Supporting Time-Based QoS Requirements in Software Transactional Memory

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

Supporting Time-Based QoS Requirements in Software Transactional Memory

WALTHER MALDONADO, University of Neuchâtel, Switzerland
PATRICK MARLIER, University of Neuchâtel, Switzerland
PASCAL FELBER, University of Neuchâtel, Switzerland
JULIA LAWALL, Inria/LIP6/UPMC/Sorbonne University, France
GILLES MULLER, Inria/LIP6/UPMC/Sorbonne University, France
ETIENNE RIVIÈRE, University of Neuchâtel, Switzerland

Software Transactional Memory (STM) is an optimistic concurrency control mechanism that simplifies parallel programming. Still, there has been little interest in its applicability for reactive applications in which there is a required response time for certain operations. We propose supporting such applications by allowing programmers to associate time with atomic blocks in the forms of deadlines and QoS requirements. Based on statistics of past executions, we adjust the execution mode of transactions by decreasing the level of optimism as the deadline approaches. In the presence of concurrent deadlines, we propose different conflict resolution policies. Execution mode switching mechanisms allow meeting multiple deadlines in a consistent manner, with potential QoS degradations being split fairly among several threads as contention increases, and avoiding starvation. Our implementation consists of extensions to a STM runtime that allow gathering statistics and switching execution modes. We also propose novel contention managers adapted to transactional workloads subject to deadlines. The experimental evaluation shows that our approaches significantly improve the likelihood of a transaction meeting its deadline and QoS requirement, even in cases where progress is hampered by conflicts and other concurrent transactions with deadlines.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms: Performance, Reliability

Additional Key Words and Phrases: Contention Management, Fairness, Quality of Service, Scheduling, Transactional Memory

1. 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 [Dice et al. 2006; Felber et al. 2008; Dragojević et al. 2009a; Sreeram et al. 2007]. 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 execute without risking harmful concurrent interleavings 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. For an important class of such QoS-constrained applications, reactivity constraints are imposed on the operations performed by one or several threads. 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 specific properties for the transactions composing the application. In particular, these applications often rely on rendering and aggregation mechanisms that use *read-mostly* transactions. Read-mostly transactions are often long transactions, whose read sets are expected to grow large as many data elements are accessed in order to compute the aggregates (e.g., images or queries results). They are executed periodically, but not as frequently as other transactions, such as the ones supporting updates to the shared state.

The criterion for measuring the achievement of reactivity for long, read-mostly transactions is naturally expressed by means of constraints on their execution time, in the form of *deadlines*. The application developer may express deadlines by specifying the maximum delay until commit, or by a

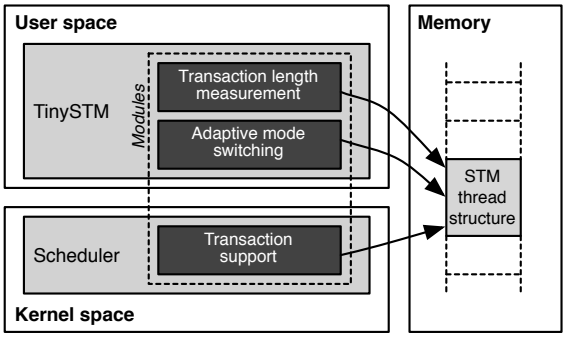


Fig. 1. Deadline-aware transaction scheduling framework.

fixed point in time by which a commit is required [Yang et al. 2008]. To achieve reactivity in practice, a majority of instances of read-mostly transactions must succeed before their given deadlines. This majority must be quantifiable: a rate of success for committing before deadlines is set as a target, and enforced by the system. The pair formed by the deadline and the expected rate of success forms a quality-of-service (QoS) requirement for a transaction.[1]

To ensure that deadlines are respected, support at the runtime level is required. So far, constraints on processing time in the context of STMs have been supported by executing transactions that are not allowed to abort in a serial irrevocable mode [Volos et al. 2009]. However, if one transaction runs in this mode, no other transaction may commit its writes during its execution. Therefore, systematically executing read-mostly transactions in irrevocable mode 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, read-mostly transactions can be executed optimistically without being aborted if the level of contention is low. This suggests that it is necessary to resort to irrevocability 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 read-mostly transactions 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 less degradation of 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 read-mostly transactions should run under at a given time, we need an oracle that provides the execution time of each atomic block. For this purpose, we continuously sample the distribution of read-mostly transactions' durations. Sampled distributions are then used as indications of the expected maximum durations for given percentages of instances of these transactions.

Our approach has been implemented in the TinySTM STM library [Felber et al. 2008] and the Linux OS (see Figure 1). The implementation relies on two modules that extend TinySTM with

[1]Note that our problem at hand is a form of *soft real-time* execution. A small fraction of instances may complete after their execution deadline. We do not target a *hard real-time* setting which would require different mechanisms such as the use of worst-case execution times and a controlled environment.

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 [Maldonado et al. 2010]. A preliminary version of this work, where only the case of a single transaction with a deadline was considered and with fewer evaluation results, was presented in [Maldonado et al. 2011].

Our achievements are as follows:

— We propose mechanisms to support deadline-aware scheduling for reactive applications based on TM. We find that the developer contract on one or several transactions subject to deadlines is met even when the number of concurrent update transactions reaches a high level. We present evaluation results on a server with 4 quad-core Opteron processors (for a total of 16 cores). Our approach is able to achieve a constant frame rate of 30 images per second on a 3D interactive simulation rendering application [Spear et al. 2008], with guarantees on the regularity of the rendering period. On a Quake-like game server engine [Lupei et al. 2010], 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. Our success rate consistently lies between 98% and 100% of that observed when using only irrevocability but our adaptive strategy divides by 3 the average number of retries needed by other transactions in the highest contention case and reduces the required CPU load on the system.
— We present and evaluate how to use sampling to effectively determine the running time of transactions in a multi-threaded setting. 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 [Minh et al. 2008].
— We describe and justify experimentally our claim that the use of 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 with execution modes for which the execution time is more predictable but which have a higher cost and impact on the progress of other concurrent transactions. These execution mode changes are nonetheless only performed when 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 starts to be needed only with 16 update threads.
— We consider the specific case of concurrent deadline requirements. We introduce a measure of *fairness* for comparing contract achievements over several threads supporting transactions with deadlines. We also describe and use four contention managers designed with different goals in terms of performance and fairness, as well as approaches for handling concurrent requests to enter irrevocability mode while avoiding excessive usage of irrevocable calls. Through testing on reactive applications we identify the three combinations that yield the most favorable results.

The rest of the paper is structured as follows: Section 2 presents our sampling mechanism for accurately measuring transaction durations. Section 3 describes the design and implementation of the adaptive switching mode, which ensures that a transaction can commit, or helps a transaction to commit. Section 4 focuses on the kernel-level scheduling extensions that provide support for better transaction length predictability for the threads running transactions with deadlines. Section 5 presents the specific mechanisms supporting fairness in the presence of concurrently executing transactions with deadlines. Section 6 presents a thorough evaluation of all proposed mechanisms. We present related work in Section 7 and conclude in Section 8.

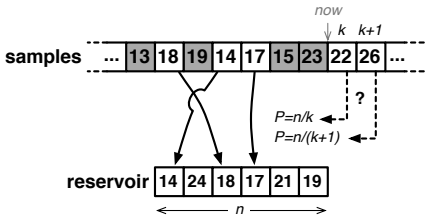


Fig. 2. Vitter's reservoir sampling [Vitter 1985].

2. 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 reactive applications, such as the ones 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 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 interested only in irrevocable mode executions in order to bootstrap the oracle, and later we are interested only 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 between cores, 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. Support from the kernel module allows detecting and discarding measurements that correspond to one of the two latter situations. Note that an additional criteria could be to ignore the first measurements in order to discard cold cache effects. We do not use this latter criteria as experiments have shown no measurable and statistically significant impact of cold caches on the measured transaction lengths.

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 [Vitter 1985]. 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 warm-up phase fills the reservoir with 1% of its capacity by systematically running the transaction in the most conservative execution mode (irrevocable). This allows the oracle estimation to act as a safeguard based on the worst case before enough measurements are available for providing a correct estimation in the general case. Once the warm-up phase is complete, optimistic mode executions are sampled, in order to base the decisions of the transaction scheduler on the common case running time.

We can obtain an estimate of a given quantile of the distribution represented inside the reservoir by simply reading the corresponding position in the reservoir if it is maintained sorted (our approach), or by scanning linearly through its elements otherwise. For instance, the 99th percentile is an upper bound on the execution time of 99% of the transactions that were sampled in the reservoir, for a given atomic block. Since the distribution of transaction length over time does not vary or varies only slightly, we can therefore use a single reservoir per atomic block subject to deadlines 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 uses a simple sliding window.

3. 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.

3.1. Design Choices

TinySTM, like several other state-of-the-art STM libraries (e.g., [Dice et al. 2006; Dragojević et al. 2009a]), relies on an optimistic execution mode with eager locking and 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[2] 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, since transactions may repeatedly abort, one cannot easily bound their execution time. Priority-based contention managers [Scherer III and Scott 2005] 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.

