Deadline-Aware Scheduling for Software Transactional Memory

Walther Maldonado*, Patrick Marlier*, Pascal Felber*, Julia Lawall†, Giller Muller‡, Etienne Rivière*
*University of Neuchâtel, Switzerland. Email: first.last@unine.ch
†DIKU, Denmark. Email: julia@diku.dk
‡INRIA/LIP6, France. Email: gilles.muller@lip6.fr

***Abstract*—Software Transactional Memory (STM) is an optimistic concurrency control mechanism that simplifies the development of parallel programs. Still, the interest of STM has not yet been demonstrated for reactive applications that require bounded response time for some of their operations. We propose to support such applications by allowing the developer to annotate some transaction blocks with deadlines. Based on previous execution statistics, we adjust the transaction execution strategy by decreasing the level of optimism as the deadlines near through two modes of conservative execution, without overly limiting the progress of concurrent transactions. Our implementation comprises a STM extension for gathering statistics and implementing the execution mode strategies. We have also extended the Linux scheduler to disable preemption or migration of threads that are executing transactions with deadlines. Our experimental evaluation shows that our approach significantly improves the chance of a transaction meeting its deadline when its progress is hampered by conflicts.**

***Keywords*-Transactional Memory, Scheduling, Contention Management**

I. Introduction

Transactional Memory (TM) is a recent paradigm for designing parallel programs that scale with the number of cores. In particular, *Software Transactional Memory (STM)* systems have received great attention due to their hardware-independent design [1], [2], [3], [4]. STM systems optimistically handle synchronization through the use of atomic blocks of code with transactional semantics: atomic blocks are executed concurrently and, upon conflict, they may need to roll back and restart. 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.

Among the applications that may benefit from the high level of concurrency allowed by transactional memory, some are subject to constraints on their reactivity. Examples include live rendering (e.g., in 3D modeling applications) and video games, where a set of modifications should result in the updated scene being displayed in a time compatible with the user's perception. Another example is that of a server running a transactional DBMS where update tasks should succeed but still allow queries, which may refer to many objects, to be answered in parallel within a reasonable amount of time. Implementing such reactive applications using transactional memory results in particular properties for the transactions composing the application. In particular, these applications often rely on rendering and aggregation mechanisms that use a *read-mostly* transaction, whose read set is expected to grow large as many data elements are accessed in order to compute the aggregate (e.g., an image). The read-mostly transaction is often a long one, executed periodically, but not as frequently as other transactions, such as updates.

To achieve reactivity in practice, a majority of the instances of the read-mostly transactions must succeed before a given deadline. This majority must be quantifiable, that is, a rate of success for committing before deadlines is set as a target, and enforced by the system. The deadline may be expressed by the application developer by specifying the maximum delay until commit, or by a fixed point in time by which a transaction must have committed [5]. To ensure that deadlines are respected, support at the runtime level is required. So far, the support of constraints on processing time in the context of STMs has been achieved by executing the transactions that are not allowed to abort in an irrevocable (also called inevitable) mode [6], [7], [8]. However, if one transaction runs in this mode, all other transactions are unable to perform writes during its execution. Therefore, systematically executing the read-mostly transaction as an irrevocable one would severely limit parallelism and would reduce the performance of the whole application.

In this paper, we present an approach stemming from the practical observation that, in many cases, the read-mostly transaction can be executed optimistically without being aborted if the level of contention is low enough. This suggests that irrevocability should be used only when the deadline nears and restarting would lead to a deadline violation, implying that aborting is no longer an option. We thus propose to execute the read-mostly transaction under an adaptive scheme depending on the remaining time before the deadline. More precisely, we have designed a strategy based on three execution modes: *optimistic*, *visible read* and *irrevocable*. These execution modes differ in the way conflicts are detected or avoided: the optimistic mode defers the detection until commit time, the visible read mode is able to inform the other transactions that a particular location has been read and can be subject to a conflict, and finally the irrevocable mode prevents the transaction from aborting upon conflict by disabling the accesses from other transactions that can lead to a conflict. The visible read mode differs from the irrevocable one in that aborting is still possible; this mode allows more transactions to run in parallel and incurs

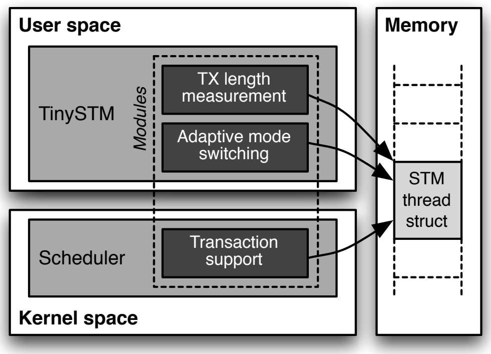


Figure 1. Deadline-aware transaction scheduling framework.

less degradation of the application performance. Finally, our runtime system ensures that a thread executing a transaction with a deadline is not preempted by the operating system scheduler by granting up to a few time slice extensions if required.

In order to decide which mode the read-mostly transaction should run under at a given time, we need an oracle that provides the execution time of each transactional block. For this purpose, we continuously sample the distribution of the read-mostly transaction's duration. The sampled distribution is then used as an indication of the expected maximum duration for a given percentage of instances of the transaction.

Our approach has been implemented in the TinySTM STM library [2] and the Linux OS (see Figure 1). The implementation relies on two modules that extend TinySTM with the adaptive transaction mode and with the measurement of transaction lengths. A third module extends the Linux kernel scheduler so as to control thread migration and preemption. Communication between the STM runtime and the kernel module is implemented using a shared memory region to keep the overhead minimal [9].

Our achievements are as follows:

- We propose mechanisms to support deadline-aware scheduling for reactive applications based on TM. The developer contract on a transaction subject to a deadline is met even when the number of concurrent update transactions reaches high levels. Our approach is able to achieve a constant frame rate of 30 images per second on a 3D interactive simulation renderer application [7], with guarantees on the rendering period regularity. On a game server engine [10], we are able to set constraints on the reactiveness of an expensive query accessing a large portion of the game state.
- Thanks to the adaptive execution mode strategy and the associated contention management mechanisms, we can achieve the same success rate in respecting the deadlines as an approach that systematically uses irrevocability, while imposing a much lower overhead on the system. Our success rate consistently lies between 98% and 100% of that observed when using only irrevocability but our adaptive strategy reduces by 3 times the average number of retries needed by other transactions in the highest contention case.
- We present and evaluate how to use sampling to effectively determine the running time of transactions in a multi-threaded setting with the help of kernel-level mechanisms. We show that these samples can be used as an oracle for execution mode switching: the deviation in the transaction length given by the oracle is at most -4% to +12% for typical examples of long-running transactions from the STAMP benchmarks.
- We describe and justify experimentally our claim that using different levels of "pessimism" makes it possible to meet deadlines while adapting to the level of difficulty for the transaction to commit: our experiments validate that, as contention increases, commits with optimistic execution modes are replaced by commits by more deterministic execution modes only as required to meet the deadlines. For instance, for the game server engine with 2 update threads, transactions almost always succeed in the optimistic mode, while with 4, 8 and 12 update threads the visible read mode is necessary to ensure committing before the deadline, respectively, 11%, 48% and 72% of the time. Finally, the irrevocable mode only starts being needed at 16 update threads.

