce an excessive overhead for these operations. Furthermore, as introducing a new system call is complex, we prefer to avoid it even for CONFLICT. To enable communication between the transaction operations and the kernel without resorting to system calls, we instead propose 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. The OS thread structure is enriched with a pointer into the shared memory thread structure.

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. As the execution of the program progresses, the application fills in data in the shared structure, and the scheduler processes this information accordingly.

It must be remarked that in this layout, it is not possible to avoid simultaneous access from the kernel and processes, which places a certain risk on data access coherency and limits the amount of information that can be placed and updated within the structure. Assignments and decisions taken by reads must for example take into account the effects of interleaving instructions between the kernel and user-space, and be careful with their ordering to avoid undesired

side-effects. Our use of the shared memory segment is structured such that any initializations at the user level that trigger behavior in the scheduler are preceded by initialization of the shared memory segment with all other information required for this behavior, and thus race conditions are not possible.

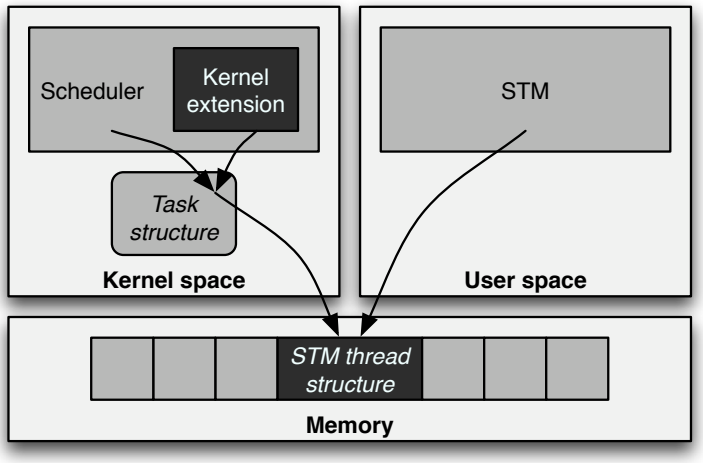

Figure 2: Layout of the interaction between the Kernel and application.

STM operations are then implemented as follows: (i) operations that must defer tasks to the kernel to obtain mutual exclusion are implemented by logging information into the shared memory thread structure, (ii) logged operations are applied by the kernel before electing a new thread.

4.1 Implementing strategy S_1

The implementation of S_1 is quite similar to the naive version described in Algorithm 1. The changes are related to the usage of the shared structure *stm_thread* containing the transaction id and time extension information. At the STM level, the START operation now initializes the *thr* field of the transaction to point to the shared memory structure associated with the current thread, rather than to the current thread itself. At the kernel level, the schedule function checks that the current thread uses the STM before accessing the shared structure. If the current thread is using the STM, the SCHEDULE function then accesses information in the shared structure to determine whether it is in a transaction and has any remaining extensions.

4.2 Implementing strategies S_2 and S_3

The changes for the implementations of S_2 and S_3 are more elaborate than for S_1 due to the need to defer treatments to the kernel function SCHEDULE. Both implementations use the same shared memory segment structure and definitions of the STM functions START, CONFLICT and COMMIT, as shown in Algorithm 5. The STM functions now only serve to store information in the shared memory

Algorithm 4: Implementation of strategy S_1

```
   // 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 ← GetNextFreeSTMStruct()

   // Start transaction tx
 6 upon Start(tx)
 7     tx.thr ← STMSharedStructPtr()
 8     tx.thr.tx ← tx
 9     tx.thr.extensions ← 0

   // Commit transaction tx
10 upon Commit(tx)
11     tx.thr.tx ← 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, Q is the OS ordered ready queue
16 upon Schedule(Q)
17     t ← CurrentThread()
       // Check that the thread uses the STM and is in a transaction
18     if t.sh_stm ≠ null ∧ t.sh_stm.tx ≠ null then
19         if t.sh_stm.extensions ≤ N then
20             t.sh_stm.extensions ← t.sh_stm.extensions + 1
21             return                          // The current thread keeps running
22     Elect(First(Q))
```

segment that will be needed by Schedule to perform the corresponding operation. Start extracts the element of the shared memory segment associated with the thread and initializes the *tx* field in this element to the current transaction id. Conflict logs the transaction with which there is a conflict and its supporting thread, and then aborts the current transaction and yields the processor. Finally, Commit sets the *tx* field of the shared memory segment element to null, indicating that the transaction has ended, and the *commit* field to true.