3.2. Optimistic Mode

We first recall TinySTM's basic optimistic mode [Felber et al. 2008]. 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.

[2]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.

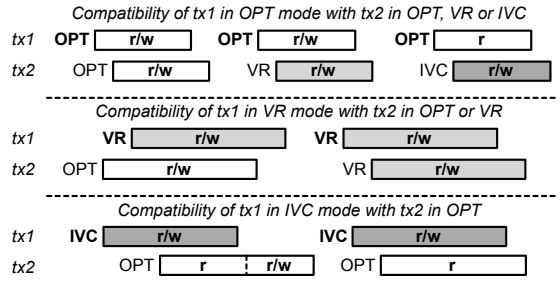


Fig. 3. Compatibility of the three execution modes supported by TinySTM. Rectangles represent transactions *tx1* and *tx2* performing reads (r) or arbitrary memory accesses (r/w), and increased shading indicates increasingly pessimistic execution modes. Each row corresponds to one thread and time flows to the right. Overlapping transactions can execute concurrently if they do not conflict.

Table I. Summary of the three execution modes for controlling the degree of optimism of transactions.

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

Upon write, a transaction first acquires the lock that covers each updated memory location 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.

3.3. 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 a reader in VR mode over other transactions executing in optimistic mode. This allows any conflicting writer to back off and let a reader complete its execution. In our framework, this mode enables read-mostly transactions to make progress while reducing the probability of aborts.

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 [Herlihy 2005]) 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 [Lev et al. 2009] 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 rationale behind this design choice is that transactions will seldom use visible reads. 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. In our evaluation, we consider that only a transaction subject to a deadline can be in VR mode, and only so if it fails to commit in optimistic mode. This is not a strict limitation: our implementation supports concurrent executions of VR-transactions, even though the considered applications do not currently use this possibility.

3.4. Irrevocable Mode

In irrevocable (also called inevitable [Spear et al. 2008]) 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 reliable, 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.

We note that it is not possible in the general case to support multiple concurrent IVC transactions. The exception to this rule would be when all memory accesses by irrevocable transactions are known in advance. This would allow certifying the absence of conflict. Otherwise, one can trivially construct an interleaving with just two irrevocable transactions that leads to a deadlock. Support for concurrent irrevocable transactions would require specific runtime and compiler support and the range of transactions that could benefit from it would be limited. We thus consider it to be outside the scope of the present work.

A promising approach is to allow an irrevocable transaction to execute concurrently with other non-irrevocable transactions in OPT and VR mode. Several such algorithms have previously been discussed and evaluated [Welc et al. 2008; Spear et al. 2008]. We propose a new variant, which also uses a global token that a transaction must acquire before it becomes irrevocable. Once the token has been acquired, no other update transaction may commit, independently 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 objects, it must validate its read set before irrevocability can be granted. Failed validation triggers an abort and the transaction directly restarts in irrevocable mode. In the context of deadline-aware scheduling, a transaction enters irrevocable mode only upon restart after a conflict is detected and when a deadline nears.

Since an irrevocable transaction is never allowed to 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.

A difference with the mechanisms in [Welc et al. 2008; Spear et al. 2008] is that we allow IVC transactions to steal the locks of concurrent transactions, instead of waiting for such transactions to revoke themselves. In order to implement the lock stealing, the IVC transaction changes the concurrent transaction status from *active* to *killed* using a CAS operation and then steals the lock.

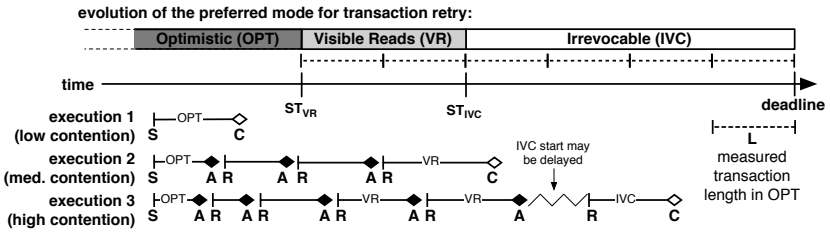


Fig. 4. Transaction execution mode switching for a transaction under increasing levels of contention, and times for changing execution mode before the deadline. S, A, R and C respectively denote `start`, `abort`, `retry`/failed `commit`, and successful `commit` operations.

Changing the status first ensures that the concurrent transaction notices the stealing before trying to validate its execution.

The concurrent execution of an IVC transaction with transactions in other modes may lead to a longer execution time for the IVC transaction compared to its execution in a *serial* mode, where a single transaction is allowed in the whole system (e.g., as implemented by [Ni et al. 2008]) and which does not require any instrumentation at the cost of greatly impacting the progress of other threads. Each write by the IVC transaction may require aborting another concurrent OPT or VR transaction, while each read may require aborting another concurrent VR transaction. We evaluate the relative costs of the different execution modes in Section 6.

4. SCHEDULING TRANSACTIONS WITH DEADLINES

We present in this section how deadlines can be expressed by the developer and how the STM library chooses the execution mode for a given atomic block. Finally, we describe two extensions to the contention manager and to the Linux scheduler to provide efficient deadline-based transaction support.

This section is related to the case where only a single transaction with a deadline is executed at a time. The next section generalizes the proposed mechanism for the case where multiple transactions with deadlines are running concurrently.

4.1. Setting Deadlines

The deadline associated with an atomic block is a QoS contract between the programmer and the STM library. As such, a deadline is a pair: a time relative to the first attempt 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 as an attribute for compilers supporting transactional memory. A deadline can be set at run time and therefore an atomic block may have a different deadline for each execution.

4.2. Dynamically Adapting the Execution Mode

Adaptation of the transaction execution mode with a deadline is performed by a module that extends TinySTM. This is done when a transaction restarts 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 2). 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 99^{th} percentile value of the transaction length distribution as recorded in the reservoir.

Using a higher value for q yields a larger value for L and 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.

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

— after time ST_{VR} and before ST_{IVC}, the transaction switches from optimistic mode to VR mode;
— after time ST_{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 ST_{VR} and ST_{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 6. We choose these times 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 ST_{VR}, we set $ST_{IVC} - ST_{VR} = 2 \times L$. This scenario is illustrated by execution 2 in Figure 4: a restart in OPT mode just before ST_{VR} still gives the transaction the opportunity to restart—and commit—in VR mode.
— Similarly, a transaction starting in VR mode just before ST_{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 $ST_{IVC} + 2 \times L$. We also take into account a possible delay of at most L due to another pending IVC transaction.[3] We thus set ST_{IVC} such that deadline $- ST_{IVC} = 4 \times L$ ($2 \times L$ for a VR execution, $1 \times L$ for the delay, and $1 \times L$ for the actual execution).

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 1% 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 rapidly. Once the number of samples in the reservoir grows to $f > 1\%$ 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 as the estimation L 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., $q' = 100$).

4.3. 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 update 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*.

[3]We consider here that this concurrent transaction is of the same expected duration as the one waiting for the execution in IVC mode. This corresponds to the common case where deadlines are associated with transactions by different threads but for the same atomic block. If multiple atomic blocks are used, the expected duration of the other transaction in IVC mode can be considered as $L_{\max}$, the maximum value of L over all sampled atomic blocks.

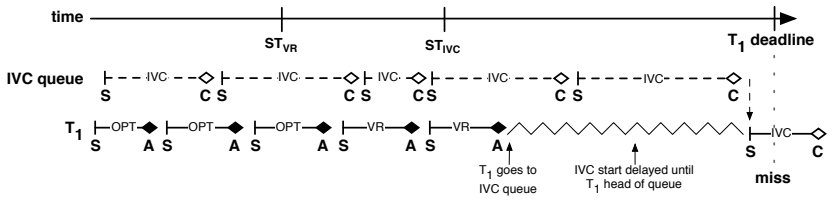


Fig. 5. Situation depicting the problem of a large *IVC* queue.

4.4. 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 time 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 [Maldonado et al. 2010] 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. We have observed that, 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 another while it is executing a deadline based transaction. This would induce a wrong time measurement since timestamp counters are not necessarily synchronized across cores, and would prevent using 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 detects thread migrations at the kernel scheduler level. It then forbids such migrations when possible, and invalidates the running time measurement otherwise.

5. SUPPORT FOR CONCURRENT DEADLINES

In this section, we describe the extension of the mechanisms presented in Section 4 for multiple (and potentially simultaneous) transactions with deadlines on different threads. There are additional considerations to take in such a setting, since ensuring that one transaction completes on time can cause others to be delayed. Further, as contention increases there comes a point where it is not possible to schedule all transactions in a way that they all meet their deadline.

We present a notion of *fairness* in the case of multiple transactions with deadlines. In this regard, we present metrics that preserve fairness when dealing with competing deadlines as well as considerations taken to avoid irrevocable transactions causing excess misses in other threads.