The rest of the paper is structured as follows. Section II presents our sampling mechanism for accurately measuring transaction durations. Section III describes the design and implementation of the adaptive switching mode that allows a transaction to commit before its deadline. Section IV focuses on the kernel-level scheduling extensions that provide deterministic support for the threads running deadline-associated transactions. Section V presents a thorough evaluation of all proposed mechanisms. We present related work in Section VI and conclude in Section VII.

II. Measuring Transaction Lengths

The foundation of our approach is the ability to adapt the execution mode of a transaction based on the amount of time remaining before its deadline expires. For this, we need to have an oracle that can provide the expected transaction length. For the reactive applications considered in this paper, we have observed that the uninterrupted execution time of each atomic block statically apparent in the source code varies very little from one execution to the next. Therefore, the oracle can use the results of past executions.

For our solution to be transparent to developers and users, we implement the collection and management of the transaction lengths as a module in TinySTM. This module instruments the `start`, `abort` and `commit` operations so as to measure the execution time of a *successful* transaction, between its beginning and the corresponding commit. The

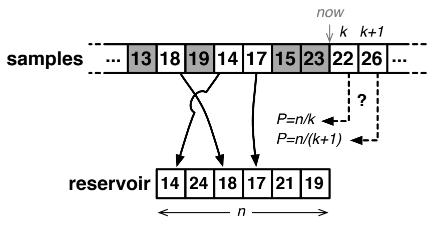


Figure 2. Vitter's reservoir sampling [11].

transaction length is obtained from the time stamp counter (`RDTSC` instruction of the x86 processor) of the core on which the `start` and `commit` operations are executed.

One of the challenges to address in implementing the measurement module is to know when to discard information about transaction executions that do not provide usable measurements. First, during the application warm up phase we are only interested in irrevocable mode executions in order to bootstrap the oracle, and later we are only interested in optimistic mode executions that are meant to be the common case execution. Second, aborted executions are ignored as we are interested only in successfully committed transactions. Third, we ignore any execution in which the supporting thread is migrated, as time stamp counters are not necessarily synchronized across cores. Finally, we ignore executions in which the supporting thread is preempted, as this artificially increases the transaction execution time.

We keep separate statistics for each atomic block that is subject to deadlines. To keep the memory overhead reasonable, these statistics are based on a subset of size at most n of all valid samples obtained during execution. The set is managed so that, at each point in time, it contains a uniform sample of elements from all measurements since the beginning of the program execution. Inline uniform sampling over a stream of events is achieved by using Vitter's reservoir sampling method [11]. This randomized algorithm maintains a set—or reservoir—of n elements. It systematically inserts the first n elements so as to fill the reservoir. Thereafter, each k^{th} sample from the stream of all valid samples is inserted in the reservoir with a probability n/k, replacing a randomly chosen element (see Figure 2). This strategy effectively maintains a uniform random sample subset of all values in the stream of observations. The warmup phase fills the reservoir with 1% of its capacity by systematically running the transaction in the most conservative execution mode (irrevocable).

We can obtain an estimate of a given quantile of the distribution represented by the reservoir by simply sorting the values and reading the corresponding position in the reservoir. For instance, the 99^{th} percentile is an upper bound on the execution time of 99% of the transactions that were sampled in the reservoir. In the applications considered in this paper, there is little to no variation in the distribution of transaction length over time. We can therefore use a single reservoir from the beginning of the execution. Applications for which the distribution changes over time may use a biased sampling strategy, e.g., that increases the likelihood of keeping recent samples, or at the extreme by using a simple sliding window.

III. Transaction Execution Modes

Our approach provides three different modes for running transactions: optimistic, visible read and irrevocable. These modes allow us to implement different levels of predictability for committing within a bounded amount of time, but have increasing costs in terms of overall throughput and contention. In this section we present their implementation in TinySTM.

A. *Design choices*

TinySTM, like several other state-of-the-art STM libraries (e.g., [1], [3]), relies on an optimistic execution mode with invisible reads and a timestamp-based algorithm for detecting conflicts. While this approach is very efficient in situations that induce few conflicts, it does not provide the predictability guarantees expected from applications with reactivity requirements:

1) As transactions use invisible reads, read/write[1] conflicts are not detected when they happen. A transaction might thus have to abort when discovering upon validation that it has read a memory location that has since been overwritten by another committed transaction.
2) Even without considering invisible reads, a transaction may abort an unbounded number of times because its writes conflict with those of other update transactions.

Both issues are problematic because, as transactions may repeatedly abort, one cannot easily bound their execution time. Priority-based contention managers [12] would not solve the problem because, with invisible reads, read/write conflicts are not detected as they occur.

By extending TinySTM with the visible read (VR) and irrevocable (IVC) modes, we make it possible to reduce *the level of optimism* of transaction execution and to increase its predictability. These modes, however, also reduce the level of concurrency achievable by other transactions. This is illustrated in Figure 3, which shows the compatibility of the different execution modes of transactions. Only the read-only part of an optimistic transaction can execute concurrently with an irrevocable transaction. Visible reads allow other non-conflicting transactions executing in visible read or optimistic mode to execute concurrently. Finally, the highest level of concurrency is achieved by optimistic transactions. Table I summarizes the main properties of the three execution modes, which are described in the rest of the section.

[1]We denote by "read/write" (or simply R/W) a conflict with the read happening before the write. W/R conflicts happen in the reverse order.

Name	Strategy	Execution	Conflicts detected	Abort on	Notes
OPT	Invisible reads	Optimistic	W/W, W/R	Access conflict Failed validation	Minimal runtime overhead Transactions may repeatedly abort despite CM
VR	Visible reads	Semi-optimistic	W/W, W/R, R/W	Access conflict	Limited runtime overhead (1 bit/orec, 1 CAS/VR) OPT- and VR-transactions can execute concurrently
IVC	Irrevocable	Pessimistic	W/W, W/R, R/W	—	OPT- and VR-transactions can execute concurrently, but their commit is delayed if they write to memory

Table I

SUMMARY OF THE THREE EXECUTION MODES FOR CONTROLLING THE DEGREE OF OPTIMISM OF TRANSACTIONS.

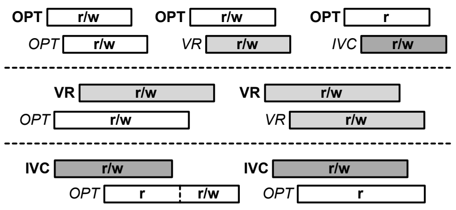


Figure 3. Compatibility of the three execution modes supported by TinySTM: irrevocable/IVC (top), visible reads/VR (middle), and optimistic/OPT (bottom). Rectangles represent transactions preforming reads (r) or arbitrary memory accesses (r/w). Each row corresponds to one thread and time flows to the right. Overlapping transactions can execute concurrently if they do not conflict.

