RenPar'20 / SympA'14 / **CFSE 8**
Saint-Malo, France, du 10 au 13 mai 2011

Kernel-Assisted Scheduling and Deadline Support for Software Transactional Memory

Walther Maldonado[1], Patrick Marlier[1], Pascal Felber[1], Julia Lawall[2], Gilles Muller[3], Etienne Rivière[1]
1 : Université de Neuchâtel, Suisse, 2 : DIKU, Danemark. 3 : INRIA, France.
Contact author : walther.maldonado@unine.ch

Résumé
Le concept de mémoire transactionelle (TM) vise à simplifier la programmation d'applications concurrentes. En particulier, le support logiciel de la mémoire transactionelle, ne nécessitant pas d'infrastructure matérielle spécifique, a été l'objet d'une grande attention ces dernières années. Une TM exécute des blocs de code dont les accès doivent apparaître atomique (transactions) de manière optimiste et résoud les conflits lorsque ceux-ci sont détéctés, avec l'aide d'un gestionnaire de contention (CM). Nous présentons dans cet articles deux approches pour améliorer la performance des TM, fondées sur un support au niveau du noyau Linux. La première approche propose des CMs collaborant avec l'ordonnanceur de tâches, et la deuxième propose la mise en œuvre de modes d'exécution adaptatifs pour le support de contraintes temporelles sur la terminaison des transactions. Nos résultats sont validés par une mise en œuvre au sein de TinySTM et par une évaluation à l'aide d'applications synthétiques et réalistes.

Mots-clés : Mémoire transactionelle, Support noyau, Ordonnancement, Échéances.

1. Introduction

Transactional Memory (TM) is a recent paradigm for programming concurrent applications. In particular, *Software Transactional Memory (STM)* has become promising due to its hardware independent design [1, 4, 6, 9, 14, 24]. Using a TM library, programmers can define code blocks as *transactional*, and the library will take care of executing them concurrently and atomically, lifting from the programmer the burdens of lock-based design. In case of a conflict between concurrent transactions (e.g., write sets that overlap), one of the transactions must be *aborted* and its execution be restarted ; this decision is performed by a *Contention Manager (CM)* [13].
TM relies on the premise that, in practice, contention is low, and under such settings it performs and scales remarkably well [1, 14]. In contrast, high-contention leads to many aborts, and longer transactions are less likely to commit. The resulting drop in performance is one of TM's main drawbacks.
This paper presents two approaches to improving performance : one is based on the creation of new contention managers for handling scheduling of concurrent transactions, by serializing conflicting transactions ; and the other is based on modifying the *visibility* of transactions to boost their chances of committing. *Scheduling* of transactions aims to improve the raw throughput of applications, whereas visibility is meant for cases where the perceived *responsiveness* of an application is determined by some key sections being executed in a timely manner. Time responsiveness is expressed through the use of *deadlines* (i.e. : lapse of/point in time before which a certain code block must finish [28]). In particular, we aim at applications whose response time is bound by a periodic task or aggregator function, which consists of a mostly reads transaction on large number of elements (e.g. : frame rendering).

Additionally, we also explore the benefits of adding kernel-level support to our approaches, by modifying the underlying OS to support the STM library. We explore two approaches : implementing scheduling CMs in the kernel's scheduler, and enabling *time-slice extensions* to allow threads supporting pending transactions to execute beyond their normally allowed time by the scheduler.

These approaches were implemented in TinySTM (a lightweight, lock-based STM [9, 10]) running on the Linux OS. Our findings show that serialization as a way of scheduling conflicting transactions yields good results, though different CMs each have weaknesses under different scenarios. Visibility changes in order to meet deadlines, on the other hand, show overall good performance, yielding results as good as existing alternatives, and with reduced contention.

The contributions described in this paper are as follow :

– We propose CMs for handling transaction scheduling, both at the user and kernel levels.
– We propose a mechanism for supporting the enforcement of deadlines for transactions.
– We study the effectiveness of system calls versus shared memory in relation to kernel interactions.
– We demonstrate the impact and efficiency of the different approaches on various scenarios, using widely used benchmarks as well as recent application simulators. Results show that different scheduling CMs have scenarios and conditions under which they excel, and performance is highly dependent on transaction length and contention, while transaction visibility manages to fulfill deadline requirements with up to 99% of cases, while at the same time greatly reducing the rate of retries.

The rest of the paper is structured as follows : Section 2 details the different levels of serialization that can be used for our scheduler CM, while section 3 details the different visibility rules meant for deadline support. Sections 4 and 5 deal with the implementation specifics involving the STM library and the kernel code, respectively. Section 6 presents the experimental evaluation of our contributions ; while section 7 resumes related work. Finally, section 8 concludes the paper.

