SAPIENZA
UNIVERSITÀ DI ROMA

Fine-Grain Time-Shared Execution of In-Memory Transactions

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Master of Science in Engineering in Computer Science

Candidate
Emiliano Silvestri
ID number 1101255

Thesis Advisor
Prof. Francesco Quaglia

Co-Advisors
Ph.D. Simone Economo
Ph.D. Alessandro Pellegrini
Ph.D. Pierangelo Di Sanzo

Academic Year 2015/2016

Fine-Grain Time-Shared Execution of In-Memory Transactions
Master thesis. Sapienza – University of Rome

© 2016 Emiliano Silvestri. All rights reserved

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: emisilve86@gmail.com

Contents

Preface iii

Acknowledgements vi

1 Introduction 1

2 Software Transactional Memory 4

3 Components 10

3.1 TinySTM 10

3.2 Extra-Tick Module 11

3.3 TPC-C Benchmark 15

4 The Preemptive STM Architecture 17

4.1 ULT 22

4.2 Memory Management 25

4.3 The Priority Queue 27

5 Policies for Priority Management 31

6 Experimental Study 35

7 Conclusions 41

Appendices **42**

A. Context Save and Restore **43**

B. Context Creation **46**

C. Preemption Check **48**

D. Scheduler **50**

Preface

In the early years of the past decade, the exponential growth of the sequential computing performance that has characterized the previous fifty years suffered a setback. Although the quantity of transistors on a chip continues to follow Moore's law, it has become increasingly difficult to continue to improve the performance of sequential processors by simply raising the clock frequency, mainly due to power and cooling motivations. To remedy this situation, the industry released the so-called "*multicore*", or "*chip multiprocessors*" systems, which provide for the presence of multiple processing units on a single chip and connected through a shared memory. In subsequent years the number of processors on a chip will increase at the Moore's law rate, as well as the peak number of instructions executed per seconds, allowing this architecture to be the potential solution to the problem of stalled performance growth.

On the other hand, a parallel program is far more difficult to design than an equivalent sequential program, and rarely offers a significant performance increase which may be attributable to the nature of the program and the impossibility to structure it in a set of parallel independent tasks. The most real-world computational problems cannot be effectively parallelized without incurring the cost of inter-processor communication and coordination because, while parts of the program mandatory need to be performed in a serial manner, the parallel parts may also need to access shared data, which in turn require particular synchronization techniques. Parallelization and synchronization can have therefore a dominant effect on performance resulting in this way

in an extremely non-linear speed-up curve tending to stall or worse. Thus, parallel programming makes more complex the programmer work with respect to the sequential one; a simple task requires much more attention in order to guarantee fundamental properties such as "*safety*" and "*liveness*", and an approach that seems to be very good in solving a problem could give rise to bad outcomes in solving another one.

While parallelism has been a difficult problem for general-purpose programming, database systems have successfully exploited parallel hardware for decades by executing many queries concurrently on multiple processors. The author of the query does not care anymore about parallelism and has only to focus on the correctness of the query itself, leaving the hard task of ensuring *atomicity*, *consistency*, *isolation* and *durability* (A.C.I.D.) to the transactional engine that is part of the database management system (DBMS). The transaction is indeed the heart of the programming model for databases and can be expressed as a group of read and write operations performed on shared objects which must appear to be executed atomically at a single point in time, before and after the effect of other transactions running or not concurrently, in a serial one-at-a-time order. Transactions offer therefore a proven abstraction mechanism in database systems for constructing reusable parallel computations, and the advent of multicore processors has renewed interest in an old idea, that of incorporating transactions into the programming model used to write parallel programs.

This is the approach followed in Transactional Memory (TM), a paradigm that enables programmers of concurrent applications to rely on *atomicity* and *isolation* guarantees provided by some TM layer. In this thesis I will present a fully innovative mechanism enabling in-memory transactions to be executed as preemptable tasks, with preemption actuated according to very fine grain intervals. This in turn enables a single thread to exploit one CPU-core in the most effective manner with respect to different priority levels of the transactions

to be processed, an issue that as not yet been tackled by any literature study.

Acknowledgements

I dedicate this thesis to the people who have supported me over the years, my *Mother* and my *Father*, aware of the fact that the economic and moral commitment given to me will be rewarded with future satisfactions. I want to thank my *grandfather*, who was able to observe closely all the advances of these Master's Degree studies. I want to thank all my old and new friends, the ones who have believed in me, who have given me the strength to reach the goal. Last but not least, I want to thank my *thesis advisor* for the opportunity to work in such an exciting environment, with all the successes achieved.