B. Optimistic mode

To explain the new execution modes, we first recall TinySTM's basic optimistic mode [2]. TinySTM uses a global time base to build a consistent snapshot of the values accessed by a transaction. Transaction stores are buffered until commit. The consistency of the snapshot read by a transaction is checked based on versioned locks (ownership records, or *orecs* for short) and a global time base, which is typically implemented using a shared counter. The orec protecting a given memory location is determined by hashing the address and looking up the associated entry in a global array of orecs. A single orec may protect multiple memory locations.

Upon write, a transaction first acquires the locks that cover the updated memory locations by atomically setting a bit in the orec with a compare-and-set operation (CAS). In contrast, no update of the orec occurs upon read. At commit time, an update transaction obtains a new timestamp from the global time base by incrementing it atomically, validates that the values it has read have not changed and, if so, writes back its updates to shared memory. Finally, when releasing the locks, the versions of the orecs are set to the commit timestamp. Reading transactions can thus see the virtual commit time of the updated memory locations and use it to check the consistency of their read set. If all loads did not virtually happen at the same time, the snapshot is inconsistent.

Read-only transactions do not need to modify the global time base and can commit without updating any of the shared metadata (orecs). Therefore, read-only transactions execute very efficiently when there are no conflicts and do not slow down concurrent transactions. They can, however, abort at commit time because they have read values that have since been overwritten by another committed transaction. Note that the same problem arises with writes when using commit-time locking (lazy conflict detection), as implemented for instance by TL2 [1], because write-write conflicts are only discovered at commit time.

C. Visible read mode

Visible read mode (VR) allows an update transaction to detect read-write conflicts with a reader transaction. The motivation is to detect R/W conflicts as they happen, and thus to favor the reader in VR mode over other transactions executing in the optimistic mode. This allows any conflicting writer to back off and let the reader complete its execution. In our framework, this mode enable a read-mostly transaction to make progress while reducing the probability of an abort.

To implement visible reads, one may consider simulating a visible read by a write. This solution however would trigger R/R conflicts with OPT-transactions, and a VR-transaction would thus prevent OPT-transactions from committing and vice versa. Some STMs (e.g., SXM [13]) implement visible reads by maintaining a list of readers for each shared object. With such an approach, one can keep track at each point in time of the number and identity of the readers, and allow multiple readers or a single writer to access the object. Writer starvation can be prevented by letting readers "drain" as soon as a writer requests ownership of the orec. The main drawback of this approach is that it imposes a significant overhead for the management of the reader list and creates additional contention. To address these problems, SkySTM [14] implements "semi-visible" reads by just keeping a counter of readers for each memory location.

We propose an even more extreme approach relying on a single additional bit in the orecs to indicate that associated memory locations are being read by some transaction(s). The bit is atomically CAS'ed when reading the associated memory location for the first time. A single visible reader is allowed at a given time, and only if there is no writer—unless the writer is the same transaction that performs the visible read. Therefore, a visible reader behaves almost identically to a writer, with one major difference: *there is no conflict between a visible reader and a transaction accessing the same memory location optimistically*, i.e., with optimistic transactions (OPT) using invisible reads.

The rationale behind this design choice is that transactions will seldom use visible reads. In fact, in the applications considered in the paper, only the read-mostly transaction may be in the VR mode if it fails to commit in the optimistic mode. Still, our implementation supports concurrent executions of VR-transactions, even though the considered applications do not currently take advantage of it.

D. *Irrevocable mode*

In irrevocable (also called inevitable [7]) mode, a transaction is protected from aborts and will eventually commit. A simple implementation of irrevocable mode is to execute an irrevocable transaction alone once no other transaction is in progress (serial mode). While this approach is safe, it does not provide any concurrency and should be reserved as a fallback mechanism for special situations, e.g., for system calls that cannot execute safely inside a transaction.

A more promising approach is to allow an irrevocable transaction to execute concurrently with other non-irrevocable transactions. Several such algorithms have previously been discussed and evaluated [6], [7]. We propose a new variant.

First, as with visible read, our hypothesis is that irrevocability (IVC) will be used rarely and, hence, should coexist as well as possible with OPT-transactions. Our approach is limited in that it allows only one irrevocable transaction to execute at a time. Again, in our context, we do not anticipate that multiple IVC-transactions will need to execute concurrently.[2]

Our implementation follows the general design of previous approaches [6], [7], by using a global token that a transaction must acquire before it becomes irrevocable. Once the token has been acquired, no other update transaction may commit, independent of its execution mode. A transaction can request to enter irrevocable mode at any point in its execution. If the transaction has already accessed some shared object, it must validate its read set before irrevocability can be granted. Failed validation triggers an abort and the transaction directly restarts in the irrevocable mode. In the context of deadline-aware scheduling, we only enter irrevocable mode upon restart after a conflict is detected and when nearing a deadline.

Since an irrevocable transaction is guaranteed to never abort, in case of a conflict the other non-IVC transaction will systematically abort. To keep the implementation lightweight, we allow a read-only non-IVC transaction to commit while an IVC-transaction is in progress, but delay the committing of concurrent update non-IVC transactions until after the completion of the IVC-transaction. This approach permits non-conflicting transactions to execute concurrently while allowing for interesting optimizations in IVC-transactions: they do not need to use visible reads or to validate the timestamp of read values, or even to maintain a read set, resulting in a reduced overhead.

[2]Supporting multiple concurrent irrevocable transactions would require advance knowledge of the memory locations read and written by all irrevocable transactions. Otherwise, one can trivially construct an interleaving with just two transactions that leads to a deadlock.

IV. Deadline-based Transaction Scheduling

Our runtime system for achieving deadline-based transaction scheduling relies on a dynamic adaptation of the execution mode. This section presents how deadlines can be expressed by the developer and how the STM library chooses the execution mode for some atomic block. Finally, we describe two extensions to the contention manager and to the Linux scheduler to provide efficient deadline-based transaction support.

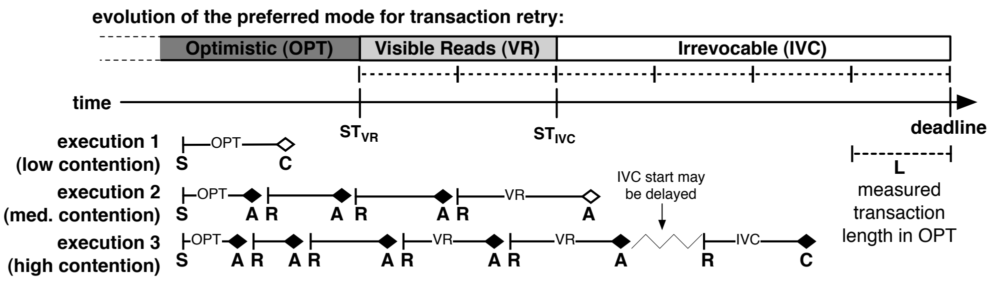
A. *Setting deadlines*

The deadline associated with an atomic block is a contract between the programmer and the STM library. As such, a deadline is a pair: a time relative to the beginning of the transaction and an associated guarantee level expected by the programmer. This level is expressed as a quantile of the distribution of transactions based on their execution length. For instance, the programmer may desire that 99% of the transactions for that particular atomic block commit in less than $1ms$: in this case the deadline will be expressed as the pair $(1ms, 99)$. The programmer can specify the deadline associated with a given atomic block as an optional parameter to the `start` API call or, when using compiler support, as an attribute of the `__transaction` statement. A deadline can be set at runtime and therefore an atomic block may have a different deadline for each execution.