We next present the specific kernel level code used to implement S_2 and S_3.

S_2 implementation

The implementation of S_2 adds to the kernel's thread structure a field *sh_stm* holding the thread's element of the shared memory segment and the field *wait* implementing the thread's wait queue for aborted threads. The function KernelSTMInit, which is called by a thread during the STM initialization process, initializes these fields.

The Schedule function handles any commit or conflict for the current thread. If the current thread is using the STM (line 11) and has its *commit* flag set (line 12), then the *commit* flag is cleared (line 13) and all of the processes on its wait queue are returned to the OS's ready queue (lines 14-16). It can, however, occur that a thread will commit one transaction and start another within a single time slice. The Schedule function thus tests on line 15 that the stored

Algorithm 5: STM-level code common to the implementations of strategies S_2 and S_3

```
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 ← STMSharedStructPtr()
8      tx.thr.tx ← tx

   // Conflict between tx and tx'
9  upon Conflict(tx, tx')
10     tx.thr.conflict_tx ← tx'
11     tx.thr.conflict_thr ← tx'.thr
12     Abort(tx)
13     Yield(tx.thr)

   // Commit transaction tx
14 upon Commit(tx)
15     tx.thr.tx ← null
16     tx.thr.commit ← true
```

conflict transaction is different from the current transaction of the committing thread, if any, so that threads in conflict with the current transaction remain in the wait queue. Next, if the thread is in conflict (lines 17-18), 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), to avoid introducing cycles that could cause deadlock. Of these operations, only the accesses to the various wait and ready queues require locks, as these queues can be accessed by multiple concurrent threads.

S_3 implementation

The implementation of S_3 adds to the kernel's thread structure a field *sh_stm* holding the thread's element of the shared memory segment and the field *conflict_count* recording how many threads the current thread conflicts with. As for S_2, the function KernelSTMInit initializes these fields.

The Schedule function again handles any commit or conflict for the current thread. In the case of a commit, Schedule clears the row and column of $\mathcal{C}$ associated with the current thread. As for S_2, it checks that a conflict is with a transaction other than the current one of the thread before clearing the corresponding entry of $\mathcal{C}$ (line 16). When an entry is newly cleared, the *conflict_count* of the associated thread is decremented (line 18). If *conflict_count* reaches 0, the thread has no more conflicts and it is returned to normal priority (lines 19-20). Except for the adjustment of *conflict_count* and the thread priority all of these operations can be performed without locks, as they only concern locations associated with the current thread. Finally, the priority and *conflict_count* of the committing thread are reset to normal and 0, respectively (lines 22-23). Next, if the thread is in conflict (lines 24-25), and it has not previously been in conflict within this transaction (line 27), then *conflict_count* is incremented and the priority of the thread is set to low. In any case, the identifier of the conflicting transaction is stored in $\mathcal{C}$.

Algorithm 6: Kernel-level code used in the implementation of strategy S_2

```
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 ← GetNextFreeSTMStruct()
6      tx.sh_stm.thr.conflict_thr ← null
7      tx.sh_stm.thr.commit ← false
8      t.wait ← InitQueue()
   // Thread election, Q is the OS ordered ready queue
9  upon Schedule(Q)
10     t ← CurrentThread()
11     if t.sh_stm then
           // Commit
12         if t.sh_stm.commit then
13             t.sh_stm.commit ← false
14             foreach thread t' in t.wait do
                   // Unblock previously conflicting threads
15                 if t'.sh_stm.conflict_tx ≠ t.sh_stm.tx then
16                     MoveToReadyQueue(t')
           // Conflict
17         t' ← t.sh_stm.conflict_thr
18         if t' ≠ null then
19             t.sh_stm.conflict_thr ← null
20             if OnReadyQueue(t') then
21                 MoveToWaitQueue(t, t'.wait)
22     Elect(First(Q))
```

5 Evaluation

To evaluate our scheduling strategies, we use a collection of micro-benchmarks, as well as a set of realistic applications. The first micro-benchmark, linked-list, commonly used to evaluate STM implementations, is an integer set implemented via a sorted linked list. Each execution consists of read transactions, which determine whether an element is in the list, and update transactions, which either add or remove an element. The list is initially populated with a given number of elements and its size is maintained constant by alternating insertions and removals.