Chapter 1

Introduction

Transactional Memory (TM) is the raising paradigm for the management of shared data accesses on multicore machines. It allows programmers to mark code blocks as transactions, which are then handled, in terms of actual memory operations, by some underlying TM layer. The latter is in charge of guaranteeing *isolation* and *atomicity* while executing those code blocks, say all or nothing execution semantic. This allows achieving similar or better level of performance than fine-grain hand-made locking. Anyhow, TM jointly guarantees much higher transparency to the programmer since he is fully relieved from the burden of hand-coding synchronization operations. Nowadays various TM implementations exist, but the most diffused ones are still based on software support and known as Software-TM (STM). Despite the offered advantages, STM environments are still doomed to improvements, particularly for what concerns the managements of differentiated transaction priority levels. Indeed, in the state of the art Software Transactional Memory systems, great attention has been paid to the possibility to either reduce the incidence of transaction aborts [4], [2], [20], or determine the well suited level of parallelism of the TM-based application [16], [17], [18], [3], [5], while the only work we are aware of which discriminates between transaction priorities in TM systems is the one in [12], where the authors introduce an approach to favour transactions

experiencing abort retries due to conflicts while getting closer to deadlines, by promoting them to execute more conservatively through the use of different transaction execution modes that make it possible to reduce the level of optimism and to increase its predictability, which in turn implicitly enable a dynamic increase of the priority.

Handling different priorities is not an easy task, and it does not find an immediate solution in naive implementations as could be using more TM-threads than the number of available cores, mostly because they show a CPU-bound execution profile such that the resulting competition to grab CPU-usage would degrade performance and has been shown to be likely adverse to TM applications [7]. Furthermore, the dynamic spawning of every new thread as a reaction of some high priority request to run an in-memory transaction might be unviable due to an excessive overhead (given the much finer-grain nature of in-memory transactions compared to their counterpart in database systems). On the other hand, resorting on a static pool of threads for processing higher priority requests, each one bound to a given CPU-core, might give rise to CPU under-utilization along execution phases which do not show the presence of such higher priority requests. Consequently, the very basic implementation of such TM systems relies on the delayed processing of an incoming high priority request up to the point in time where some thread ends its last started transaction, and become able again to take care of the execution of a new transaction, the highest priority one available at that time. We also want to underline that, even if a signaling mechanism were used to notify the materialization of a standing request, the time needed to deliver the signal would be still bound to the conventional operating system timer-interrupt interval (typically ranging from 1 to 4 milliseconds on most operating systems' configurations), thus being not fully adequate to fastly react to the arrival of a higher priority transaction.

The solution we found to this problem resides in the design and implementa-

tion of a preemptive STM environment to be run on top of Linux/x86 systems. The core component is an ad-hoc timer management Linux module, which allows for (periodical) control flow variations along any running thread with no intervention by the chain of kernel-level mechanisms used for supporting Posix signals, hence leading to minimal run-time overhead. Clearly, we also manage differentiated execution *contexts* within the STM layer such that the transaction which leaves the CPU is not aborted, rather it can be resumed later along the original thread or another one running the STM application. Indeed, the module mechanism allows any registered thread running whatever transaction to be interrupted, and provides context saving for transactions that are going to be switched off the CPU to favour a higher priority one. The context associated to the latter is then loaded in CPU and the relative execution flow nested along the same thread, with no need to rely on additional threads and preventing in this way the aforementioned problems. Moreover this proposal does not create any bias in terms of CPU assignment across threads running on top of the Linux system given that the fine-grain timer-interrupt mechanism we adopt does not alter the original operating system planning in terms of overall CPU time to be assigned to the different threads, but it only allows an original tick destination to those threads to be partitioned into subintervals, at whose end control flow variation may occur.

Chapter 2

Software Transactional Memory

Transactional Memory aims to simplify concurrent programming by allowing a set of read and write operations on shared objects to appear as they have been atomically executed at a certain instant in time, before and after other transactions executed or not concurrently in a one-at-a-time order, as if each transaction had started from a consistent state and its effects had produced another consistent state. This high level of abstraction is a welcome solution for programmers who do not want to have to deal with low level thread synchronization mechanisms which could lead to make mistakes. Further, even if mistakes do not occur the whole execution could risk a significant performance degradation due to a not well handled synchronization procedure among the involved parts. Differently, transactional memory provides *optimistic concurrency control* by allowing threads to run in parallel with minimal interference. In this optimistic concurrency control scenario, a transaction is executed speculatively, that is it makes tentative changes to objects and if it completes without encountering a synchronization conflict, then it *commits* (the tentative changes become permanent), otherwise it *aborts* (all changes are discarded). Hence, the execution of a set of transactions on a set of shared objects can be modelled by a *history*, a total order of operations, commit and abort event. By this we may give the first very intuitive *safety* property to