B. *Dynamically adapting the transaction execution mode*

Adaptation of the execution mode of a transaction with a deadline is performed by a module that extends TinySTM. The adaption is done at transaction restart after detecting a conflict and aborting. The goal here is to satisfy the quantile of success q expressed by the developer when setting the deadline contract. The new running mode is chosen based on the transaction length estimation L and the time remaining before the deadline.

The transaction estimation length L for a quantile q is obtained from the transaction length distribution gathered in the reservoir by the measurement module (see Section II). The higher the quantile is, the longer is L. For instance, if the objective is that 99% of the transactions for an atomic block should commit in less than t milliseconds, L will be the 99th percentile value of the transaction length distribution as recorded in the reservoir. Using a higher value for q will result in more transactions committing by the deadline but will also result in using the visible read (VR) and irrevocable (IVC) execution modes more often. Conversely, using a lower value for q, such as the median of the reservoir, can result in deadlines being missed.



evolution of the preferred mode for transaction retry:
Optimistic (OPT)
Visible Reads (VR)
Irrevocable (IVC)
time
ST_VR
ST_IVC
deadline
execution 1 (low contention)
execution 2 (med. contention)
execution 3 (high contention)
OPT
VR
IVC
S
A
R
C
IVC start may be delayed
L
measured transaction length in OPT

Figure 4. 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.

Depending on the remaining time before the deadline, the transaction execution mode is set as follows (see Figure 4):

- after time $\mathrm{ST}_{\mathrm{VR}}$ and before $\mathrm{ST}_{\mathrm{IVC}}$, the transaction switches from optimistic mode to VR mode;
- after time $\mathrm{ST}_{\mathrm{IVC}}$, the transaction is executed in IVC mode. However, if another transaction is running in IVC mode, the restart will be delayed until that transaction commits.

To simplify the computation of the $\mathrm{ST}_{\mathrm{VR}}$ and $\mathrm{ST}_{\mathrm{IVC}}$ deadlines, we consider that the running time in IVC mode is of the same order as for optimistic mode. We also conservatively consider that the time allotted for running in the VR mode is twice the time for running in the optimistic mode. The validity of these simplifying assumptions is experimentally verified in Section V. The choices for these times are based on the following rationales:

- In order to ensure that the transaction gets at least one chance to commit in VR mode before switching to IVC mode, even when the transaction restarts in OPT just before $\mathrm{ST}_{\mathrm{VR}}$, we set $\mathrm{ST}_{\mathrm{IVC}} - \mathrm{ST}_{\mathrm{VR}} = 2 \times L$. This scenario is illustrated by execution 2 in Figure 4: a restart in OPT mode just before $\mathrm{ST}_{\mathrm{VR}}$ still gives the transaction the opportunity to restart—and commit—in VR mode.
- Similarly, a transaction starting in VR mode just before $\mathrm{ST}_{\mathrm{IVC}}$ must be allowed to restart and commit in IVC mode. Based on our assumption on the length of the execution time in VR mode, this transaction can restart at $\mathrm{ST}_{\mathrm{IVC}} + 2 \times L$. We also take into account a possible delay of at most L due to another pending IVC transaction. We thus set $\mathrm{ST}_{\mathrm{IVC}}$ such that $\text{deadline} - \mathrm{ST}_{\mathrm{IVC}} = 4 \times L$ ($2 \times L$ for a VR execution, and $1 \times L$ for the delay).

When the reservoir is empty or contains too few elements, we must make a conservative estimate for the value of L. As long as less than 10% of the reservoir is populated, we systematically switch the transaction to the IVC execution mode, so that the best time guarantee possible is achieved and the reservoir fills in rapidly. Once the number of samples in the reservoir grows to $f >$10% of its total capacity, we start selecting the value of L based on the reservoir's content, while conservatively considering that empty entries contain larger values than the ones already present. We consider two cases, depending on the quantile q associated with the deadline. If $q < f$ we select the q' th percentile value of all the samples present in the reservoir, with $q' = (q \times 100)/f$. Otherwise we use the maximum value (i.e., $q' = 100$).

C. Contention manager support

The default contention manager (CM) in TinySTM uses the *suicide* strategy, which is simple and effective in low-contention cases: a transaction that detects a conflict during execution or upon validation simply aborts and restarts. This strategy can be problematic for a long-running read-mostly transaction associated with a deadline. As the deadline nears, the transaction will switch to the VR execution mode, where read/write conflicts can be detected as they happen. If other updates transactions are running at the same time, using the *suicide* CM results in the read-mostly transaction being aborted with a high probability, eventually switching to the more costly IVC execution mode. We therefore extend the contention manager so that it gives priority to a transaction associated with a deadline and running in the VR mode, before resorting to the default *suicide* contention management strategy. We call this extended CM *deadline-aware*.

D. Scheduler support

We extend the Linux scheduler in two ways to ensure that the execution mode policy will be successful in ensuring transactions commit before their deadline.

First, threads supporting on-going transactions for which a deadline is set may reach the end of their execution slice, in which case they are at risk of being timed out and rescheduled much later in the future. This results in a likely abort of the OPT and VR modes, and a large increase in retry rate and thus execution time, as the risk of conflict grows during the interruption. For the IVC execution mode, this may cause other transactions to repeatedly abort because they cannot progress until the irrevocable transaction has completed. To accommodate deadlines, we need to make sure that the corresponding transactions are given enough time to commit even if they were started just before the end of a time slice. We implement time slice extension at the kernel scheduler level to deal with this issue: a thread running a transaction with a deadline can be granted up to three additional time slices to finish its execution. In this case, in order to be fair to other threads running on the system, the transactional library automatically detects the time slice extension and yields back the processor as soon as the transaction commits. In practice, only in a very few cases does a transaction need more than one extension to be able to commit.

Second, the load balancing mechanisms in the kernel may decide to migrate the thread from one core to the other while it is executing a deadline based transaction. This would induce a wrong time measurement since timestamp counters are not synchronized across cores, and prevent from implementing the execution mode change policy as we could not rely anymore on the elapsed time measurement after a core migration. In order to avoid this problem, the scheduler extension module forbids the migration of a thread running a deadline based transaction.