The second micro-benchmark simulates a mix of short and long transactions. It models a simple bank application with two transaction types: (1) *transfer*, i.e., a withdrawal from one account followed by a deposit to another account; and (2) *balance*, i.e., a computation of the aggregate balance of all accounts. The transfer transaction type is short and contains two read and two write accesses. The balance transaction type is long and contains only read accesses (one per account). We experiment with workloads containing only transfers, and with a mix of transfer and balance transactions.

The realistic applications are taken from the STAMP [3] benchmark suite, which is together with STMbench7 [8] the most widely used STM benchmark. The reasons for choosing STAMP over STMbench7 are twofold. First, STMbench7 was designed for object-based STMs and the reference implementation is written in Java (although there also exist a C++ version now), while STAMP

Algorithm 7: Kernel-level code used in the implementation of strategy S_3

```
1  C[*][*] ← null                                                  // Conflict matrix
2  struct thread_extension
3      sh_shm : struct stm_thread
4      conflict_count : int
   // Kernel initialization of shared data structures
5  upon KernelSTMInit(t)
6      t.sh_stm ← GetNextFreeSTMStruct()
7      tx.sh_stm.thr.conflict_thr ← null
8      tx.sh_stm.thr.commit ← false
9      t.conflict_count ← 0
10 upon Schedule(Q)
11     t ← CurrentThread()
12     if t.sh_stm then
           // Commit
13         if t.sh_stm.commit then
14             t.sh_stm.commit ← false
15             foreach thread t' such that t'.sh_stm ≠ null do
                   // Clear column
16                 if C[t'][t] ≠ null ∧ C[t'][t] ≠ t.sh_stm.tx then
17                     C[t'][t] ← null
18                     t'.conflict_count ← t'.conflict_count − 1
19                     if t'.conflict_count = 0 then
                           // Reset priority if no more conflicts
20                         t'.priority ← NORMAL
21                 C[t][t'] ← null                                  // Clear row
               // Reset priority
22             t.priority ← NORMAL
23             t.sh_shm.conflict_count ← 0
           // Conflict
24         t' ← t.sh_stm.conflict_thr
25         if t' ≠ null then
26             t.sh_stm.conflict_thr ← null
27             if C[t][t'] = null then
28                 t.conflict_count ← t.conflict_count + 1
29                 t.priority ← LOW
30             C[t][t'] ← t.sh_stm.conflict_tx
31     Elect(First(Q))
```

is written in C and uses word-based STMs. Second, STAMP provides a larger variety of applications and real-world workloads than STMbench7, which only considers a specific type of data structures and accesses, albeit quite versatile.

We use six of the eight STAMP applications:[2] `genome` takes a large number of DNA segments and matches them to reconstruct the original source genome; `intruder` emulates a signature-based network intrusion detection system; `kmeans` is an application that partitions objects in a multi-dimensional space into a given number of clusters; `labyrinth` executes a parallel routing algorithm in a three-dimensional grid; `ssca2` constructs a graph data structure using adjacency arrays and auxiliary arrays; and `vacation` implements an online travel reservation system. Additionally, two sets of parameters are recommended by the developers of STAMP for `vacation` and `kmeans`, for producing executions with low and high contention. The single-threaded execution time

[2] The remaining two (`bayes` and `yada`) have bugs, which have been reported to the developer but were not fixed by the time we made our experiments.

of STAMP applications takes from a few seconds to several minutes depending on the benchmark and parameters. Table 1 summarizes the characteristics of the transactional workloads produced by these applications.