guarantee a correct behaviour, that is the "*serializability*" property [13], for which a history of committed transactions is serializable if there exists a serial history that contains the same transactions belonging to the original history, if it is sequential (all transactions are not concurrent), and if every read returns the last value written. It's immediately clear that this property is needed to ensure the already mentioned one-at-a-time order of atomically executed and committed transactions, which in turn make it possible to understand the evolution of consistent states when the execution follows a correct behaviour. However, while this property is fully adequate to guarantee safety in a database environment, this is not sufficient in a STM environment where observing inconsistent values may either crash or hang an otherwise correct program, and we therefore need of a more strict property than serializability. This property is called "*opacity*" [9], and requires that every operation sees a consistent state, even if the transaction ends up aborting. In the classic concurrency control scheme, writes are buffered to private workspace and atomically applied at commit time while reads are optimistic and transaction is validated at commit time; hence this scheme does not guarantee opacity and should be fitted with a per-operation validation mechanism in order to safety execute STM applications. Besides the safety property, it is of particular interest the *liveness* property for those transactions that are subject to repeated aborts; however it's almost impossible to give strict progress guarantees in an asynchronous system and a desirable condition is that a correct transaction, which may abort and immediately restart a finite number of times, eventually commits. This is true in modern STM environments, where the aborts are unavoidable due to contention, but anyhow handled by mean of different contention-managers (CM) that encapsulate policies to deal with different contention scenarios.

However, despite the modern STM systems implement all the mechanism needed to support the correct optimistic execution of the transactions according to the properties described before, they do not intrinsically provide any method

to support the execution of transactions with different priority levels, that is the ability of the system to understand each time what is the highest-priority transactional request currently pending into the system itself, which by definition should take precedence over all the other ones so as to condition the time elapsed between its arrival and the final commit in positive way. In an hypothetical scenario, an application interfaced with a front-end system may need to perform critical activities before others, from which results depend the executions of subsequent tasks, hence it issues transactions labelled with different priority levels. Differently, a back-end system receiving an high-load of transactional request may dispatch predefined transactional profiles according to different priority levels, as it could be for the shortest-job-first (SJF) approach which is a classical way of managing priorities in computer systems in order to allow the optimization of server side run-time dynamics. Then, even if we cannot avoid transactions to abort upon conflict in an asynchronous system, this does not leave that the threads in charge of executing transactions may schedule the higher-priority ones before the others, affecting in this way the time they spend within the system.

Lot of attention has been paid in literature to the ability to adaptively adjust the concurrency degree of threads executing transactions, say the one that avoids thrashing due to excessive transaction aborts caused by oversized level of parallelism, by adopting the so-called *thread scheduling* policies. These techniques rely on either analytical or machine learning approaches, or the mixture of them as described in [3], [5], [16], [17], [18], which allow the threads to dynamically pause or resume depending on the workload profile that determines whether transactions are more prone to access the same data, which in turn yields to higher conflict (hence abort) rate. Other works focused on the possibility to reduce as much as possible the incidence of aborts. Along this path, several approaches have been based on the so-called *transaction scheduling* policies [2], [4], [20], which control whether some standing transaction can be

admitted to the processing stage, or need to be delayed for a while, because of a high likelihood of conflicts with already running transactions.

However, none of these works deals with transaction priorities, and with the possibility to timely pass control to higher priority transactions along an already running thread, which is instead under our investigation scope. Therefore, they are in such a way orthogonal to our work and they could be ideally combined with our one. The state of the art Software Transactional Memory is devoid of such topic, and the only work we are aware of is the one in [12]. Here, the authors give to the application the possibility to stipulate a QoS contract between the programmer and the STM library by associating an atomic block with a deadline, that is the point in time before which they want the related transaction completes. Such a value is a time relative to the first attempt of the transaction to commit, and it is computed as a multiple of an average among the collected times of runs which end up committing. Indeed, a reservoir of sampled times is continuously fed so as to maintain an history of the last successful runs (not aborted, only committed). This window of time is also composed by a certain number of subintervals, representing each one the range within which a transaction that repeatedly aborts must be re-started in a predefined execution mode. A transaction associated with a deadline starts executing in the *optimistic* mode by performing invisible reads, and in case it undergoes repeated aborts it restarts again in this mode until the end of the first interval, after which a more conservative mode is employed, the *read-visible* one, implying by definition the possibility to mark an object as read so as to allow other transactions running concurrently to detect the conflict at the same time at which it takes place. If the transaction experiencing conflict is running optimistically, then it aborts in favour of the read-visible one; conversely, one of the contention managers that the authors propose is engaged. Again, if the transaction in read-visible mode ends up aborting after this second interval of time, then it should be re-started in *irrevocable* mode. This last mode is the

most conservative one, and it provides that the transaction running in this way does not undergo abort by marking conflicting transactions as *killed* and by *stealing* their locks on already accessed shared objects. It's quite obvious that one and only one transaction at a time may be in irrevocable mode, which implies that multiple transactions experiencing abort in this interval must be enqueued and their execution delayed. Hence, this mechanism intrinsically implies a sort of prioritized execution of transactions which start executing more conservatively while approaching their deadlines.