V. Experimental Evaluation

In this section, we perform a thorough evaluation of all components of the system. We demonstrate their ability to achieve the desired success rate for committing before deadlines, while keeping the overhead and impact on the overall throughput minimal. We first evaluate the transaction length measurement module and then illustrate the tradeoffs between the overall and individual throughput using each of the three different modes for a privileged transaction. We demonstrate our claim that the execution length of a transaction can be safely estimated based on the continuous sampling. Finally, we evaluate the ability of the adaptive mode switching module to make transactions commit before their deadline and the corresponding impact on the other transactions' throughput, using two realistic applications: *swarm* [7] and *synquake* [10].

A. Experimental setup and benchmarks

All tests have been carried out on an AMD Opteron server with four 2.3 GHz quad-core CPUs (16 cores in total) and 8GB RAM running Linux 2.6.34. Where applicable, we consider a reservoir size of 10,000 elements. All results in this section are averaged/aggregated over 10 runs of the following applications.

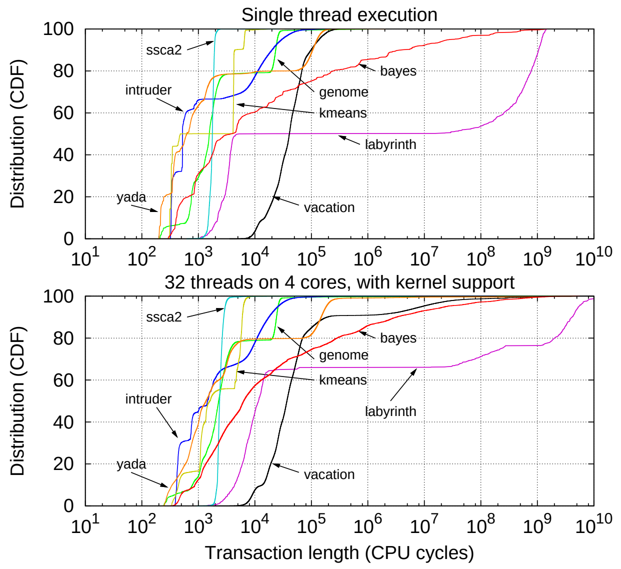
The `bank` micro-benchmark models a simple bank application performing various operations on accounts (transfers, aggregate balance, etc.). We consider a variant in which all transactions are read-mostly aggregations that consult the balance of 50 random accounts out of 10,000 and update one random account. We use this benchmark only for illustrating the inherent tradeoffs of our three execution modes in the presence of read-mostly transactions.

We also consider the STAMP [15] benchmark suite, one of the most widely used STM benchmarks. STAMP benchmarks present a large variety of transaction lengths and read/write set sizes. Henceforth, we use them to evaluate the correctness of our transaction length measurement module. `bayes` uses a hill-climbing algorithm that combines local and global search to learn the structure of Bayesian networks from observed data; `genome` matches a large number of DNA segments to reconstruct the original source genome; `intruder` emulates a signature-based network intrusion detection system; `kmeans` partitions objects in a multi-dimensional space into 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. Two sets of parameters are recommended by the STAMP developers for `vacation` and `kmeans`, to produce executions with low and high contention. We only consider the ones yielding a high contention. The single-threaded execution time of the STAMP applications ranges from a few seconds to several minutes.

We then consider two realistic applications, `swarm` and `synquake`, that are representative of the class of reactive applications targeted by our approach. `swarm` is a realistic rendering application from the RSTM distribution [7]. It performs asynchronous rendering and updates of a 3-dimensional scene graph. One thread is responsible for the rendering while the other threads are moving collection of objects from the shared state and detecting their collisions in the scene. In order to obtain a given number of frames per second, we have performed one slight modification to swarm so that all the rendering is performed in a single periodic transaction that is assigned a deadline. In our instances of *swarm*, the renderer always reads the whole set of 26,856 objects from the scene. `synquake` emulates a game server with multiple clients, interacting over a 2D map [10]. The execution is split into cycles, during which each thread applies a set of actions on behalf of the clients. Each action is associated with a scope and range, which respectively translate into an amount of transactional reads and writes. In order to generate a high level of contention that does not allow a majority of OPT transactions to commit directly, we use the following parameters. We use the "quest" (bias of players' moves towards a point of attraction) that yields the highest contention. We further increase contention by using 128 players, each being a 2x2 tile in a 64×64 tile map. We remove the map "walls" that could prevent interactions and thus reduce contention. We also increase the range of all actions. In particular, the longest transaction action (*attack*) operates on read and write sets of respectively 1,490 and 62 elements on average. This is the action for which we set a deadline in our experiment.

B. Measuring Transaction Lengths

We evaluate the ability of the transaction length measurement module to evaluate the duration of transactions,



application	variation	application	variation
genome	+5.30%	ssca2	+0.88%
intruder	+6.05%	vacation	+29,841%
kmeans	+3.01%	yada	+44,497%
labyrinth	+3,465%	bayes	+370,45%

Figure 5. Transaction lengths measurements in a single thread execution and content of the reservoirs for a 32 threads execution on 4 cores. The table lists the variations of the value of the 99th percentile of the distribution when no pruning of invalid samples due to migrations and preemptions is used.

while pruning out invalid measurements due to migration and preemption in a multithreaded setting. We consider all STAMP applications and build a reservoir for each atomic block of the application.

We first consider the accuracy of our mechanism for single-threaded runs. In this case, there is no contention as there is only at most one transaction at a time. Additionally, there are no interruptions of the thread as we use only one core out of 16. We show the cumulative distribution of transaction lengths in the uppermost plot of Figure 5. The other plot presents the contents of the reservoirs for all transactions, in a multithreaded setting. We use 32 threads on 4 cores in order to generate high levels of migrations and preemptions of the threads supporting the transactions. Unused cores are disabled and cannot be used by the operating system. Apart from the natural shift to longer transaction lengths due to cache misses and contention, we observe that the distribution of lengths follow the same trends as for a single threaded execution, and that potentially outlying measurements due to migration and preemption do not appear in the reservoirs. Note that disabling the pruning of invalid samples can result in a major shift towards extreme values for the last percentiles, making these values unusable for predicting the transaction length when not interrupted. This effect is shown by the relative difference for the 99th percentile value in the table at the bottom of Figure 5.

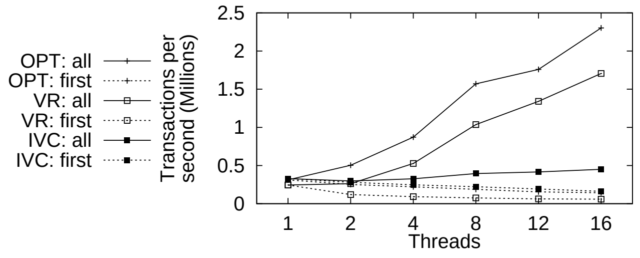


Figure 6. Illustration of the individual throughput of `bank` with a special thread ("first") running a transaction in OPT, VR, or IVC mode while other threads run transactions only in OPT mode, and impact on the application global throughput ("all").