2. Transaction Scheduling

A contention manager (CM) implements a policy that decides, upon the detection of a conflict between two transactions, which transaction should be aborted and which should be favored. A CM is critical to guarantee that the overall application makes progress. Conventional CMs handle high contention by delaying the restart of the aborted transaction, often with an increasing exponential back-off value. However, this approach has proven to be ineffective in several test-cases [20, 21], as often it suffers from (i) too many aborts, often the case when a long-running transaction loses to shorter ones, (ii) lack of precision, as the exponential back-off can often lead to threads idling far longer than needed, and (iii) unpredictable benefits, as delaying a long transaction does not ensure its success on restart. For this reason, our attention turns towards *serializing*, which involves delaying the re-execution of a transaction until the *winner* transaction of a conflict commits.

The reasoning behind serializing is that, once two transactions conflict, it is likely that they will continue to do so if re-executed together, and thus parallel progress is not possible. Serialization algorithms can be categorized on how strictly the approach is followed : it can be hard, or soft, serialization.

2.1. Hard-Serialization

Hard-serialization ensures that an aborted transaction will not re-execute until the other has completed. Aborted transactions are enqueued until the winner transaction commits [5, 29]. This approach performs well when the two following criteria are met : (i) transactions are deterministic, and (ii) the overhead of handling the queue is low compared to the transaction's length. If transactions are not deterministic, then progress would be possible without enqueueing, which leads instead to unnecessary delays. A high overhead cancels out the benefits of serialization when a transaction can take less time to complete with a few retries, than it does without aborting (due to the serialization overhead).

2.2. Soft-Serialization

In cases where transactions are not deterministic, enabling the aborted transaction to re-execute, but at lesser priority, could improve performance as it allows other awaiting threads to execute first, and the aborted transaction still has a chance to commit later on.

Because a restarted transaction can conflict with many others, this approach would keep track of them using a matrix : upon first abort, a transaction records the thread/transaction it conflicted against, and lowers its priority before restarting. Further conflicts would only update the table. When all the conflicting transactions have committed (or the transaction itself has), its priority is returned to normal.

2.3. Yielding

Because maintaining a conflict table and changing the priority of transactions can be expensive, a simplified version of soft-serialization would simply invoke a kernel *yield* call, to yield to other tasks in the scheduling queue and re-execute last after all of their time-slices have expired. This approach has the advantage of being lightweight, yet its benefits may only be seen in cases where there are multiple tasks in each scheduler queue for the yield call to take effect.

3. Deadlines

Until recently [19, 22], TM systems haven't really dealt directly with user imposed deadlines. A transaction could either be assured fairness of executing (e.g., through a timestamp-based CM), it could be assured priority over others (priority-based CMs [21]), or it could be given guarantee of commit without aborting (inevitability/irrevocability of transactions [23, 26, 27]). However, these approaches either lack prioritization support, or unnecessarily penalize other transactions.

A deadline is an imposed limit over the execution length of a task, and they can be categorized as either *hard deadlines* or *soft deadlines*. Hard deadlines are those in which failing to meet the deadline leads to a system failure. These are often used in machinery (e.g., in the electronics of a car's braking system), whereas soft deadlines deal with situations in which a missed deadline results in a temporary decrease of user-perceived responsiveness from the system. While hard deadlines are expressed only as the time by which work must complete, soft deadlines can also express an additional requirement : the expected *quality of the service (QoS)* (i.e. : how often the deadline desired has to be met in order to be considered as acceptable). Because hard deadlines require *a priori* knowledge of transaction durations, our approach focuses only on soft deadlines, which is more viable for a broader range of setups.

Our approach considers several execution modes for transactions. These modes have increasing costs on throughput and allowed concurrency, but allow for earlier conflict detection through greater visibility to the rest of the system. As a transaction progresses, and the time to deadline diminishes, execution modes get changed to increase the chance of committing.

3.1. Optimistic Execution

The base assumption behind TM systems is that most transactions will finish without a conflict. For our deadline support approach to work correctly, a significant number of transactions must be able to meet their deadline executing in *optimistic mode (OPT)*. This mode is the one using *invisible reads* as we describe in the next subsection. Otherwise, there wouldn't be a need to switch execution modes, and performance would be better off running directly with higher priority or irrevocability.

3.2. Visible Reads

Normally, each transaction keeps track of its read and write sets (memory locations accessed), and has access to the write sets of other transactions to detect conflicts. However, most STMs optimize their performance by keeping reads *invisible* to other transactions. In the case of a *write, then read* conflict,

the reader can detect it right away and invoke the CM. But, in the case of a *read, then write* conflict, it isn't detected until the reader is in the validation stage before committing. If the writer already committed, the reader can only abort, and this poses a risk to transactions approaching a deadline.