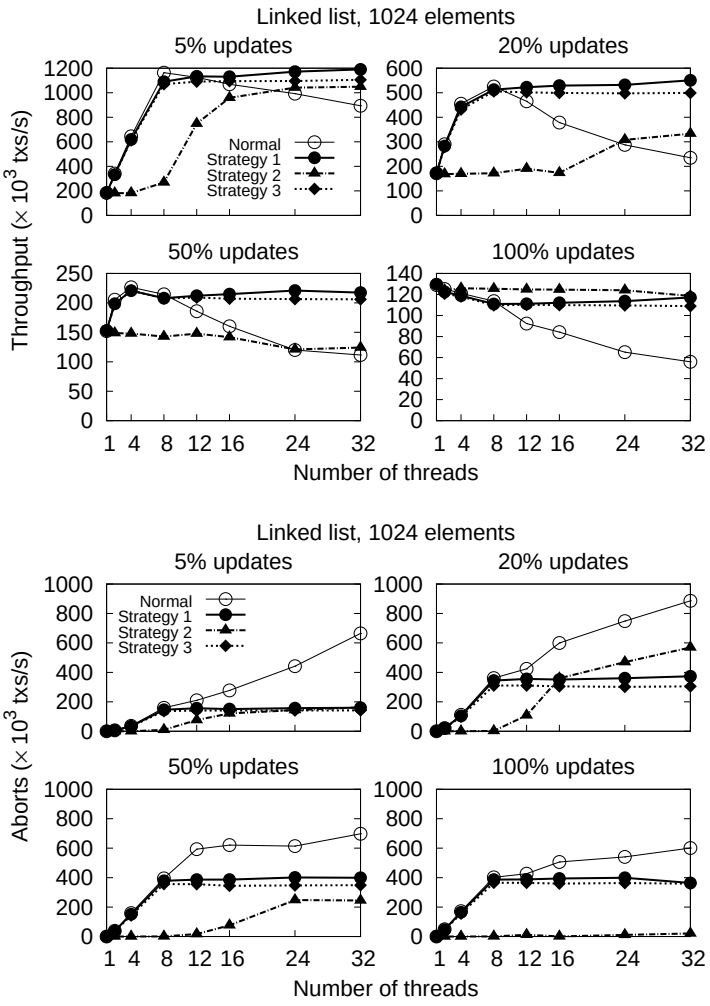

Figure 3: Performance of strategies S_1, S_2, and S_3 for the linked list micro-benchmark. The graphs show the commit rate (top) and abort rate (bottom) for executions with 5%, 20%, 50%, and 100% update transactions.

All benchmarks were run on a machine with two quad-core 3 GHz Intel Xeon processors (X5365) and 5 GB of main memory running Linux 2.6.25 SMP (64-bit). Up to 8 threads can execute concurrently. Benchmarks were compiled using gcc version 4.2.3 with the `-O3` optimization option. For strategy S_1, we used the value 255 for parameter N, which in practice corresponds to an unlimited number of extensions for our benchmarks.

The main metrics observed in our tests are the commit rate (throughput) and abort rate. For the STAMP applications, we also consider the scaling, i.e.,

Application	Tx length	R/W set	Tx time	Contention
`genome`	medium	medium	high	low
`intruder`	short	medium	medium	high
`kmeans`	short	small	low	low
`labyrinth`	long	large	high	high
`ssca2`	short	small	low	low
`vacation`	medium	medium	high	low/medium

Table 1: Characteristics of the transactional workloads of the STAMP benchmarks: length of transactions in terms of instructions, number of read/write accesses, time spent in transactions, and amount of contention (data from [3]).

the improvement factor as compared to an execution of the application with a single thread and no STM. Our general goal is to improve the behavior of STM when the number of threads is greater than the number of cores. Still, we do not want our strategies to induce an overhead when the number of cores is greater than the number of threads.

5.1 Linked List Micro-Benchmark

Figure 3 show the performance of the different strategies with the linked list micro-benchmark and four workloads corresponding to different update rates (and hence contention levels). Without any scheduling strategy (i.e., the normal case), the performance (in transaction throughput) drops when there are more threads than cores. At the same time, the number of aborts grows significantly. The intensity of this performance degradation increases with the rate of updates.

With respect to our goals, we notice that strategy S_1 induces almost no overhead when there are enough cores to accommodate all threads. The main benefit of S_1 is that that throughput remains flat once the number of threads exceeds the number of cores. The benefits of the S_1 are more noticeable when there is more contention, i.e., when there are more update transactions.

The number of aborts follows a similar trend: it remains flat when using S_1 while it grows without. This increased abort rate explains the degradation in throughput for the normal case.

Strategy S_2 is remarkably successful in limiting aborts, especially when there is high contention. Still, it is not that successful in term of throughput, except at high update rates where it performs better than the other strategies. The reason is that S_2 restrains too much the potential parallelism by assuming that if a transaction that has just aborted restarts, it will abort again. The poor performance of S_2 for even such a simple application suggests that the determinism assumption does not always hold.

Finally, S_3 releases the determinism assumption and allows restarted transactions to run at a low priority. It improves significantly on S_2 and performs similarly to S_1 on this benchmark.