C. Adaptive Transaction Execution Modes

We evaluate the different adaptive execution modes in two ways. First, we demonstrate the tradeoff between the gain in execution determinism for the transactions running in each mode and the relative impact on throughput. Then, we show that using transaction length predictions based on OPT execution for the other modes is accurate for read-mostly long-running transactions.

Determinism/throughput tradeoff: Figure 6 presents the individual and global throughput of `bank` where one of the thread (denoted *first*) can run its transactions in any of the modes while the other threads only run them in OPT mode. It illustrates the claim, made in Section III, that the impact of the three execution modes on the overall throughput is progressive: running the first transaction in VR mode only results in a 26% total throughput reduction for 16 threads compared to using OPT, while using IVC results in a severe reduction of 80% for the same settings. We also note that the individual throughput in VR mode is smaller: this is due to the overhead of marking visible reads during the transaction execution.

Predictable execution lengths: We characterize the difference in execution time of transactions when executed in the different modes (IR, VR, and IVC) for all STAMP applications, and for each separate atomic block in these applications. Table II presents the execution times for transactions running in OPT mode, and the relative difference for VR and IVC, for the 90th and 99th percentiles. These extreme values of the distribution are the ones that matter in our context, as these are the ones that will be used for setting the contract associated with a deadline. Note that the results are similar when considering the median of the distribution. For a majority of transactions, the difference between OPT and VR, and OPT and IVC, is very limited (less than $\pm 10\%$). The running time in VR mode is typically larger than for OPT mode due to the overhead imposed by setting the orecs with a CAS operation (see Section III-C), while the lengths of IVC transactions are smaller or comparable to that of OPT transactions. Exceptions mostly concern short-running transactions, where the cost of acquiring the global IVC

App.	Block	90th percentile OPT	90th percentile VR	90th percentile IVC	99th percentile OPT	99th percentile VR	99th percentile 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%*
bayes	1	9.94μs	*+13%*	*+28%*	9.94μs	*+13%*	*+28%*
	2	0.74μs	*+39%*	*+2%*	10.08μs	*+1%*	=
	3	0.54μs	*+17%*	*+6%*	1.09μs	*+36%*	*-17%*
	4	10.52μs	*+2%*	*-1%*	22.20μs	*+8%*	*-6%*
	5	8.54ms	*+1%*	=	31.32ms	*+1%*	*+1%*
	6	0.32μs	*+16%*	*+40%*	0.65μs	*+57%*	*+52%*
	7	311.76μs	*+1%*	*-1%*	29.36ms	*+2%*	*+3%*
	8	366.47μs	*+2%*	*+1%*	10.63ms	*+2%*	*+1%*
	9	0.25μs	*+27%*	*+24%*	9.85μs	*+23%*	*+1%*
	10	64.52ms	*+1%*	=	592.4ms	*+1%*	*+1%*
	11	16.05ms	*+1%*	*+2%*	226.8ms	*+2%*	*+2%*
	12	19.99ms	*+2%*	*+1%*	314ms	=	=
	13	1.15μs	*+31%*	*-13%*	9.87μs	*-22%*	*+2%*

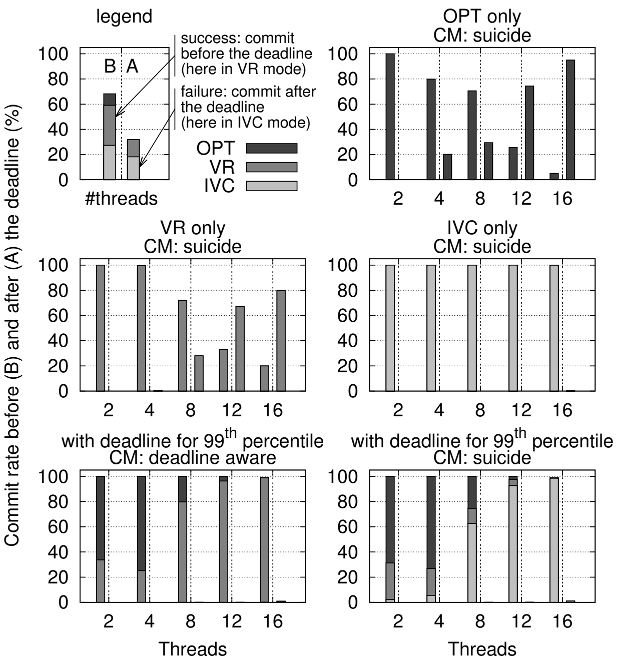
Table II
STAMP: TRANSACTION LENGTHS W/ DIFFERENT EXECUTION MODES.

token is higher than the gain observed for not having to maintain a read set with long transactions. For long-running transactions (such as labyrinth:2, yada:3 or bayes:5&10), predicting the execution time in VR and IVC modes based on OPT executions is accurate within a small error margin (resp. [-2%:+12%] for VR and [-4%:+1%] for IVC).

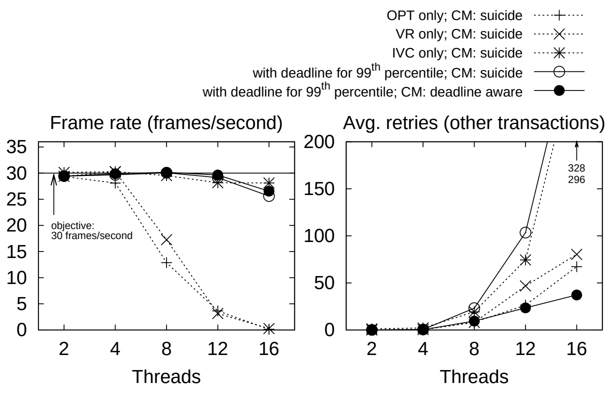
D. Deadline Aware Scheduling

We now evaluate the effectiveness of the combination of all our mechanisms in committing a transaction before a deadline, both for periodic tasks with the *swarm* rendering application, and for one-shot tasks with the *synquake* application, while keeping the impact on the system as low as possible. When setting deadlines, we use the 99th percentile for the associated contract. Using a contract of 100% would risk of selecting an unrepresentative outlying measurement as the value of L and be overly conservative in switching to pessimistic execution modes. Using the highest reasonable percentile also constitutes a worst-case scenario for contention and throughput, as the VR and IVC modes are then more likely to be used for respecting deadlines.

Periodic rendering — swarm application: The rendering transaction in *swarm* periodically compiles the objects that are then sent to an OpenGL rendering library (`glut`). As



(a) Execution modes and success of committing by the deadline.



(b) Frame rate and contention.

Figure 7. Performance and success rate of *swarm* executions with a deadline of 33% of the frame generation period (1/180 second). 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.

such, it basically performs a consistent snapshot of the scene while allowing concurrent execution of update transactions. We consider a target frame rate of 30 images per second. We set the deadline to be one sixth of a rendering period from the time the rendering starts, that is, 1/180s from the beginning of the transaction. At most one rendering starts per 1/30s period. Transactions that commit after the deadline are considered to be failures, while missed frames due to transactions that last for longer than a rendering period result in a drop in the effective frame rate.