5.1. Fair QoS Degradation

Fairness, in our context, refers to ensuring similar performance between the competing deadlines. Fairness could be ensured on a per transaction basis (by comparing against past deadline executions), but also by comparing between different threads. We now look at the motivation behind each approach.

Rate based fairness. The *hit-rate*, in our context, refers to the ratio of transactions that meet their deadlines. It is desired, from a performance point of view, that all threads perform equally well in this regard. As an example, consider a multi-user application where each user is handled by its own

thread. If some threads are systematically penalized over others in meeting the deadlines associated with their transactions, it translates into a handicap against some users.

Overflow distribution based fairness. The *overflow time* is defined as the time elapsed between a missed deadline and the commit of the transaction. Even if a transaction has missed its deadline, it should still complete as soon as possible. Furthermore, the overflow time should be similar for all missed deadlines, as this makes it easier for users to adapt to the decreased *QoS*. For instance, in an interactive application, it is easier to deal with a constant increased response delay than it is to adapt to varying response times [Pantel and Wolf 2002].

Note that in both cases, we consider that the different threads (and henceforth the different transactions associated with deadlines that they run) have the same priority. Specifying different priority levels for each of the threads would require extending the API or the set of attributes known to the compiler, and applying weights in the decision processes described below. We do not implement such extensions.

5.2. Fairness-Oriented Contention Managers

Fairness rules can be enforced at conflict time, through different contention managers (CM). The following CM strategies target fairness through different criteria:

— ***Basic***: The base CM gives priority to the transaction closest to its deadline. This strategy serves as a baseline.
— ***Fair***: The fair CM aims for a leveled hit-rate among threads. This strategy prioritizes the thread with the lower hit-rate.
— ***Overflow***: The overflow CM aims to stabilize overflow cycles among threads. This strategy prioritizes the thread with the higher average overflow times.
— ***Compound***: The compound CM represents a compromise between *basic* and *overflow*. The transaction with the earlier deadline gets priority except when both transactions have missed their deadline already. In that case, priority goes to the thread with the greater overflow time.

We present an evaluation of the impact of these various contention managers in Section 6.6.

5.3. Irrevocable Mode and Impact on Fairness

The default single-token based implementation proposed in Section 3.4 is insufficient in the presence of multiple, concurrent threads with deadlines. To guarantee fairness, a queue must be used to grant threads the right to use the IVC mode. This *IVC queue* ensures that transactions are executed in order, respecting their priorities based on the deadline. The IVC queue can be sorted by the deadline time (i. e., priority goes to the transaction closest to deadline), by the thread *hit-rate* or by the *overflow time*.

One issue with using a queue is that when there is a long queue of transactions waiting to enter IVC mode, a new transaction has to wait for the complete queue to finish before it can make progress. If the wait time is long, the transaction can end up being added to the queue, too. Figure 5 illustrates this situation. A new transaction *T1* is unable to make any progress while the queue is not empty. And when the time ST_{IVC} arrives, the transaction is placed on the queue itself. However, by the time the transaction executes in IVC mode, it is already too late to meet its deadline. Worse, its execution in IVC mode delays other transactions, potentially causing more transactions to be added to the queue. If the arrival rate for transactions with a deadline is high enough, the queue will never empty as every transaction will be added to the queue when its deadline approaches. Thus, transactions are being executed serially even if there would be no conflict among them. Executing all transactions in optimistic mode would have given better results. We note that this situation is specific to IVC mode transactions, as a single such transaction is allowed at any given time. It does not happen with VR and OPT transactions, as multiple transactions using these modes are allowed concurrently.

Avoiding this situation requires additional information: the number of concurrent deadlines, the time remaining to the deadlines of the other threads, and the expected time to empty the IVC queue. One approach with low overhead is to track the number of concurrent deadlines, and to not use IVC

when this number exceeds a certain threshold. Using this approach, some deadlines might be missed but queue saturation is avoided.

6. EXPERIMENTAL EVALUATION

In this section, we present an exhaustive evaluation of our approach. We demonstrate its ability to achieve the desired success rate for committing before deadlines, while keeping the overhead and impact on the overall throughput minimal. We then evaluate the ability of the adaptive mode switching strategy to make transactions commit before their deadline using two applications, *swarm* [Spear et al. 2008] and *synquake* [Lupei et al. 2010]. We also assess the corresponding impact on the other transactions' throughput. Our evaluations consider both the case of a single transaction associated with a deadline and the case of concurrent deadlines. In the latter case, we also evaluate the fairness achieved with respect to the different policies presented in Section 5.

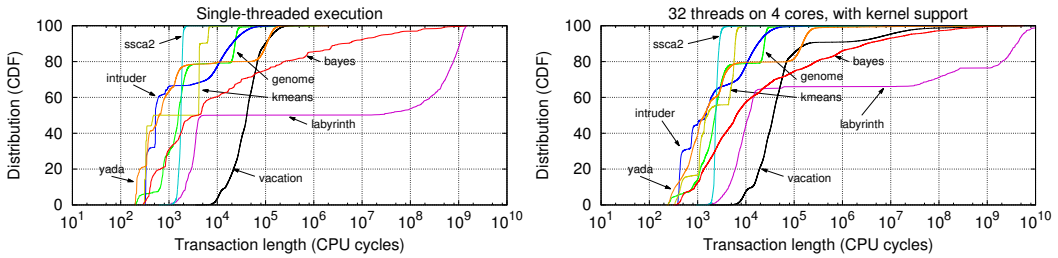
6.1. Experimental Setup and Benchmarks

All tests have been carried out on an AMD Opteron server with four 2.3 GHz quad-core CPUs (16 cores in total) and 8 GB RAM running Linux 2.6.34. We use a reservoir size of 10,000 elements. All results in this section are averaged/aggregated over 10 runs of the following applications: *bank*, the STAMP [Minh et al. 2008] benchmark suite, the *swarm* [Spear et al. 2008] rendering application and the *synquake* [Lupei et al. 2010] game server. These applications are described below.

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 [Minh et al. 2008] benchmark applications, which present a large variety of transaction lengths and read/write set sizes. 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. *Vacation* and *kmeans* are associated with two sets of parameters, to produce executions with low and high contention. We consider only those yielding high contention. The single-threaded execution time of the STAMP applications ranges from a few seconds to several minutes. The multi-threaded execution time is detailed in our experiment. It strongly depends on the level of contention and on the transactional memory runtime and algorithm that are used.

We then consider two 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 [Spear et al. 2008]. It performs asynchronous rendering and updates of a 3-dimensional scene graph. One thread is responsible for the rendering while the other threads move collections of objects in the shared state and detect their collisions in the scene. In order to obtain a given number of frames per second, we have slightly modified *swarm* so that all the rendering is performed in a single periodic transaction that is assigned a deadline. In our *swarm* instances, the renderer always reads the whole set of 26,856 scene objects. *Synquake* emulates a game server with multiple clients, interacting over a 2D map [Lupei et al. 2010]. 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 *synquake* quest "worst" that yields the highest contention. We further increase contention by using 128 players, each being a 2x2 tile



application	variation	application	variation
genome	+5.30%	ssca2	+0.88%
intruder	+6.05%	vacation	+29.84%
kmeans	+3.01%	yada	+44.49%
labyrinth	+3.46%	bayes	+370.45%

Fig. 6. Transaction length measurements in a single thread execution and reservoir content for an execution with 32 threads 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.

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 deadlines in our experiment.

6.2. Measuring Transaction Length

We evaluate the ability of the transaction length measurement module to determine the duration of transactions. In a multithreaded setting, support from the kernel module allows discarding invalid measurements due to migrations and preemptions. 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, the thread is not interrupted, as we use only one core out of 16. The upper left plot of Figure 6 shows the cumulative distribution of transaction lengths as retrieved from the reservoirs. The other plot presents the same information in a multithreaded setting. In order to evaluate the effectiveness of our mechanisms to effectively predict transactions duration even in difficult scenarios, we use 32 threads on 4 cores. This generates 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 follows 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 6: when the pruning of invalid values is disabled, the 99th percentile of the length distribution may vary by as much as +370.45% for the bayes application.

6.3. Adaptive Transaction Execution Modes

We evaluate the different adaptive execution modes in two ways. First, we demonstrate the tradeoff between the gain in execution length predictability 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.

6.3.1. Transaction Length Predictability and Throughput Tradeoff. Figure 7 presents the individual and global throughput of *bank* where one thread (denoted *first*) can run its transactions in any mode

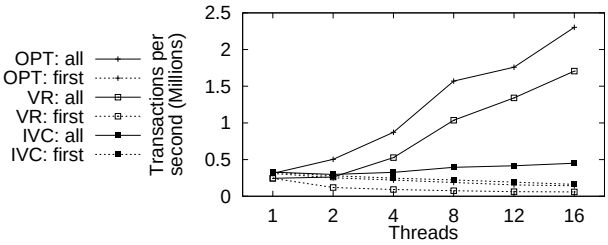


Fig. 7. Illustration of the individual throughput of *bank* with a special thread ("first") running a transaction in OPT, VR, or IVC mode while other threads run transactions only in OPT mode, and impact on the application global throughput ("all").

while the other threads run them only in OPT mode. It illustrates the claim, made in Section 3, 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 transaction execution.