5.2 Bank Micro-Benchmark

Figure 4 evaluates our strategies on the bank micro-benchmark and two workloads. One can observe that, with only short update transactions and little

contention (Figure 4 (top)), the improvements of all three strategies S_1-S_3 are remarkable both in terms of throughput and number of aborts when there are more threads than cores. The dramatic increase in number of aborts with the normal case is due to some transactions being preempted in the middle of a transfer and the conflicting transaction repeatedly aborting.

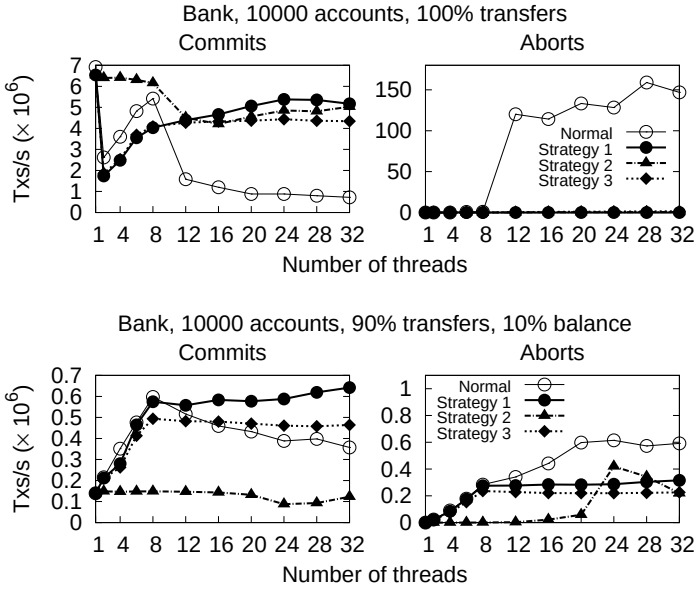

Figure 4: Performance of strategies S_1, S_2, and S_3 for the bank micro-benchmark with only transfer transactions (top) and with a mix of 90% transfer and 10% balance transactions (bottom). The graphs show the commit rate (left) and abort rate (right).

We observe an initial drop between single-threaded and multi-threaded execution. This due to the extreme nature of the workload, which only consists of short update transactions that rarely conflict. Upon commit, update transactions have to acquire a unique timestamp and thus generate much contention on TinySTM's shared clock. There are known techniques to address this problem (e.g., [4] and [16]) but we did not apply them here to allow for a fair comparison with the other benchmarks. One can observe that S_2 suffers much less from this problem since it limits the amount of parallelism.

Because the bank benchmark has very short transactions and little contention when there are fewer threads than cores, the communication overhead between the STM and the kernel makes up a larger part of the transaction time, thus leading to a slight observable overhead between the normal case and both S_1 and S_3.

With the second workload containing 10% long read-only transactions (Figure 4 (bottom)), we observe behavior comparable to linked list, with the exception of S_2 that does not scale. The reason is that S_2 may unnecessarily prevent short transfer transactions from running while a long balance transaction is ac-

tive. We also observe that S_1 is noticeably better in terms of throughput than S_3, but not in terms of aborts.