Enabling *visible reads (VR)* means that the TM library makes the read set accessible to the rest of the system, and a *read, then write* conflict can be detected immediately. In TinySTM this is implemented by using a single digit in the *ownership records*, to mark that there's a reader accessing the data. This approach allows only one reader at a time, which is more lightweight than previous approaches [12, 15].

3.3. Irrevocability

Inevitability, or *Irrevocability (IVC)*, is a TM feature that guarantees that once a transaction starts, it will commit without conflicts. It can be seen as a stricter version of hard serialization ; in order to ensure success, using irrevocable mode means that no other transaction can commit until the IVC one does.

4. STM Design

This section emphasizes the design and implementation decisions for STM library extensions. In order for our approaches to be non-intrusive on the code, they were implemented as modules, which are executed through callbacks to the key transactional events : START, COMMIT, and CONFLICT (abort).

4.1. Serialization

Two approaches were used for hard serialization : *system calls* and *spinning*. Threads can use *system calls* to signal the kernel that the calling thread must wait for an event (WAIT), and for resuming waiting threads (RELEASEALL), as shown in algorithm 1. However, this approach can be expensive in the case of short transactions, because of the RELEASEALL call in each commit.

One alternative is to use spinning, which is to wait on a *spinlock* until the other transaction commits. Spinning has the advantage of not imposing any overhead on the other transaction, and the waiting time is precise. However, it wastes CPU cycles, which can be expensive when there are other tasks waiting to be executed. TinySTM already implements *spinlock*-based retrying through its *delay* CM.

Soft Serialization was implemented as described in section 2 and is portrayed by algorithm 2. It uses a table and a counter for storing the conflict state between any two threads, and each entry holds a boolean value to determine if a conflict has occurred. The first time there is a conflict, the aborting transaction will lower its priority through a CHANGEPRIO system call. On each commit, the table is scanned and threads get their priority restored if their conflict counter drops back to zero.

In the case that the transactions are relatively short, it is possible to use a simplified version of soft serialization, by replacing the tracking and re-prioritizations with a single call to YIELD. This *yield* CM is described in algorithm 3.

4.2. Deadlines

In order to support deadlines, we need to (i) know in advance the duration of the transaction, and (ii) measure accurately elapsed time since the transaction started, in order to know how much time remains before the deadline is reached.

For the type of applications considered, the execution time for each atomic block varies little from call to call, when left undisturbed by other threads or kernel interruptions. Therefore, the past execution time of a transaction can be used to estimate the duration of future instances. Uniform sampling over a stream of data can be effectively done using Vitter's reservoir sampling algorithm [25], which over time gives us an appropriate model for the distribution of the transaction execution lengths. Vitter's algorithm (shown in figure 1) behaves as follows : the initially empty reservoir of size n is filled with the first n samples. Afterwards, each element k^{th} is inserted in a random spot of the reservoir with a probability of n/k. This

Algorithm 1: Hard Serialization

```
   // 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      WAIT(tx.thr, tx'.thr.wait )                          // Serialize after winner

   // Commit transaction tx
6  upon COMMIT(tx)
7      RELEASEALL(tx.thr.wait )
```

Algorithm 2: Soft Serialization

```
1  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 ¬C[tx.thr][tx'.thr] then
8          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 C[t][tx.thr] then
14             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         C[tx.thr][t] ← false                             // Clear row
19     CHANGEPRIO(tx.thr, NORMAL)                           // Reset priority
```

Algorithm 3: Yield Soft Serialization

```
   // 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
```

gives us an uniform sampling over the stream of data. The expected duration of a transaction is acquired by sorting the reservoir and picking the top percentile of interest. For instance, if our *QoS* is 95%, then the 95^{th} percentile is used as target time, thus giving us the duration upper bound for 95% of cases.

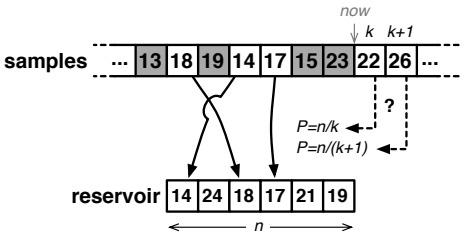


FIGURE 1 – Vitter's reservoir sampling [25].

To measure elapsed time, the time stamp counter (RDTSC instruction on x86 processors) was used, as it provides the least overhead. However, timestamps shouldn't be trusted between cores, as there is no guarantee that they will be synchronized. Therefore, samples that migrated core since execution start are not to be included in the reservoir. Upon an abort, a transaction renews its elapsed time and, depending on the remaining time before the deadline, may switch its executing mode. In determining the thresholds between switching modes (OPT, VR, or IVC), one has to consider that the execution time can vary between them. That is, transactions using VR take longer than in OPT mode, and this has to be taken into account. To keep calculations simple, we assume that IVC execution is of the same duration as OPT, while VR takes twice as long. The validity of these conservative estimates is confirmed in section 6.