6.3.2. Predictable Execution Length. Table II presents the execution times for OPT transactions, and the relative difference for VR and IVC, for the 90^{th} and 99^{th} percentiles, for the STAMP applications running with 32 threads on 4 cores. The total running time is given next to each application name. 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 using the median of the distribution.

For a majority of transactions, the difference between OPT and VR, and OPT and IVC, is 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 3.3). The lengths of IVC transactions, compared to the length of OPT transactions for the same block, depend on the nature of this atomic block. For small transactions, the cost of acquiring the acquiring the global IVC token is higher than the gain observed for not having to maintain a read set. Longer transactions that are read-dominated run faster in the IVC mode than in the OPT mode as they do not pay the overhead of maintaining that read set. Nonetheless, longer transactions that are write-dominated can have longer execution times in the IVC mode than in the OPT mode. This is due to the fact that each write may require aborting a concurrently running OPT transaction in order to steal the locks it holds, and is a result of allowing concurrent OPT and VR transactions with the IVC transactions as long as the latter are not allowed to commit their writes. Despite these small fluctuations, we note that for long-running transactions (such as in labyrinth, yada, or bayes), predicting the execution time in VR and IVC modes based on OPT executions is accurate within a small margin of error (resp. [-2%:+12%] for VR and [-4%:+1%] for IVC). Profiled transaction lengths, with outliers due to migration/pre-emption removed, are therefore a reasonable estimate of the expected running duration in any of the modes.

6.4. Application Characterizations

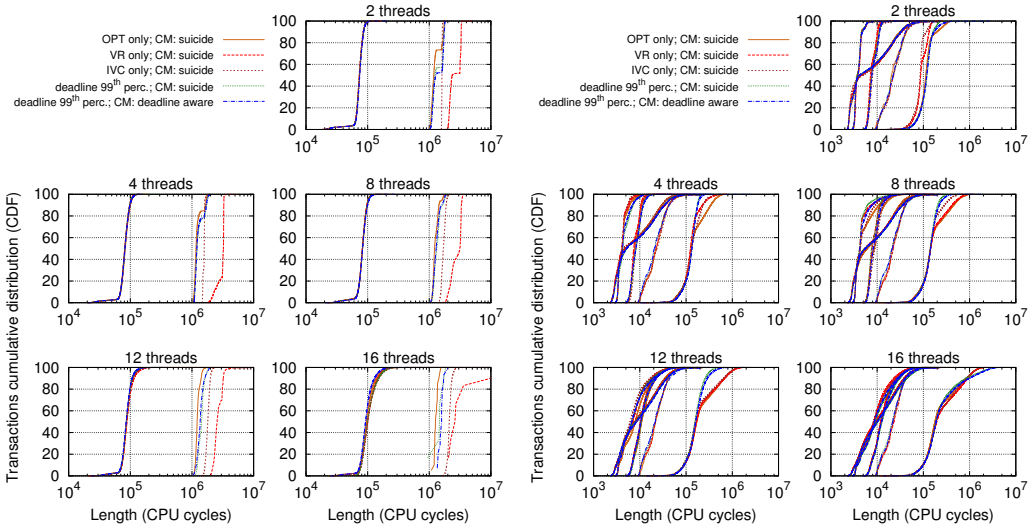
Figure 8 characterizes the *swarm* and *synquake* applications in terms of the distribution of their transaction lengths measured in CPU cycles. For each atomic block, we present the distribution of the lengths of the corresponding transactions. Each set of CDFs corresponds to a distinct atomic block. There are two atomic blocks for *swarm* and five for *synquake*. The atomic block to which we assign deadlines always correspond to the set of CDFs that are furthest to the right (i.e., longest execution times). Only transactions associated with this atomic block may use different execution modes and other transactions always use the OPT mode. In more details, we consider the following five scenarios. In (OPT only), all transactions run in OPT mode and the contention manager used is suicide. The (VR only) and (IVC only) scenarios are similar, but transactions for the longest atomic block always use the VR mode or IVC mode, respectively. With the (deadline/suicide) scenario,

Table II. STAMP: Transaction lengths for different modes.

App.	Block	90th percentile			99th percentile		
		OPT	VR	IVC	OPT	VR	IVC
genome (8.65s)	1	11.90 μs	*+8 %*	*-2 %*	13.20 μs	*+9 %*	*-1 %*
	2	0.38 μs	*+28 %*	*-6 %*	0.48 μs	*+22 %*	*-3 %*
	3	1.28 μs	*+17 %*	*-2 %*	233.3 7μs	*+47 %*	*-3 %*
	4	0.80 μs	*+7 %*	*-10 %*	0.94 μs	*+9 %*	*-11 %*
	5	0.99 μs	*+10 %*	*-3 %*	1.20 μs	*+9 %*	*-2 %*
intruder (60.21s)	1	0.38 μs	*-3 %*	*-14 %*	0.45 μs	*-3 %*	*-14 %*
	2	16.63 μs	*+14 %*	*-20 %*	39.91 μs	*+8 %*	*-20 %*
	3	0.17 μs	*+1 %*	*-3 %*	0.43 μs	*+8 %*	*-3 %*
kmeans (155.38s)	1	2.97 μs	=	*-1 %*	3.12 μs	=	*-1 %*
	2	0.20 μs	*-21 %*	*-12 %*	0.22 μs	*-2 %*	*+2 %*
	3	0.35 μs	*-36 %*	*+14 %*	7.27 μs	*-37 %*	*+26 %*
labyrinth (68.16s)	1	1.78 μs	*+12 %*	*+25 %*	2.10 μs	*+9 %*	*+21 %*
	2	543.6 ms	*-2 %*	*-2 %*	633.7 ms	*+2 %*	*+2 %*
	3	1.60 μs	*+31 %*	=	1.60 μs	*+31 %*	=
ssca2 (25.89s)	1	12.41 μs	*-21 %*	*-7 %*	12.41 μs	*-21 %*	*-7 %*
	2	2.09 μs	*+51 %*	*+55 %*	2.09 μs	*+51 %*	*+55 %*
	3	0.88 μs	*+1 %*	*+2 %*	0.98 μs	*+1 %*	*+3 %*
vacation (53.76s)	1	31.45 μs	=	=	35.75 μs	*+1 %*	=
	2	71.21 μs	=	*+1 %*	113.80 μs	=	*+1 %*
	3	19.99 μs	=	=	24.68 μs	*+1 %*	=
yada (250.73s)	1	0.72 μs	*+23 %*	*+7 %*	4.72 μs	*+5 %*	*+2 %*
	2	0.28 μs	*+28 %*	*+36 %*	0.40 μs	*+23 %*	*+24 %*
	3	67.60 μs	*+11 %*	*-3 %*	92.89 μs	*+12 %*	*-4 %*
	4	0.18 μs	*+25 %*	*+48 %*	0.24 μs	*+19 %*	*+34 %*
	5	0.85 μs	*+33 %*	*+4 %*	4.54 μs	*+8 %*	*+2 %*
	6	0.49 μs	*+8 %*	*+10 %*	0.49 μs	*+8 %*	*+10 %*
bayes (48.5s)	1	9.94 μs	*+13 %*	*+28 %*	9.94 μs	*+13 %*	*+28 %*
	2	0.74 μs	*+39 %*	*+2 %*	10.08 μs	*+1 %*	=
	3	0.54 μs	*+17 %*	*+6 %*	1.09 μs	*+36 %*	*-17 %*
	4	10.52 μs	*+2 %*	*-1 %*	22.20 μs	*+8 %*	*-6 %*
	5	8.54 ms	*+1 %*	=	31.32 ms	*+1 %*	*+1 %*
	6	0.32 μs	*+16 %*	*+40 %*	0.65 μs	*+57 %*	*+52 %*
	7	311.76 μs	*+1 %*	*-1 %*	29.36 ms	*+2 %*	*+3 %*
	8	366.47 μs	*+2 %*	*+1 %*	10.63 ms	*+2 %*	*+1 %*
	9	0.25 μs	*+27 %*	*+24 %*	9.85 μs	*+23 %*	*+1 %*
	10	64.52 ms	*+1 %*	=	592.4 ms	*+1 %*	*+1 %*
	11	16.05 ms	*+1 %*	*+2 %*	226.8 ms	*+2 %*	*+2 %*
	12	19.99 ms	*+2 %*	*+1 %*	314 ms	=	=
	13	1.15 μs	*+31 %*	*-13 %*	9.87 μs	*-22 %*	*+2 %*

all transactions initially run in OPT mode but the ones for the longest atomic block, which use the dynamic change of execution mode between the three availble modes. The contention manager is suicide. Finally, (deadline/deadline aware) is similar to the previous but using the deadline aware contention manager.