5.3 STAMP Macro-Benchmarks

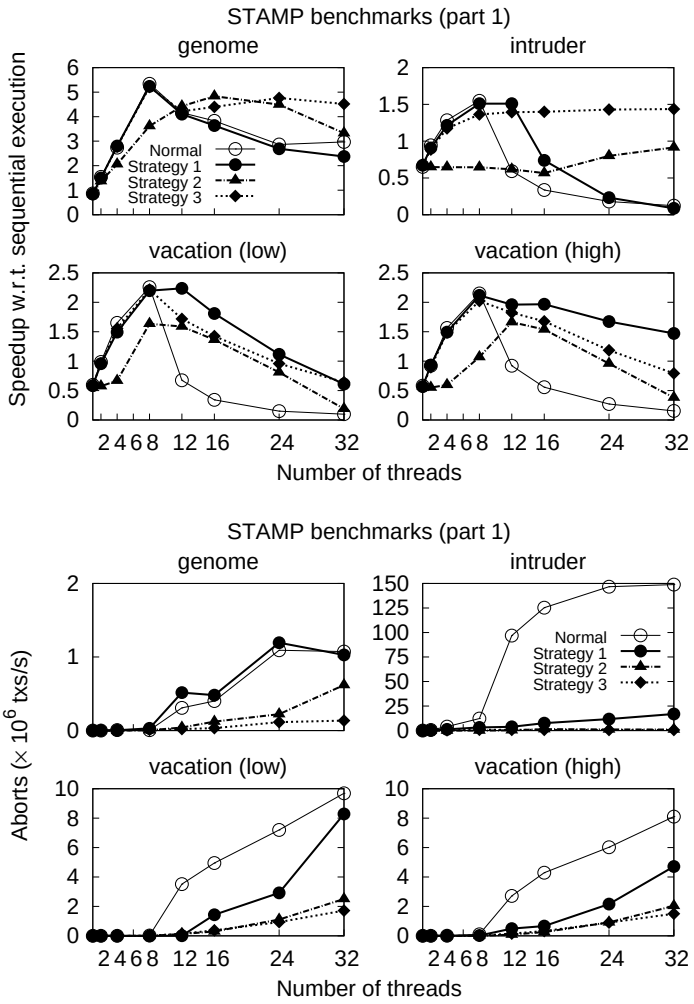

Figure 5: Performance of strategies S_1, S_2, and S_3 with the STAMP benchmarks (part 1). The graphs show the commit rate (top) and abort rate (bottom).

For the STAMP benchmarks (Figures 5 and 6), we can make several interesting observations. First, most benchmarks, with the exception of `labyrinth` and `ssca2`, suffer when there are more threads than cores. For `labyrinth` and `ssca2`, we observe that the number of aborts grows without penalizing throughput. This can be explained by the fact that the number of aborts remains very low compared to the number of commits.

Second, for benchmarks where throughput degrades with more threads than cores, S_3 provides the most steady throughput and abort rate of all strategies. This is remarkably clear in `genome` and `intruder`.

Third, the scheduling strategies do not help much in terms of throughput for `kmeans`, although they reduce the number of aborts. This is likely due to the short length of the transactions, which means that aborts are not costly and do not have much influence on the throughput.

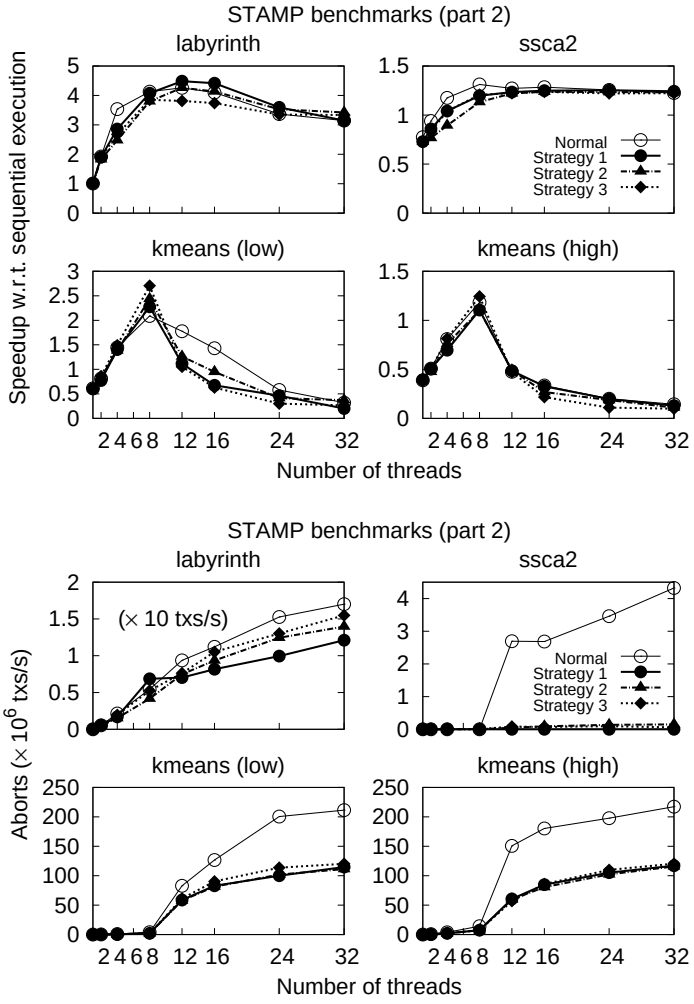

Figure 6: Performance of strategies S_1, S_2, and S_3 with the STAMP benchmarks (part 2). The graphs show the commit rate (top) and abort rate (bottom).