Figure 2 visualizes the considered requirements. The time ST_{IVC}, at which a transaction must switch to IVC from VR, must be far enough from the deadline to enable a VR transaction to start executing right before ST_{IVC}, abort right before it commits, and restart in IVC, while still meeting the deadline. If we name L the expected transaction length during using OPT (as taken from the reservoir), then $3 \times L$ should be enough to meet the deadline. However, restarting in IVC can involve a delay while other concurrent transactions are committing, so the value used is $4 \times L$ instead. Similarly, the time between ST_{VR} (switching to VR mode) and ST_{IVC} should be enough to allow transactions to try VR at least once before switching to IVC. Assuming that an OPT transaction can start right before ST_{VR} and abort right before L, it follows that ST_{IVC} should be longer than $L : 2 \times L$.

5. Kernel Support

The proposed CM for serialization interacts with the kernel scheduler, as aborted transactions need to be removed from the execution queues to be reawakened later. However, communication with the kernel is normally done through system calls, which can be expensive in the presence of short transactions or low contention (in regards to the RELEASEALL call on commit). One way to avoid this expense is by communicating directly using a shared memory region (see figure 3).

The kernel interface is exposed to the system through a device file (`/dev/stm`), and through I/O control and memory mapping calls the shared region is initialized. The mapped region represents an array of

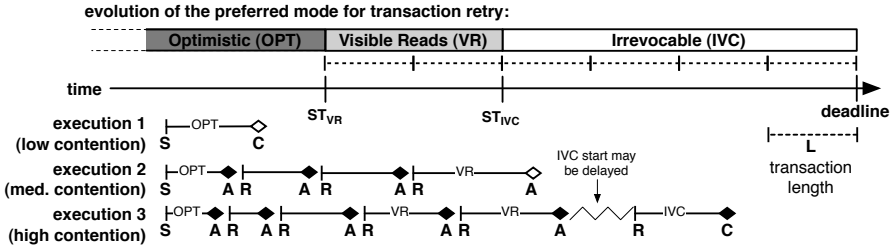


FIGURE 2 – Transaction execution mode switching for a transaction under increasing levels of contention, and times for changing execution mode before the deadline. S, A, R and C respectively denote `start`, `abort`, `retry`/failed commit, and successful `commit` operations.

entries, from which each running thread can claim ownership of one position. Communication towards the kernel is achieved then by writing to these entries directly, rather than using system calls. In turn, because the kernel is not interrupted to handle changes to the shared region, it must periodically poll the shared region for changes, at times where it can perform actions of interest (such as handling an invoked YIELD or performing the scheduling routine).

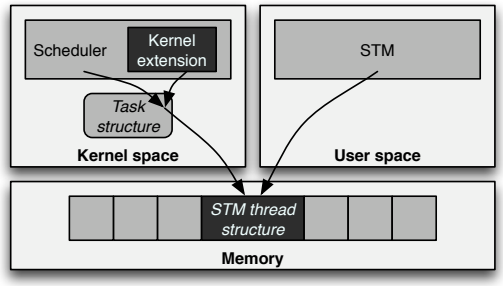


FIGURE 3 – Kernel/STM interaction.

When using the kernel module, the STM system only needs to write information on the shared region to indicate the execution state. In particular, in an abort it must specify the data for the *other* transaction (thread and transaction ID). Since the transaction that aborted needs to be delayed, an explicit YIELD will hand control over to the kernel, which will then perform the necessary tasks.

Serialization. Hard Serialization in the kernel is implemented through the use of linked lists, so that aborted transactions can be effectively removed from the run queue and inserted into a waiting list belonging to the other transaction. Handling YIELD calls will add elements to the queue, while on the scheduler, a pass will be given over executing transactions to release any enqueued threads when the thread's current transaction was marked as committed.

Soft Serialization is very similar to the user-space version, handling an inner table of states. The difference is that the change of priorities is handled by the kernel, rather than user-space. So it is done periodically during each scheduling period.

The main advantage of using the kernel to handle serialization is that the work on the user side is diminished, while kernel side work gets enqueued, and executed in batches during scheduling. However, the polling delay from the kernel can also diminish performance when, for instance, transactions could have committed multiple times during the time they spent enqueued in the kernel, since they did not get awakened until the next schedule time (up to one time-slice duration).

Deadline support. As mentioned in section 4, time stamp measurements cannot be relied upon if the thread supporting the transaction has been migrated between cores during execution. Using the kernel module, it is possible to forbid threads from being migrated while they are executing a transaction, thus ensuring accurate time stamp measurements.