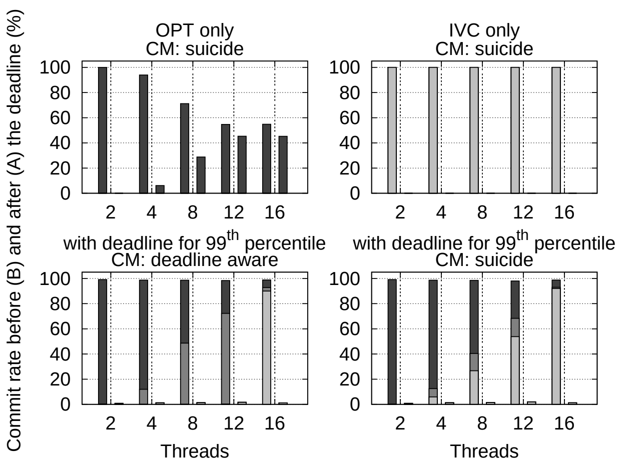
Figure 7(a) presents the success rates (commits before the deadline) for various execution modes, while the left graph of Figure 7(b) shows the corresponding frame rate. We also measure the impact on the overall throughput by observing the contention, measured by the average number of retries experienced by other transactions in the system, in this case update transactions (right graph of Figure 7(b)).

Running all transactions in OPT mode does not allow us to reach a reasonable frame rate for more than 4 threads. Note that for 8 threads, even if 60% of the rendering transactions succeed by the deadline, the ones that do not succeed typically last for multiple rendering periods, resulting in a drop to 10 frames per second. This highlights the unpredictability of transaction lengths when running in OPT-mode. Similarly, running the rendering transaction in VR mode only does not succeed in reaching the target frame rate, as transactions are aborting frequently, and also frequently span over multiple rendering periods. As expected, running this transaction in IVC mode is the ideal case for the success rate and the frame rate, but also leads to a large level of contention, as illustrated by the retry rate.

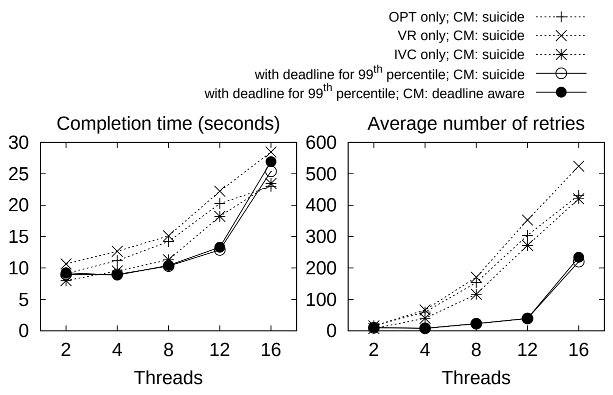
We now consider the results with a deadline contract on the 99th percentile value from the reservoir, using either the *deadline-aware* or the *suicide* CM described in Section IV. In both cases, we succeed in committing before the deadline in 99% of the cases for up to 12 threads, and 98% for 16 threads. We nonetheless observe that using the *deadline-aware* CM provides a significant advantage in many respects. First, with 16 threads, it achieves a better frame rate, meeting the objective of 30 frames per second, while using the *suicide* CM leads to a few long VR transactions that reduce the frame rate. Second, using the *suicide* CM results in a much larger number of deadlines being met by using the IVC mode, as the transaction typically aborts in VR mode whereas the *deadline-aware* CM would prioritize it over other transactions. Third, a corollary to the previous observation is that using the *suicide* CM results in a much larger contention in the system, as one has to pay the price of contention of an IVC transaction plus potentially many OPT and VR transactions before it. On the other hand, the *deadline-aware* CM not only meets a majority of deadlines using OPT or VR modes (a few IVC, not visible on the figure, are still necessary for part of the success rate for 8 threads and more), but also results in a contention that is *3 times lower* than using directly IVC with 16 threads, as illustrated by the average number of retries for other transactions in Figure 7(b) right.

Our final observation is on the time slice extension mechanisms provided by the transaction support kernel scheduler module. In the worst case (for 16 threads, with the *deadline-aware* CM), transactions seldom require more than one extension to be able to commit by the deadline. Out of 900 render commits, 1.5% required an extension, and 0.1% required two. None required three or more extensions.

Batch simulation — synquake *application:* We set a deadline on the time used for an attack operation performed by the first thread only, and set its deadline to be 10 times the 90th percentile of its execution as observed in sample runs for the same workload, that is, 750μs. Unlike swarm, operations in *synquake* are performed as fast as possible, which means that the impact of IVC transactions on throughput is likely to be more important.



(a) Execution modes and success of committing by the deadline.



(b) Completion time for the complete workload and contention.

Figure 8. Performance and correctness of synquake executions with a deadline of 750μs. The legend is shared with Figure 7

Figure 8(a) presents the success and failure rates for committing before the deadlines in the same way as for *swarm*. However, since *synquake* attempts to simulate a given workload as fast as possible, we consider the running time of the application instead of a frame rate for measuring the overall application throughput. We also consider the average number of retries for all transactions in the system as a measure of the contention for each run. Due to space restrictions, we do not present the results for VR-only for deadline success rate: they are on par with the observations made for swarm, that some more deadlines are met than in OPT but far from enough to satisfy the deadline requirements. We note though that the smaller transactions of *synquake* do not allow for the same difference in throughput and contention for the two different CM that we observe for *swarm* (where the rendering transaction is very long as it reads all objects). We also observe that the deadline-aware CM is as effective as pushing the use of VR transactions up to 12 threads. For 16 threads, and as the number of updates rises, the limit of usability of VR mode is clearly reached and the system has to rely on IVC transactions to meet the deadlines.

VI. Related Work

We discuss related work along two axes: (1) TM scheduling in general and (2) real time and deadline management for transactional memory.

TM scheduling: Conflict resolution and progress guarantees in TM are typically the responsibility of an application-level *contention manager* [12]. As argued in [9], conventional (non-scheduling) contention managers notably lack precision and have very limited control (or no control at all) on the scheduling of transactional threads. Further, to be able to guarantee progress, they must detect every conflict, which rules out the "invisible read" design adopted by many efficient STM implementations.

Several researchers have explored the use of a dedicated transactional scheduler for improving the performance of STM. CAR-STM [16] takes a scheduling-based approach for contention management. It maintains per-core transaction queues, where the transactions in each queue are executed by a dedicated thread in a sequential manner. Upon collision, the loser transaction is enqueued behind the winner transaction. Yoo and Lee [17] implemented a simple adaptive user-level scheduler that essentially serializes transactions once a high level of contention is detected. This approach is effective in specific settings where parallelism actually degrades performance. Ansari et al. [18] designed a transaction scheduler that avoids wasted work by allowing transactions to "steal" conflicting transactions so that they execute serially. Dragojevic et al. [19] proposed another user-level transaction scheduler that bases its scheduling decisions on the access patterns of past transactions.