Fourth, S_1 is the most efficient strategy when there are no more threads than cores, and it systematically outperforms the normal case with additional threads.

Finally, we observe that S_2 is the most effective strategy overall in terms of reducing the number of aborts. However, it is consistently the least efficient

strategy with few threads. The same conclusion as for the linked list benchmark applies here; S_2 restrains too much the potential parallelism by assuming that if a transaction that has just aborted restarts, it will abort again. On the STAMP benchmarks that scale well, the determinism assumption does not hold.

5.4 In-depth evaluation of S_1

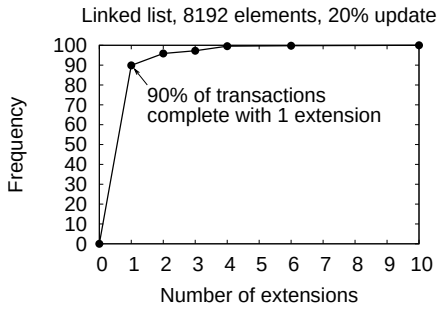

Figure 7: Number of extensions with strategy S_1 for the linked list micro-benchmark.

We have first evaluated the maximum number of extensions that are used in practice to execute completely a transaction. Results of Figure 7 show that 90% of the transactions that need an extension complete with only 1 extension. A limit of 10 is never reached in practice and could be used as maximum value for N.

We have also evaluated the effectiveness of S_1 in limiting the effects of other programs running on the same cores as the STM applications. For this, we have run the linked list benchmark simultaneously with another multithreaded application running on four cores. Four situations have been studied by varying the other application and the potential impact on the scheduler: a computation intensive application, another STM application (linked list) using the standard Linux scheduler, another STM application using S1, and an I/O intensive application. The results are presented in Figure 8. In the 4 cases, the linked list benchmark running alone (with and without S1) is shown as baseline.

The main result of these benchmarks is that when S1 is used, there is no degradation of performance due to the parallel application. The performance increases with the number of cores until the number of threads is greater than 4. Subsequently, there is a slight increase in performance as the number of threads increases, since more of the CPU time is devoted to threads that are running transactions. Without S1, the performance of linked list is always lower than the baseline curve (i.e., no parallel application) due to the increase in aborts when the application is suspended.

5.5 Overhead of Transaction Activation

To better understand whether our approach introduces overhead, we have measured how many times schedule is called during the execution of the linked

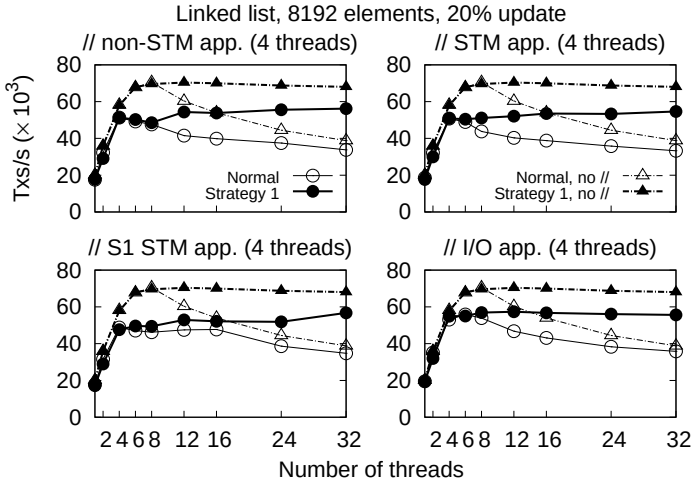

Figure 8: Performance of strategy S_1 for the linked list micro-benchmark with competition from a parallel application running on 4 threads: a computation intensive application, a second instance of the linked list benchmark with and without S1, and an application that performs many I/Os.

list benchmark (see Figure 9 (left)), and the overhead induced by the strategy within schedule (see Figure 9 (right)). The results are for the strategy S_3 which is the most expensive because it loops over all threads.

Figure 9 (left) shows that the number of calls to schedule depends on the number of cores used by the application, since there is no increase when the number of threads is greater than the number of cores.