Time-slice extensions. It may happen that, while a transaction is running, its execution time-slice expires, causing the kernel to suspend it until its next turn in the queue. This raises the duration of the transaction, which increases the chance of conflict, and may even cause a missed deadline. Therefore, using the kernel support it is possible to provide time-slice extensions, which grants tasks (in mid-transaction) up to N additional execution time-slices, N being typically small (e.g. : one or two). To keep system fairness, the thread will yield (at commit time) when it detects that an extension was granted.

6. Experimental Evaluation

All runs were done on an AMD Opteron server with four 2.3 GHz quad-core CPUs (16 cores total) and 8GB of RAM. The scheduling related tests were run on Linux 2.6.30, while the deadline-specific ones were run on version 2.6.34 (the kernel related code has undergone no changes between these versions). Our kernel module works with the default CPU scheduler (CFS) which uses per-CPU task queues.

Transaction Lengths. Measuring accurately transaction lengths provide us with information regarding (i) the type of transactions and their distribution, and (ii) the impact on the transaction execution time of running transactions in VR or IVC over OPT mode. The measured applications were taken from the STAMP benchmark suite, which includes various applications that simulate realistic scenarios. `bayes`

App.	Block	90th percentile			99th percentile		
		OPT	VR	IVC	OPT	VR	IVC
genome	1	11.90μs	*+8%*	*-2%*	13.20μs	*+9%*	*-1%*
	2	0.38μs	*+28%*	*-6%*	0.48μs	*+22%*	*-3%*
	3	1.28μs	*+17%*	*-2%*	233.37μs	*+47%*	*-3%*
	4	0.80μs	*+7%*	*-10%*	0.94μs	*+9%*	*-11%*
	5	0.99μs	*+10%*	*-3%*	1.20μs	*+9%*	*-2%*
intruder	1	0.38μs	*-3%*	*-14%*	0.45μs	*-3%*	*-14%*
	2	16.63μs	*+14%*	*-20%*	39.91μs	*+8%*	*-20%*
	3	0.17μs	*+1%*	*-3%*	0.43μs	*+8%*	*-3%*
kmeans	1	2.97μs	=	*-1%*	3.12μs	=	*-1%*
	2	0.20μs	*-21%*	*-12%*	0.22μs	*-2%*	*+2%*
	3	0.35μs	*-36%*	*+14%*	7.27μs	*-37%*	*+26%*
labyrinth	1	1.78μs	*+12%*	*+25%*	2.10μs	*+9%*	*+21%*
	2	543.6ms	*-2%*	*-2%*	633.7ms	*+2%*	*+2%*
	3	1.60μs	*+31%*	=	1.60μs	*+31%*	=
ssca2	1	12.41μs	*-21%*	*-7%*	12.41μs	*-21%*	*-7%*
	2	2.09μs	*+51%*	*+55%*	2.09μs	*+51%*	*+55%*
	3	0.88μs	*+1%*	*+2%*	0.98μs	*+1%*	*+3%*
vacation	1	31.45μs	=	=	35.75μs	*+1%*	=
	2	71.21μs	=	*+1%*	113.80μs	=	*+1%*
	3	19.99μs	=	=	24.68μs	*+1%*	=
yada	1	0.72μs	*+23%*	*+7%*	4.72μs	*+5%*	*+2%*
	2	0.28μs	*+28%*	*+36%*	0.40μs	*+23%*	*+24%*
	3	67.60μs	*+11%*	*-3%*	92.89μs	*+12%*	*-4%*
	4	0.18μs	*+25%*	*+48%*	0.24μs	*+19%*	*+34%*
	5	0.85μs	*+33%*	*+4%*	4.54μs	*+8%*	*+2%*
	6	0.49μs	*+8%*	*+10%*	0.49μs	*+8%*	*+10%*

TABLE 1 – STAMP : Transaction lengths w/ different execution modes.