The CDFs show the distributions of lengths for all OPT transactions, under the five scenarios. This allows highlighting the influence of the long transaction running in another mode on the duration of other transactions. As expected, the influence on the duration of OPT transactions that actually commit is minimal. What is influenced is the abort rates over all transactions for the corresponding atomic blocks, as we detail in the next section. As expected, for the considered read-mostly long transactions subject to deadlines, VR mode executions are costlier due to the acquisition of the lock for each read and not only for writes as in OPT mode, while the running time in IVC mode is comparable or slightly higher than the running time in OPT mode. Note that some CDFs correspond to very small set of values and are therefore not statistically significant. As only successful executions of the transactions can be used in the CDF, this happens when the read-mostly transactions seldom succeed in committing. This is the case for instance with 16 threads for the *swarm* application in the VR only scenario. The throughput figures shown in the remainder of this section highlight cases where the read-mostly transaction is prevented from committing and where the CDF is not significant.



(a) ***swarm:*** Transactions lengths.

(b) ***synquake:*** Transactions lengths.

Fig. 8. Transaction length distribution for the two test applications, for the different execution modes and when deadline-based execution mode change is enabled. Each line corresponds to a distinct atomic block: there are two distinct atomic blocks for *swarm* and five distinct atomic blocks for *synquake*. In each case, the distinctly longer transactions correspond to the atomic block to which we assign a deadline contract.

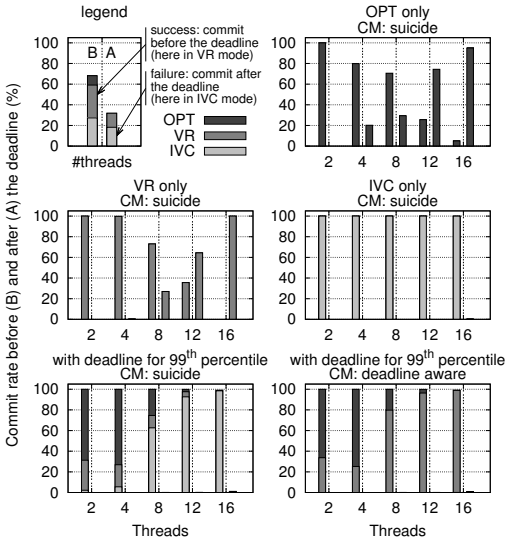
6.5. Deadline Aware Scheduling

We now evaluate the effectiveness of our approach in ensuring that a transaction commits before its deadline, both for periodic tasks with the *swarm* rendering application, and for one-shot tasks with the *synquake* application, while keeping the throughput of commits by other transactions high (and their abort rate low). When setting deadlines, we use the 99th percentile for the associated contract. Using a contract of 100% would risk 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.

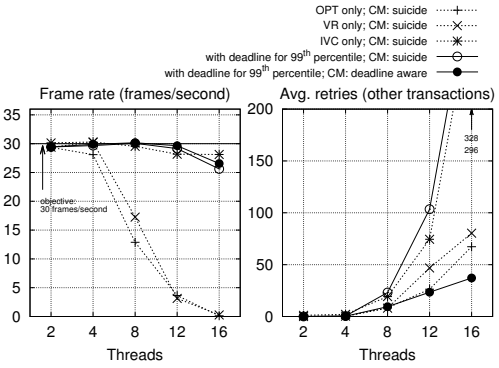
6.5.1. Periodic Rendering — Swarm Application. The rendering transaction in *swarm* periodically compiles objects that are then sent to an OpenGL rendering library (*glut*). As such, it is a read-mostly transaction that performs a consistent snapshot of the scene while allowing concurrent execution of 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 9(a) presents the success rates (commits before the deadline) for various execution modes, while the left graph of Figure 9(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 9(b)). We note that the overall throughput of update transactions depends directly on the number of update threads and the average number of retries.

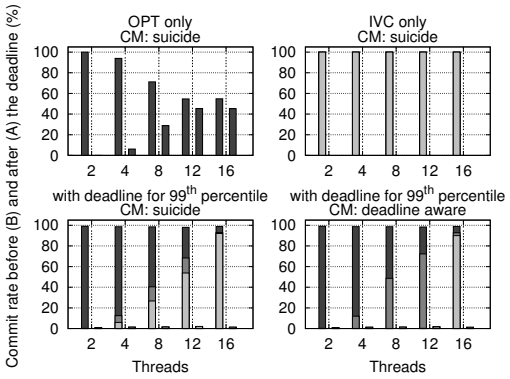
Running all transactions in OPT mode does not give a reasonable frame rate for more than 4 threads. For 8 threads, even if 60% of the rendering transactions succeed by the deadline, those that do not succeed typically last for multiple rendering periods, reducing the frame rate to 10 frames



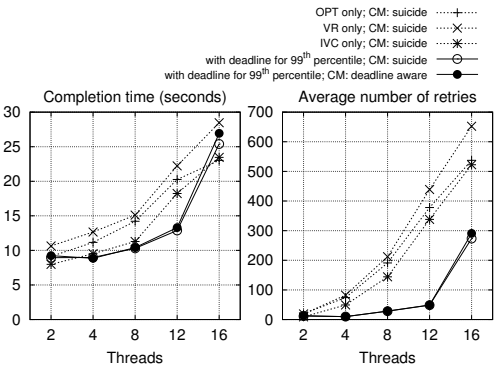
(a) ***swarm:*** Execution modes and success of committing by the deadline.



(b) ***swarm:*** Frame rate and contention.



(c) ***synquake:*** Execution modes and success of committing by the deadline.



(d) ***synquake:*** Completion time for the complete workload and contention.

Fig. 9. Performance and success rate of the two test applications, *swarm* (top) and *synquake* (bottom). For *swarm*: executions with a deadline of 33% of the frame generation period (1/180 second). Baselines: OPT-, VR- and IVC-only mean that the transaction for which we set a deadline executes in this mode, while the others all run in OPT. For *synquake*: executions with a deadline of 750μs for the transaction implementing the "attack" action.

per second. This highlights the unpredictability of transaction lengths when running in OPT-mode. Similarly, running the rendering transaction only in VR mode does not succeed in reaching the target frame rate, as transactions abort frequently, and also frequently span 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, and strongly impairs the progress of the scene update.

When executing with a deadline contract on the 99$^{\text{th}}$ percentile value from the reservoir using either the *suicide* or the *deadline-aware* CM, 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 transactions that resort to IVC mode, as the transaction typically aborts in VR mode, whereas the *deadline-aware* CM prioritizes it over other transactions allowing it to commit in VR mode in the majority of cases. Third, a corollary of the previous observation is that using the *suicide* CM results in much larger contention in the system and higher abort rates, as one has to pay the price of contention of an IVC transaction plus potentially many OPT and VR transactions before it. Moreover, the *deadline-aware* CM not only meets a majority of deadlines using OPT or VR modes (a few uses of IVC mode, not visible on the figure, are still necessary for part of the success rate for 8 threads and more), but also results in contention that is *3 times lower* than using directly IVC with 16 threads, as shown by the average number of retries for other transactions in Figure 9(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.

6.5.2. 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 90^{th} 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.

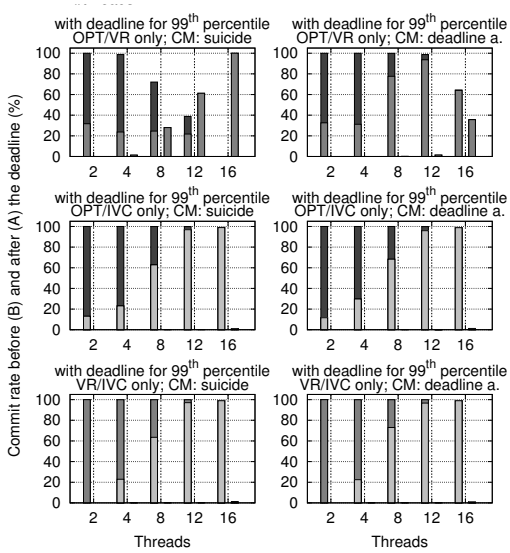
Figure 9(c) presents the success and failure rates for committing before the deadlines in the same way as for *swarm*. However, since *synquake* attempts to simulate a given workload as fast as possible, we consider the running time of the application instead of a frame rate for measuring the overall 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 (Figure 9(d)). We do not present the results for VR-only for deadline success rate: as for *swarm*, 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 CMs 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 in the *swarm* case, making use of VR transactions for up to 12 threads. For 16 threads, and as the number of updates rises, the limits of VR mode are clearly reached and the system must rely on IVC transactions to meet the deadlines.

6.5.3. Impact and Importance of Using Three Execution Modes. Our final experiments in this section seek to evaluate the interest of using all three execution modes, against the use of only two of these modes. We consider the following cases. In (OPT/VR only), transactions associated with a deadline start in OPT mode, switch to VR mode when reaching the ST_{VR} threshold, but do not switch to the IVC mode. In (OPT/IVC only), these transactions also start in OPT, but only switch directly to the IVC mode when reaching the ST_{IVC} threshold without using the VR mode. Finally, in (VR/IVC only), these transactions start directly in VR mode and switch to IVC mode when reaching the ST_{IVC} threshold without using the OPT mode. We consider for each case both the use of the suicide contention manager and the deadline aware contention manager.