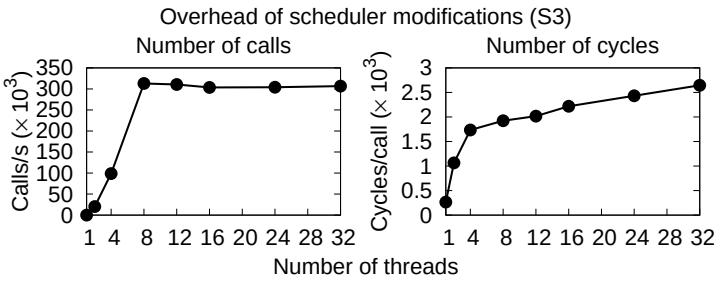

Figure 9: Schedule overhead

Figure 9 (right) demonstrates that the additional cost of schedule depends on the number of threads. Even though the increased cost is linear in the number of threads, the slope is quite low and the overhead has no impact on the application execution duration.

6 Related Work

This work was inspired in part by *scheduling activations* that allows a flexible mapping of user threads to processors [2] and by progress-based scheduling that permit the monitoring of application progress to take scheduling decisions [15].

In STM, contention managers [12, 13] are the standard approach to determine, when a conflict occurs, which transaction may proceed and which must wait or abort. Their main objective is to guarantee progress of the applicaton and prevent livelocks. As discussed in Section 2, application-level contention managers have several limitations, notably they lack precision and have no (or limited) control on the scheduling of threads. Contention managers also take a reactive approach, by only acting once a conflict has been detected. In contrast, our approach can also prevent conflicts from happening by controlling when a thread can execute or be preempted.

Yoo and Lee [17] have proposed to introduce a simple scheduler in the STM that essentially serializes transactions once a high level of contention is detected. This basic approach is effective in specific settings where parallelism actually degrades perfomance.

TL2's implementation on Solaris [4] uses the *schedctl* mechanism[3] to request short-term preemption deferral during the commit phase. Like our strategy S_1, 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, and for which preemption would increase the chances of an abort.

CAR-STM also takes a scheduling-based approach to approving STM contention management [5]. It proposes a strategy very similar to that of S_2, but implemented at the user level. In our work, we have instead chosen to implement our approach at the OS kernel level, to take advantage of its existing scheduling capabilities. The strategy S_1 furthermore cannot be implemented at the user level, because it relies on limiting the time extension to a fixed number of time slices, and the expiration of a time slice is not visible at the user level.

TxLinux [10] 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. We instead support STM on commodity hardware and we target user applications. Our approach requires only localized changes to the operating system.

7 Conclusion

In this paper, we have presented a system approach to STM contention management. For this, we have presented three strategies that adapt the scheduling behavior of the system to try to reduce conflicts between transactions. These strategies either try to prevent a transaction from being interrupted, to reduce the chance that conflicting transactions will execute concurrently, or delay threads that encounter a conflict, until the transaction causing the conflict has

[3] This strategy is not used in TL2's x86 implementation on Linux as the *schedctl* mechanism is not supported.

committed. Our approach is implemented in the scheduling function of the operating system, to take advantage of existing mechanisms for electing, pausing, and restarting processes. These changes require adding a total of less than 1000 lines of code at the beginning of the kernel schedule function for all three strategies. The changes do not have any impact on the existing scheduling logic.

S_1 and S_3 are clearly the winning strategies, but neither of these two performs better than the other. Since S_1 targets primarily inter-application issues, while S_3 targets intra-application ones, we plan to investigate the possibility of combining these two strategies, perhaps choosing dynamically between S_1 and S_3. Because a thread may execute the same transaction code many times, e.g., in a loop, another line of possible future work is to maintain a history of conflicts over multiple transactions. Such a history could be used to identify threads that are likely to conflict and prevent them from running concurrently.

References

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

[2] Tom Anderson, Brian Bershad, Ed Lazowska, and Hank Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of SOSP*, October 1991.

[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] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[5] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of PODC*, August 2008.

[6] Robert Ennals. Software Transactional Memory Should Not Be Obstruction-Free. Technical report, Intel Research Cambridge, 2006. IRC-TR-06-052.

[7] P. Felber, T. Riegel, and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of PPoPP*, February 2008.

[8] 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.

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

[10] 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*, October 2007.

[11] 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*, 2006.

[12] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, Jul 2004.

[13] W.N. Scherer III and M.L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of PODC*, Jul 2005.

[14] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of PODC*, Aug 1995.

[15] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *OSDI '99*, 1999.

[16] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based Transactional Memory with Scalable Time Bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2007.

[17] R. M. Yoo and H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of SPAA*, 2008.

Contents

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399