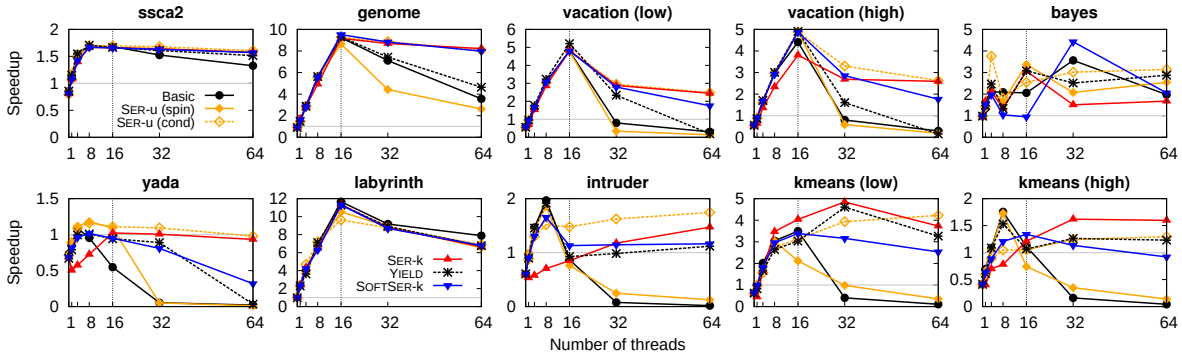
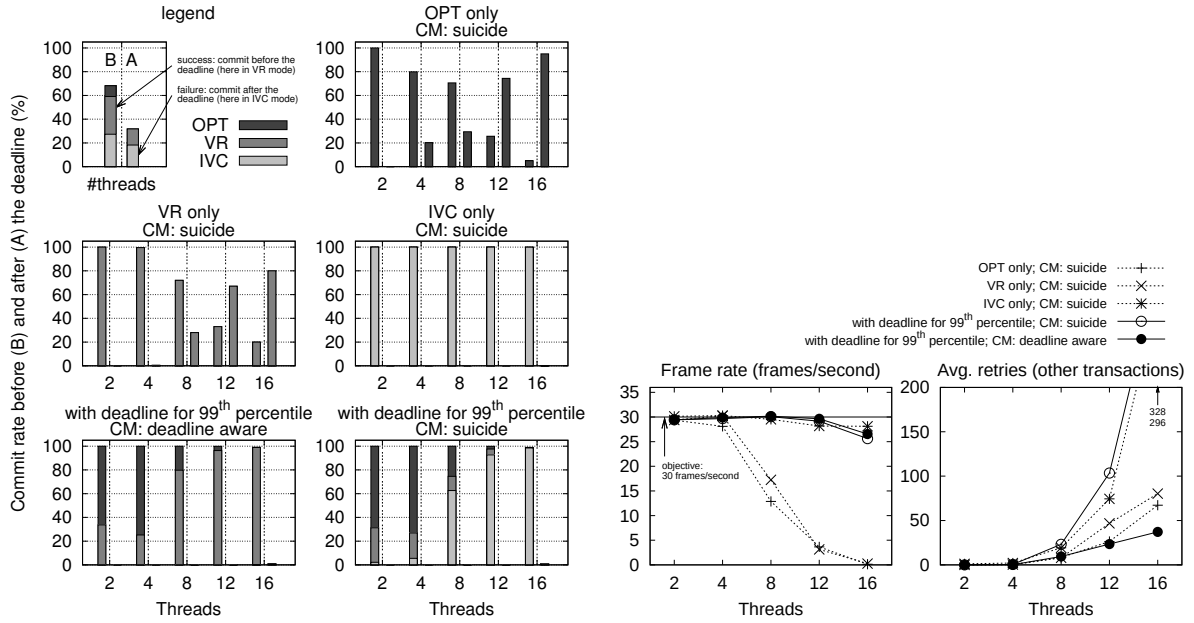


FIGURE 4 – Speedup of the STAMP benchmarks as compared to single-threaded execution.

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 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.

Figure 5 consolidates the transaction durations for the different benchmarks, while in table 1 the breakdown for each individual transaction is provided (`bayes` is omitted for space restriction reasons), as well as the effect of running in VR or IVC. The duration of the latter are expressed as a percentage relative to the OPT execution, which confirm our original hypothesis regarding transaction lengths :that the transaction running time in the VR and IVC can be deducted with reasonable precision based on the running time in the optimistic mode.

Serialization in the STAMP suite. In figure 4, the different CMS are contrasted against each other in the different STAMP benchmarks. They have varying degrees of performance depending on the workload. BASIC is the baseline against which to compare. It is TinySTM using the Suicide CM(the transaction that detects the conflicts aborts), without exponential backoff before retrying. SER-U (SPIN) is user-space se-



(a) *Swarm* : deadline success rates

(b) *Swarm* : impact on contention & frame rate

FIGURE 6 – Execution modes and deadline fulfillment rates for the *Swarm* application, and associated retry rate and display framerate. Baselines : OPT-, VR- and IVC-only means that the transaction for which we set a deadline executes in this mode, while the others all run in OPT.

rialization, using spin-locks. It tends to have the highest performance when the number of threads is less than the number of cores, but it unnecessarily wastes CPU cycles in order to achieve this result. That is also the reason why it tends to perform poorly when using more threads, often performing even worse than BASIC. SER-U (COND) uses pthread signals. It yields good performance in high contention scenarios with multiple threads, however its overhead makes it perform poorly in cases with low contention, specially when using less threads than cores. SER-K is kernel-space serialization. In many cases it is the best performing CM, however in cases where contention is lower, the kernel's delay with dealing with committed transactions lowers performance. YIELD performs very well in the simple case, with short transactions and few threads. Beyond that, its naive implementation makes it hardly better than BASIC. SOFTSER-K is Soft-Serialization, in the Kernel. Since deprioritizing causes the current thread to yield, it performs similarly when the number of threads is low. Beyond that, it does not perform particularly better or worse in the different workloads.