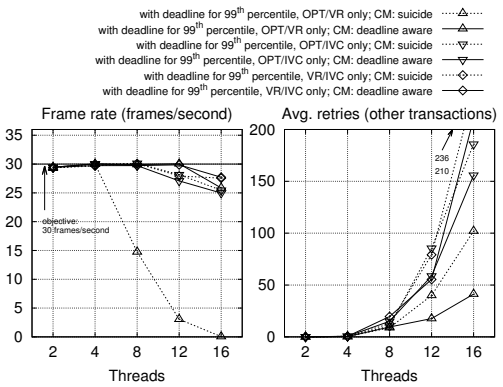
Figures 10(a) and 10(b) present the success of committing by the deadline and the frame rate and contention for the *swarm* application, similarly as the previous figures 9(a) and 9(b). Figures 10(c) and 10(d) present the success of committing by the deadline and the completion time and contention for the *synquake* application, similarly as the previous figures 9(c) and 9(d).

For *swarm*, we make the following observations.

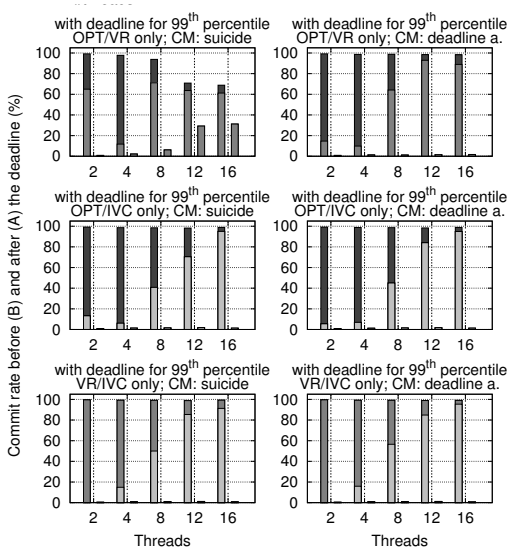
The (OPT/VR only) case with the suicide contention manager performs similarly to the (VR only) case, and does not allow reaching the target frame rate with more than 4 threads. Using the deadline aware contention manager in this case yields a large improvement, as VR transactions are more likely



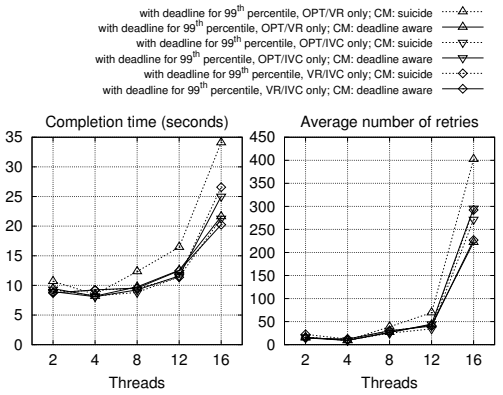
(a) ***swarm:*** Execution modes and success of committing by the deadline, when only two execution modes are enabled.



(b) ***swarm:*** Frame rate and contention, when only two execution modes are enabled.



(c) ***synquake:*** Execution modes and success of committing by the deadline, when only two execution modes are enabled.



(d) ***synquake:*** Completion time for the complete workload and contention, when only two execution modes are enabled.

Fig. 10. Performance and success rate of the two test applications, *swarm* (top) and *synquake* (bottom) when only two out of the three execution modes are enabled. Other settings are rigorously the same as for Figure 9. The legend for the left plots is shared with the left plots of Figure 9.

to commit before the deadline than when using suicide, as observed in our previous experiments. Avoiding the use of IVC in this case also reduces the contention for other transactions, but at the same time only allows committing before the deadline for roughly two thirds of the transactions when using 16 threads.

Using the (OPT/VR only) case with both of the suicide and deadline aware contention managers yield the same results: all transactions commit by the deadline and the target frame rate is reached. This is similar to the case when all three modes are used. Nonetheless, the contention for other transactions is much higher than when using the three modes with the deadline aware contention manager. This applies even for low thread counts. By systematically switching to the most pessimistic IVC mode, this configuration does not take advantage of the lower contention and intermediate predictability the VR mode provides. This highlights, in particular for configurations using 4 to 12 threads, that the use of the three modes combined with the deadline aware contention manager provides the best compromise in terms of deadline satisfaction and impact on contention for the update transactions.

Finally, using the (OPT/IVC only) case with both contention managers allow respecting the deadlines in all cases but performs almost as bad as the (IVC only) case considered in the previous section.

The observation for the *synquake* application are the following.

(OPT/VR only) with the suicide contention manager only slightly outperforms the (OPT only) case. Using the deadline aware contention manager however allows meeting almost all deadlines and yields a similar running time and contention for other threads compared to the best case achieved with three modes in the previous experiments. Clearly, for this configuration the impact of the contention manager is greater than the impact of the different execution modes, provided that the VR mode is used (other experiments, not shown, indicate that using (OPT-only) with the deadline aware contention manager does not improve since the transaction with a deadline systematically aborts upon commit due to other previously committed transactions).

Using the (OPT/IVC) and (VR/IVC) modes on the other hand shows little impact of the contention manager used. For all thread counts, the proportion of transactions committed in either of the two modes considered is similar and the contention slightly above the one obtained by using the three modes. The impact of using the three modes in this case is not as apparent as for the *swarm* application, but it does not impair performance either.

6.6. Concurrent-Deadline Aware Scheduling

We evaluate now the effectiveness of our approach in a concurrent-deadline scenario, as well as the specific mechanisms defined for them. For this section, we reuse the *synquake* application with the same settings, but the attack action has the deadline requirement enabled in all threads. Each execution setup was performed fifty times, and the run with the median result was taken for analysis.

6.6.1. Measuring Fairness. Fairness is defined in our context as an even distribution of the measure being considered. In the case of hit-rate fairness, it means that the range between the maximum and minimum hit-rate (when comparing per-thread results) should be low. In the case of overflow cycles, a more complex approach is needed, as fairness is related to the variance of the overflow times for each missed deadline of each thread. A variant of the transaction length measurement module is used to measure these times. This variant allows tracking of arbitrary values and is used by each thread to track the overflow time for their missed deadlines. This way we get access to an online computation of the variance and the average, as well as a reservoir that represents the overflow time distribution per thread.

6.6.2. Contention Manager Effectiveness. Figure 11 shows the hit-rate performance for each contention manager. Each bar represents the performance spread across the different threads. Of the contention managers that take fairness into account, *compound* gives the best fairness for higher number of threads. This is due to the dual nature of the contention manager. As the number of conflicts increases and more deadlines are missed, its behavior, which favors the threads with the most wasted cycles, increases the fairness. Both the *fair* and *cycle* CMs yield higher hit-rate, at the expense of reduced fairness.

The *fair* contention manager gives fairness similar to the other contention managers, despite being designed to maximize fairness. The reason it performs poorly can be attributed to the lack of a proper queue to handle irrevocable executions.

6.6.3. IVC Queues. The IVC Queue is implemented by using a sorted array, into which each thread wanting to enter irrevocable mode inserts its priority. Such a thread then spins until its turn. Sorting the list or using spinlocks incurs no additional overhead since a transaction entering IVC mode can make no progress while another is in IVC mode. By default the priority is the deadline value, which makes threads enter irrevocable mode according to how close their deadlines are. An alternative approach, which considers fairness, is to use the average number of overflow cycles of the thread as the priority. This way, threads that have missed their deadlines by longer times will get preference during irrevocable contention.

Figure 12(a) displays the effect of the queue using the two proposed priority metrics. In general, using a queue increases fairness (except in the single case of the *compound* contention manager and 12 threads, where the fairness without the queue was already high), and almost always increases the hit-rate as well. Between the two types of queue, the timestamp-based one yields better fairness in most scenarios, even if the overall hit-rate is worse in some of them.

One interesting effect is that a queue which uses the same priority measure as the contention manager (i.e., *fair CM* with a timestamp based queue or the *cycles CM* with an average cycles queue) generally gives worse performance than no queue at all. The opposite holds as well: the two combinations that give overall good results use the *fair CM* together with an average-cycles queue, or use the *cycles CM* with a timestamp based queue. An explanation for this phenomenon is that these combinations provide a conflict resolution scheme that uses more than one dimension, and that this avoids the situation where all new transactions get scheduled to execute in irrevocable mode.

6.6.4. IVC Threshold. As explained in Section 5.3, performance can degrade while using IVC queues when these queues fill up and never become empty. It is possible to avoid this issue by setting a limit on the number of transactions that can try to enter IVC mode at once. The number of concurrent deadlines is tracked, and a *threshold* is set so that no transaction attempts to enter IVC mode unless the number of them is less than the threshold.