These approaches are complementary to our work. We do not explicitly serialize conflicting transactions except when they are irrevocable. Our scheduler extension serves mainly to control thread migration and preemption in certain cases.

One should note that avoiding thread preemption has also been explored in TL2's [1] implementation on Solaris using the *schedctl* mechanism to request short-term preemption deferral during the commit phase. 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 active transaction before the commit phase nor does it handle conflicts as they are encountered.

At the level of the operating system, TxLinux [20] 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.

In previous work, we proposed an operating system scheduler [9] that serializes conflicting transactions on the same core and avoids preempting threads that are executing active transactions. We did not consider deadlines nor the execution time of transactions, as the objective was to reduce the abort rate of the application without consideration for the latency of individual transactions. We reuse some of these mechanisms for our deadline-aware scheduling framework.

Real-time TM and deadlines: RT-STM [21] is an extension of Fraser's STM [22] that supports real-time transactions. RT-STM has been integrated in the LITMUS^{RT} real-time operating system [23]. The modifications to Fraser's STM are minimal: the conditions under which a transaction helps another one to commit have been modified such that higher-priority transactions are helped by lower-priority ones. This modification only applies to the commit procedure in its read and write phases. Note that with Fraser's STM, there is no guarantee that a transaction will commit even if it has a high priority because it may abort before reaching the commit phase. Therefore, unlike in our approach, RT-STM cannot enter a privileged mode in which it is guaranteed to run uninterrupted: it merely changes the priority of transactions closer to their deadlines and does not take into account their expected length. Evaluation was conducted only on red-black trees and showed slightly reduced jitter and a higher number of transactions that meet their deadlines.

RTTM [24] is a proposal for a hardware transactional memory for chip-multiprocessors in real-time systems. It has been designed to support small transactions with few read/write operations and has been evaluated on a simulated processor. The major contribution of RTTM is to bound the maximum number of retries for periodic threads.

Fahmy et al. propose an algorithm to compute an upper bound on the response time of transactions in distributed multiprocessor real-time systems [25], but they do not address the issue of implementing a real-time STM.

Several approaches for implementing irrevocability (also called inevitability) have been proposed and compared in [6], [7], [8], but they are not used in the context of deadlines. Our design also slightly differs as our objective is to allow optimistic transactions to progress concurrently.

VII. Conclusion

Transactional memory relies on optimistic concurrency control and, as such, is not directly applicable to reactive applications where operations must be completed within a bounded amount of time. In this paper, we have presented a novel approach to handling transaction with deadlines. Our deadline-aware scheduler framework allows programmer to associate deadlines with transactions. We use a combination of mechanisms to (1) estimate the duration of transactions, and (2) adaptively modify the execution mode of the transactions as the deadline nears, switching from the most optimistic approach that provides the highest level of concurrency to more pessimistic modes (visible reads and, eventually, irrevocable) that provide more deterministic guarantees regarding execution time but also limit the exploitable parallelism.

Our framework is implemented as a combination of mechanisms added to an existing software transactional memory library for transaction length measurements and adaptive mode switching, and an extension of the operating system scheduler that avoids thread preemption and migration for transactions subject to deadlines.

Experimental evaluation of reactive applications shows that our deadline-aware scheduler framework significantly improves the number of transactions that commit by their deadlines without noticeable degradation in the overall transaction throughput.

Acknowledgments

We are grateful to the authors of *synquake* [10] for providing us with their application. The research leading to the results presented in this paper has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement No 216852.

References

[1] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *DISC'06: 20th International Symposium on Distributed Computing*, 2006, pp. 194–208.

[2] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *PPoPP '08: 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 237–246.

[3] A. Dragojević, R. Guerraoui, and M. Kapałka, "Stretching transactional memory," in *PLDI'09: ACM SIGPLAN Conference on Programming Languages Design and Implementation*, Jun. 2009.

[4] J. Sreeram, R. Cledat, T. Kumar, and S. 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.

[5] T. Yang, T. Liu, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Redline: first class support for interactivity in commodity operating systems," in *OSDI'08: 8th USENIX conference on Operating systems design and implementation*, pp. 73–86.

[6] A. Welc, B. Saha, and A.-R. Adl-Tabatabai, "Irrevocable transactions and their applications," in *SPAA '08: 20th annual symposium on Parallelism in algorithms and architectures*. ACM, 2008, pp. 285–296.

[7] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott, "Implementing and exploiting inevitability in software transactional memory," in *ICPP'08: 37th International Conference on Parallel Processing*. IEEE CS, 2008.

[8] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc, "xCalls: safe i/o in memory transactions," in *EuroSys'09: 4th ACM European conference on Computer systems*, 2009, pp. 247–260.

[9] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller, "Scheduling support for transactional memory contention management," in *PPoPP'10: 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2010, pp. 79–90.

[10] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza, "Transactional memory support for scalable and transparent parallelization of multiplayer games," in *EuroSys'10: 5th European conference on Computer systems*. ACM, 2010, pp. 41–54.

[11] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985.

[12] W. N. Scherer III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *PODC'05: twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, Jul. 2005, pp. 240–248.

[13] M. Herlihy, "SXM: C# Software Transactional Memory. Unpublished manuscript, Brown Univ." may 2005, http://www.cs.brown.edu/~mph/.

[14] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski, "Anatomy of a scalable software transactional memory," in *TRANSACT 2009: 4th ACM SIGPLAN Workshop on Transactional Computing*, feb 2009.

[15] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi- processing," in *IISWC'08: IEEE International Symposium on Workload Characterization*, sep 2008.

[16] S. Dolev, D. Hendler, and A. 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*, Aug. 2008.

[17] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *SPAA'08: 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Jun. 2008, pp. 169–178.

[18] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. C. Kirkham, and I. Watson, "Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering," in *HiPEAC: 4th International Conference on High Performance Embedded Architectures and Compilers*, 2009, pp. 4–18.

[19] A. Dragojevic, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing versus curing: Avoiding conflicts in transactional memories," in *PODC'09, 28th Annual ACM Symposium on Principles of Distributed Computing*, Aug. 2009, pp. 7–16.

[20] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, A. Bhandari, and E. Witchel, "TxLinux: Using and managing hardware transactional memory in an operating system," in *SOSP'07: 22nd ACM symposium on Operating systems principles*, Oct. 2007, pp. 87–102.

[21] T. Sarni, A. Queudet, and P. Valduriez, "Real-time support for software transactional memory," in *RTCSA'09: 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009, pp. 477–485.

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

[23] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, “LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers,” *RTSS’06: 27th IEEE International Real-Time Systems Symposium*, pp. 111–126, 2006.

[24] M. Schoeberl, F. Brandner, and J. Vitek, “RTTM: Real-time transactional memory,” in *SAC’10: 25th ACM Symposium on Applied Computing*, March 2010.

[25] S. F. Fahmy, B. Ravindran, and E. D. 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.