Deadlines and reactive applications. For evaluating deadline support, two reactive simulations were considered, *Swarm* [23] and *Synquake* [16]. In this paper, we present results for *Swarm* only. Our findings are similar with *Synquake*. *Swarm* [23] is an OpenGL based rendering application from the RSTM distribution. It performs asynchronous updates of a 3 dimensional scene, with one thread dedicated exclusively to rendering the current scene to a buffer which is then displayed on-screen. The rest of the threads perform the physics calculations for the scene objects. In order to support a target number of *frames per second*, the rendering thread was modified to invoke a periodical update (rather than updating as often as possible). This is also the thread for which a deadline has been

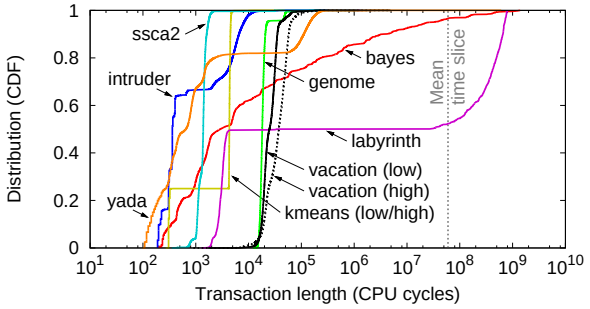


FIGURE 5 – Transaction lengths for STAMP.

associated. We consider a target frame rate of 30 images per second, with a deadline set to be 1/6th of the rendering period, that is, 1/180s from the beginning of the transaction. The objective given to the deadline enforcement mechanism is to achieve a *QoS* of 99% (i.e. : 99% of render frames must finish before their deadline).

We consider two CMs : *suicide*, where the transaction that detects the conflict aborts, and a *deadline-aware* CM that only differs in one aspect : transactions with a deadline will be given priority in case of conflict, if they are in VR mode. Intuitively, this second CM will ensure that long transactions with deadlines will be more likely to commit before entering IVC mode than with the *suicide* CM.

Figures 6(a) and 6(b) show the performance results for *Swarm*. Figure 6(a) presents the success rate in committing the rendering transaction before the deadline, using different execution modes including baselines : each such baseline consists in executing the transaction with a deadline entirely in OPT, VR, or IVC mode. Figure 6(b) presents the corresponding application-level performance figures : the framerate must reach 30 frames per second, while the progress of other transactions is impaired by their retry rate, shown in the second plot.

Our results are positive : with either CM, the deadlines are met up to the objective of 99% hitrate except for the case of a fully congested system where the performance drops to 98%. Even though using directly IVC would yield these results as well, our approach induces up to three times less contention (seen in the abort rate of the other transactions).

We also measured the number of time-slice extensions that were used at the kernel scheduler level. In the 16 thread case, a successful rendering transaction required an extension in 1.5% of the cases, while 0.1% required two extensions (none required more). 1.5% is a significant amount, considering that deadlines can be missed only 1% of the time, showing the necessity of the time-slice extension mechanism and kernel support in ensuring time constraints for transactions.

7. Related Work

CAR-STM [5] also uses a scheduling-based CM, which maintains per-core transaction queues. The approach is similar to hard-serialization in user space, but aborted transactions see their threads migrated to the core of the thread of the other conflicting transaction, thus causing all enqueued transactions to execute serially in regards to each other.

Yoo and Lee [29] presented a simple user-level adaptive scheduler which can serialize transactions once high-contention is detected. Ansari et al. [2] propose Steal-on-abort, which is a transaction scheduler which enables threads to "steal" conflicting transactions, thus making them execute serially. These approaches are performed entirely in user space and don't use the kernel's scheduling capacities.

Dragojevic et al. [7] presented another user-level transactional scheduler, *Shrink*, which performs scheduling decisions based in past access patterns. Serialization is done similarly to that of Yoo and Lee [29].

RT-STM [19] is an extension of Fraser's STM [11] meant to support real-time transactions (it has been integrated to the LITMUS^{RT} real-time operating system [3]). The modifications to Fraser's STM boil down to updating the logic behind "helping" other transactions, so that higher-priority transactions are helped by the lower-priority ones. This change applies only to the commit-phase, and as such a high priority cannot guarantee that the transaction will not be aborted before committing.

Fahmy et al. present an algorithm for computing upper bounds to response times of transactions in real-time systems [8], however they don't actually address implementing a real-time STM.