Figure 12(b) presents the results using varying threshold values. The first thing to remark is that the threshold value needs to be high or the application suffers a sharp decline in performance. This is because the threshold offers no benefit unless the full queue problem manifests. On the contrary, a low threshold can only lower performance as threads cannot use irrevocable mode in cases where it is safe to do so.

This explains why there is no visible improvement for contention managers that had an over 90% overall hit-rate. In contrast, using a threshold has an impact for the *basic CM* and for the combination of the *fair CM* with a timestamp based queue, since these two setups trigger a full queue situation more often than the others. A threshold of 12 is sufficient to obtain good fairness and performance results for these cases.

6.6.5. Overflow Times. Finally, we look at the distribution of overflow times for missed deadlines. These are presented by Figure 13 as a CDF comparing the distribution for the worst performing

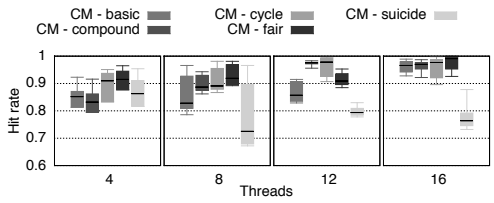
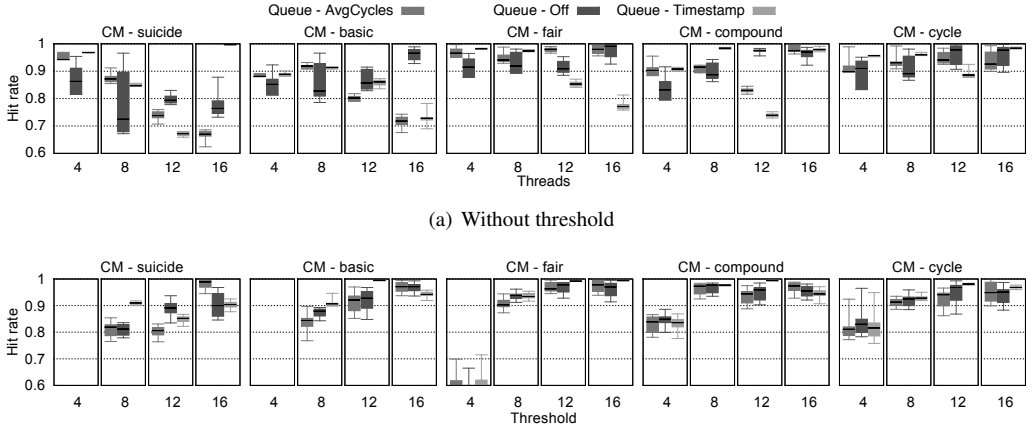


Fig. 11. Performance and correctness of synquake executions with relative deadlines of 750μs.



(a) Without threshold

(b) Using varying threshold values (16 threads)

Fig. 12. Hit-Rate performance for the synquake executions using different irrevocable queue types and thresholds.

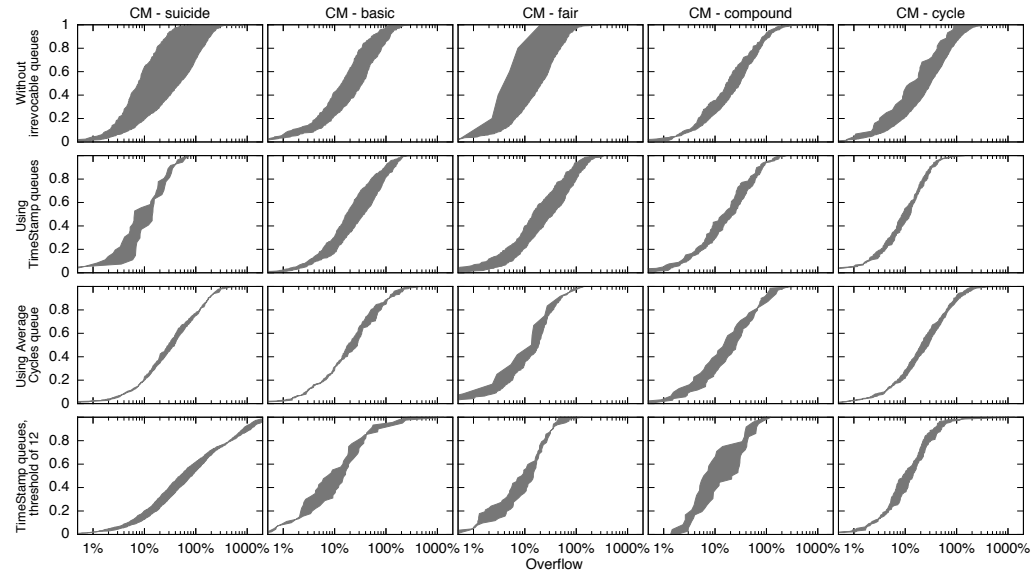


Fig. 13. Overflow time CDF for all contention managers using different setups.

thread against the best performing one (when using average per-thread hit-rate as the criterion). The *x* axis, representing the amount of overflow time, is presented as a percentage of the total deadline time, for context.

In rough terms, the fairness between threads can be seen in the graph as the marked area; the thinner it is, the more fair the execution. Another point of interest is by how much deadlines are missed. This is displayed in the graphs as the position of the curve. The more to the left that the curve is placed in a graph, the better its fairness.

The average-cycles based queue gives the best fairness between threads. Additionally, the combination *cycles CM* with a timestamp based queue gives similarly good fairness. This combination also has the smallest amount of cycles by which deadlines are missed.

Considering the hit-rate fairness of the previous section and the overflow time graphs, we can draw the following conclusions regarding the three best performing scenarios:

— The *fair CM* with an average cycles queue gives a good hit-rate, but the fairness between threads is less than with the other alternatives.
— The *cycles CM* with a timestamp queue gives in most cases an excellent hit-rate, with high fairness. From a cycles time perspective, it is also the one with the best fairness.
— The *fair CM* with a timestamp queue and a threshold of 12 also gives excellent hit-rate and fairness. However, it is slightly worse when looking at the distribution of overflow cycles.

7. RELATED WORK

We discuss related work along four directions: (1) support for multiple execution modes in transactional memory runtimes, (2) transactions scheduling in general, (3) work considering time-based constraints for transactional memory and (4) work considering runtime observation of transactional memory systems and dynamic adaptation.

7.1. Transactional Execution Modes

Welc et al. [2008] proposed single-owner read locks that allow one irrevocable transaction to run concurrently with revocable transactions. Spear et al. [2008] proposed Inevitable Read Locks which is a similar approach. While their mechanisms are close to our IVC execution mode, we specifically focused our implementation on time constraints. We must therefore limit the unpredictability of the transaction length. As a result, an important difference is that, instead of waiting for concurrent transactions to be revoked as they did, we allow the IVC transaction to abort and steal the locks of concurrent transactions as soon as the conflict is detected.

Ni et al. [2008] presented the implementation of transactional memory support in the Intel C++ proprietary compiler. A similarity to our work is that the runtime supports four execution modes for transactions: optimistic, pessimistic, obstinate, and serial. The *optimistic* and *pessimistic* modes are similar to our OPT and VR modes, using invisible reads and commit-time invalidation for the former and visible reads for the latter. A transaction using the *obstinate* mode allows other transactions to run concurrently but gets the highest priority upon conflict resolution. It corresponds to our VR mode for a transaction subject to a deadline, when using the deadline-aware contention manager. Transactions in the optimistic, pessimistic and obstinate mode are allowed to run concurrently. Finally, the *serial* mode ensures that a transaction using this mode is never aborted. Unlike our IVC mode however, or the irrevocable transactions of Welc et al. [2008] and Spear et al. [2008], it does not allow any concurrent transaction to run concurrently *at all*, similarly to the mechanism in Volos et al. [2009]. The runtime support allows dynamic mode switching during the execution of a transaction from the optimistic to the pessimistic mode. However, the paper does not consider how, and when, dynamic mode changes are decided. It only provides an implementation description of the support mechanisms for multiple execution modes. In particular, it does not consider runtime support for constraints on the transaction execution, nor does it monitor runtime statistics about the transactions to support such a decision making process. Decisions of when to use a given execution mode, or switch between modes, are left to the programmer through the appropriate language construct at the compiler level. The contributions presented in the present paper could be adapted to use the four execution modes of the Intel C++ compiler. The support for dynamic mode switching during the execution of a transaction, for instance, could be explored as a mechanism for improving transaction length predictability when approaching a deadline, rather than aborting and running the transaction again as we currently do. This would require access to the source of the compiler and runtime however, which is made more difficult due to their being proprietary and closed-source.