There are several approaches for implementing irrevocability/inevitability, some of which have been proposed and compared before [23,26,27]. However, they incur a high cost on other transactions, and do not consider an adaptive mechanism based on deadlines.

In previous work, an operating system supported scheduler was proposed which serializes conflicting transactions [18], and deadlines where considered later in [17]. This article is a synthesis of these. To the best of our knowledge, there are no proposals on adaptive execution modes for STM transactions.

8. Conclusion

In this paper we have proposed various scheduling oriented contention managers, implemented both in user-space and kernel-space, to improve throughput in systems under stress, as well as visibility-based deadline support for reactive applications. Direct support from the operating system where possible has also been considered and tested. Our approach was implemented on TinySTM, running on Linux.

Scheduling CMs have proven to yield varying results depending on the workload. For instances with low contention and/or small transactions, a simple yield often performs best, whereas hard serialization (either in user space or kernel space) work best for cases with high/varying contention. However, kernel level serialization has poor performance in middle-ground scenarios, due to the kernel being too slow to react to committing transactions.

Our deadline approach relies in existing methods to accurately measure elapsed time, as well as properly sample past execution times to predict duration of future runs. Furthermore, the kernel is able to provide support to overcome the innate shortcomings of our approach by controlling for thread migrations and time-slice interruptions. Experimental evaluation proves that our approach performs well, keeping the required success rate of 99% in all but the most congested scenarios. It performs as well as forcing irrevocability on the transactions of interest, while impact on other transactions is much lower.

Bibliographie

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. John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. LITMUS^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers. *RTSS'06 : 27th IEEE International Real-Time Systems Symposium*, pages 111–126, 2006.
4. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC'06 : 20th International Symposium on Distributed Computing*, pages 194–208, 2006.
5. Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM : scheduling-based collision avoidance and resolution for software transactional memory. In *PODC'08, 27th Annual ACM Symposium on Principles of Distributed Computing*, August 2008.
6. Aleksandar Dragojević, Rachid Guerraoui, and Michał Kapałka. Stretching transactional memory. In *PLDI'09 : ACM SIGPLAN Conference on Programming Languages Design and Implementation*, June 2009.
7. Aleksandar Dragojevic, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing : Avoiding conflicts in transactional memories. In *PODC'09, 28th Annual ACM Symposium on Principles of Distributed Computing*, pages 7–16, August 2009.
8. Sherif Fadel Fahmy, Binoy Ravindran, and E. Douglas Jensen. On bounding response times under software transactional memory in distributed multiprocessor real-time systems. In *DATE'09 : Conference on Design, Automation and Test in Europe*, 2009.
9. Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08 : 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246, 2008.
10. 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.
11. K. Fraser. Practical lock-freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, 2004.

12. Maurice Herlihy. SXM : C# Software Transactional Memory. Unpublished manuscript, Brown Univ. http://www.cs.brown.edu/~mph/, May 2005.
13. 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.
14. James Larus and Christos Kozyrakis. Transactional memory. *Communication of the ACM*, 51(7) :80–88, July 2008.
15. Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT 2009 : 4th ACM SIGPLAN Workshop on Transactional Computing*, feb 2009.
16. Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *EuroSys'10 : 5th European conference on Computer systems*. ACM, 2010.
17. Walther Maldonado, Patrick Marlier, Pascal Felber, Julia L. Lawall, and Gilles Muller. Deadline-aware scheduling for software transactional memory. In *DSN'11 : Proceedings of the 41st annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Hong Kong, P.R. China, jun 2011.
18. Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *PPoPP'10 : 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–90, 2010.
19. Toufik Sarni, Audrey Queudet, and Patrick Valduriez. Real-time support for software transactional memory. In *RTCSA'09 : 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 477–485, 2009.
20. 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.
21. 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.
22. Martin Schoeberl, Florian Brandner, and Jan Vitek. RTTM : Real-time transactional memory. In *SAC'10 : 25th ACM Symposium on Applied Computing*, March 2010.
23. Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and exploiting inevitability in software transactional memory. In *ICPP'08 : 37th International Conference on Parallel Processing*. IEEE CS, 2008.
24. Jaswanth Sreeram, Romain Cledat, Tushar Kumar, and Santosh Pande. RSTM : A relaxed consistency software transactional memory for multicores. In *PACT'07 : 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE CS, 2007.
25. Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1) :37–57, 1985.
26. Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. xCalls : safe i/o in memory transactions. In *EuroSys'09 : 4th ACM European conference on Computer systems*.
27. Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08 : 20th annual symposium on Parallelism in algorithms and architectures*, pages 285–296. ACM, 2008.
28. Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline : first class support for interactivity in commodity operating systems. In *OSDI'08 : 8th USENIX conference on Operating systems design and implementation*, pages 73–86.
29. Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of SPAA*, pages 169–178, June 2008.