7.2. Transactional Memory Scheduling

Typically, an application-level *contention manager* [Scherer III and Scott 2005] manages TM conflict resolution and progress guarantees. Conventional (non-scheduling) contention managers notably lack precision and have very limited control (or no control at all) on the scheduling of transactional threads [Maldonado et al. 2010]. 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 STM performance. CAR-STM 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 [Dolev et al. 2008]. Yoo and Lee [2008] implemented a simple adaptive user-level scheduler that essentially serializes transactions once high contention is detected. This approach is effective in specific settings where parallelism actually degrades performance. Ansari et al. [2009] designed a transaction scheduler that avoids wasted work by allowing transactions to "steal" conflicting transactions so that they execute serially. Dragojević et al. [2009b] proposed another user-level transaction scheduler that bases scheduling decisions on past transactions' access patterns.

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. Avoiding thread preemption has also been explored in TL2's [Dice et al. 2006] 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, preventing the progress of others. It does not, however, control the scheduling of an active transaction before the commit phase nor does it handle conflicts as they are encountered.

In previous work, we proposed an operating system scheduler [Maldonado et al. 2010] 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.

7.3. Time-Constrained Transactional Memory

RT-STM [Sarni et al. 2009] extends Fraser's STM [Fraser 2004] to support real-time transactions. RT-STM has been integrated in the LITMUS^{RT} real-time operating system [Calandrino et al. 2006]. The modifications to Fraser's STM are minimal: the conditions under which one transaction helps another to commit have been modified such that higher-priority transactions are helped by lower-priority ones. This modification only applies to the commit procedure's 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 [Schoeberl et al. 2010] 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.

SEL-TM [Zhao et al. 2012] is another proposal for hardware transactional memory which uses eager conflict detection, except for selected cache-lines deemed as contention hot-spots, which use lazy conflict detection to improve performance. This approach however seeks to maximize the overall

throughput of the application but it does not take into consideration deadlines and the progress of a single transaction in particular, distinguishing it from our work.

Fahmy et al. [2009a] propose an algorithm to compute an upper bound on the response time of transactions in distributed multiprocessor real-time systems. Their work does not address the issue of implementing a real-time STM. The estimation is based on an oracle [Fahmy et al. 2009b] for estimating the worst-case execution time of transactions in the specific context of a real-time system running periodic, identical tasks under earliest-deadline-first scheduling. This requires knowing the execution time of periodic tasks, including both their non-transactional and transactional components. Belwal and Chen [2011] study schedulability conditions under a similar real-time model. Our approach considers the more general case of an arbitrary scheduler on a general-purpose kernel, and targets QoS requirements rather than strict/hard-real-time scenarios.

7.4. Runtime Statistics and Adaptation for Transactional Memory

Wang et al. [2012] proposed methods to dynamically select the appropriate TM runtime algorithm based on a combination of information obtained from a static analysis of the application code and runtime information collected during its execution. The profiling of a transactional application at runtime that is used in their work is also based on monitoring the transactions' execution times. The profiling of execution times is done by switching to a specialized TM runtime called ProfileTM, in which transactions run in isolation and have characteristics such as the size of their read and write sets monitored in addition to their running time. This can somehow be linked to our initial bootstrap of the reservoir using IVC transactions, albeit our IVC transactions still allow concurrent transactions that do not attempt to commit their writes. The profiling also does not consider the effect of thread migrations and other scheduling issues. Although the goal of Wang et al. is more to optimize the choice of the TM runtime for a given application and workload, we believe that a static analysis phase similar to the one they use could be investigated for better supporting time-based constraints for transactional memory.

8. 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, making it possible to support QoS requirements for operations supported by transactions. Our deadline-aware scheduler framework allows the programmer to associate deadlines with transactions. We use a combination of mechanisms to (1) estimate the duration of transactions, (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 read and, eventually, irrevocable mode) that provide more guarantees regarding the predictability of transaction execution time but also limit the exploitable parallelism, and (3) adaptively handle concurrent deadline requests in order to maintain fairness and ensure that the deadlines are met evenly across requests. While our approach does not target hard-real time systems, where stronger guarantees on the maximal execution time of transactions would be required–typically at the price of restrictions on the model of execution and large performance impact–the support of QoS time requirements can allow a large body of applications to benefit from STM support while maintaining a level of predictability in their execution.

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, and fair degradation in the case of multiple concurrent deadlines that cannot all be met at the same time.

Acknowledgments

We are grateful to the authors of *synquake* [Lupei et al. 2010] 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

Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris C. Kirkham, and Ian Watson. 2009. Steal-on-Abort: improving transactional memory performance through dynamic transaction reordering. In *HiPEAC: 4th International Conference on High Performance Embedded Architectures and Compilers*.

Chaitanya Belwal and Albert M. K. Chen. 2011. Scheduling conditions for real-time software transactional memory. *Embedded Systems Letters, IEEE* 3, 3 (2011), 93–96.

John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. 2006. LITMUS^{RT} : a testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS'06: 27th IEEE International Real-Time Systems Symposium*.

Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *DISC'06: 20th International Symposium on Distributed Computing*.

Shlomi Dolev, Danny Hendler, and Adi Suissa. 2008. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *PODC'08: 27th Annual ACM Symposium on Principles of Distributed Computing*.

Aleksandar Dragojević, Rachid Guerraoui, and Michał Kapałka. 2009a. Stretching transactional memory. In *PLDI'09: ACM SIGPLAN Conference on Programming Languages Design and Implementation*.

Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. 2009b. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC'09: 28th ACM Symposium on Principles of Distributed Computing*.

Sherif F. Fahmy, Binoy Ravindran, and E. Douglas Jensen. 2009a. 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*.

Sherif F. Fahmy, Binoy Ravindran, and E. Douglas Jensen. 2009b. Response time analysis of software transactional memory-based distributed real-time systems. In *SAC'09: ACM Symposium on Applied Computing*.

Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *PPoPP'08: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

Keir Fraser. 2004. Practical Lock-Freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge. (2004).

Maurice Herlihy. 2005. SXM: C# Software Transactional Memory. Unpublished manuscript, Brown Univ. (2005). http://www.cs.brown.edu/~mph/.

Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. 2009. Anatomy of a scalable software transactional memory. In *TRANSACT'09: 4th ACM SIGPLAN Workshop on Transactional Computing*.

Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. 2010. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *EuroSys'10: 5th ACM European Conference on Computer Systems*.

Walther Maldonado, Patrick Marlier, Pascal Felber, Julia Lawall, Gilles Muller, and Etienne Riviere. 2011. Deadline-aware scheduling for software transactional memory. In *DSN'11: 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. 2010. Scheduling support for transactional memory contention management. In *PPoPP'10: 15th ACM SIGPLAN Symposium on Principles and Practice of*

Parallel Programming.

Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *IISWC'08: IEEE International Symposium on Workload Characterization*.

Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. 2008. Design and implementation of transactional constructs for C/C++. In *OOPSLA'08: 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*.

Lothar Pantel and Lars C. Wolf. 2002. On the impact of delay on real-time multiplayer games. In *NOSSDAV'02: International Workshop on Network and Operating Systems Support for Digital Audio and Video*.

Toufik Sarni, Audrey Queudet, and Patrick Valduriez. 2009. Real-time support for software transactional memory. In *RTCSA'09: 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

William N. Scherer III and Michael L. Scott. 2005. Advanced contention management for dynamic software transactional memory. In *PODC'05: 24th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*.

Martin Schoeberl, Florian Brandner, and Jan Vitek. 2010. RTTM: Real-Time Transactional Memory. In *SAC'10: 25th ACM Symposium on Applied Computing*.

Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. 2008. Implementing and exploiting inevitability in software transactional memory. In *ICPP'08: 37th International Conference on Parallel Processing*.

Jaswanth Sreeram, Romain Cledat, Tushar Kumar, and Santosh Pande. 2007. RSTM: a relaxed consistency software transactional memory for multicores. In *PACT'07: 16th International Conference on Parallel Architecture and Compilation Techniques*.

Jeffrey S. Vitter. 1985. Random sampling with a reservoir. *ACM Trans. Math. Software* 11, 1 (1985).

Haris Volos, Andres Jaan Tack, Neelam Goyal, Michael M. Swift, and Adam Welc. 2009. xCalls: safe I/O in memory transactions. In *EuroSys'09: 4th ACM European Conference on Computer Systems*.

Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. 2012. A transactional memory with automatic performance tuning. *ACM Trans. Archit. Code Optim.* 8, 4, Article 54 (Jan. 2012).

Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. 2008. Irrevocable transactions and their applications. In *SPAA'08: 20th Annual Symposium on Parallelism in Algorithms and Architectures*.

Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2008. Redline: first class support for interactivity in commodity operating systems. In *OSDI'08: 8th USENIX Conference on Operating Systems Design and Implementation*.

Richard M. Yoo and Hsien-Hsin S. Lee. 2008. Adaptive transaction scheduling for transactional memory systems. In *SPAA'08: 20th ACM Symposium on Parallelism in Algorithms and Architectures*.

Lihang Zhao, Woojin Choi, and Jeff Draper. 2012. SEL-TM: selective eager-lazy management for improved concurrency in transactional memory. In *IPDPS'12: 26th IEEE International Parallel & Distributed Processing Symposium*.

Received 08 2013; revised 07 2014; accepted 11 2014