On our side, the solution we provide does not alter the execution mode of running transactions, as well as it does not change the behaviour of the CMs employed by the TM-layer. Rather, it aims to support the execution of incoming transactional requests coupled with an in-birth priority label by reacting as fast as possible upon the arrival of a higher-priority transactional request. We also care of the time that elapses between the aforementioned arrival and the time at which the transaction starts running, which is typical in back-end systems where incoming requests are continuously accepted and enqueued by threads that are listening on a pool of sockets. In this scenario, a great number of different priority transactional requests, greater than the number of available CPU-cores, may be waiting in a ready-to-run state before their first execution, and it's clear they should not be handled in the same way, rather the environment must be designed in such a way that higher-priority transactions take precedence over the lower-priority ones, even if they are currently running in CPU. On the other hand, we do not want the already started transactions lose their progress whenever they are forced to undergo preemption in favour of a higher-priority transaction. It's therefore in our interest the responsiveness to the transaction's arrival events, without harming too much the already started transactions. Their contexts should be preserved so as to resume them later, at the time at which a no more busy thread will select one of these to continue its execution; no matter if conflicts have occurred

during the time the transaction is paused, because we leave this task to the TM-layer in charge of handling conflicts in the more appropriate way. In these conditions, more contexts than available CPU-cores can be active, and the classical architecture for STM-based applications is no more helpful in this sense. We will explain in detail how the architecture is structured and its setup phase in Chapter 4, immediately after discussing its core components in Chapter 3.

Chapter 3

Components

The implementation of our system, as we briefly discussed in the introduction chapter, relies on different technologies. It is integrated with the open source TinySTM package [8], which is indeed the STM system that we have chosen to support the execution of the transactions. The core part is the aforementioned Extra-Tick module, in charge of the control flow variations when needed. Further, the benchmark we adopt to test our architecture is the classical TPC-C [19], which simulates a complete computing environment where a population of users executes transactions against a database.

3.1 TinySTM

TinySTM is a very simple and performing *word-based* STM implementation that uses a single version of the LSA (Lazy Snapshot Algorithm) discussed in [15], which rests on a *time-based* design. Word-based means that it is possible to directly map transactional accesses to the underlying memory subsystem at the granularity of machine words or larger chunks of memory (a memory region), while time-based refers to the use of a global time to reason about the consistency of data accessed by transactions, and about the order in which transactions commit. The LSA algorithm has been introduced to resolve the

very expensive operation of validation needed to ensure the opacity property, that in its naive implementation requires to check all data previously read with a cost that grows quadratically with the number of accessed data. LSA, instead, allows to construct an always consistent snapshot for transactions derived from the intersection of the validity ranges of all those read/write operations performed by the transaction itself. If the intersection is a non-empty interval, then the versions of all accessed data overlap and we are still working in a consistent state, otherwise opacity is no more ensured and one of the transactions might need to wait or be aborted. In case one or more transactions incur a conflict, several contention management strategies are available, among which the CM_SUICIDE is the one we used to configure the TinySTM's CM in our implementation, which provides the immediate abort for those transactions that detect the conflict. Furthermore, TinySTM has the possibility to be configured with either WRITE_BACK_ETL (encounter-time-locking) or WRITE_BACK_CTL (commit-time-locking) to access data. Since we have introduced within TinySTM a fully innovative preemption facility, we decided to experiment with the commit-time-locking configuration. Encounter-time-locking, instead, would require an extra scheme to manage locks held by transactions that are going to be suspended, and which is actually out of our investigation scope.

3.2 Extra-Tick Module

Before explaining the structure of the module and its operation, we give some basic concepts about what hardware component is responsible for delivering interrupts and how the Linux operating system manages them. Architecture x86 processors are equipped with a timer facility exploited to drive the passage of time, named LAPIC-timer supported by APIC (Advanced Programmable Interrupt Controller), which is a timer-component local to the CPU-core. The

LAPIC-timer can be configured to operate in different modes, among which the one used by the Linux kernel is the periodic-interrupt mode. At boot-time the so-called *calibration procedure* sets the LAPIC-timer to a well defined scale value, so as to generate interrupts with a predefined period of time (typically ranging from 1 to 4 milliseconds).

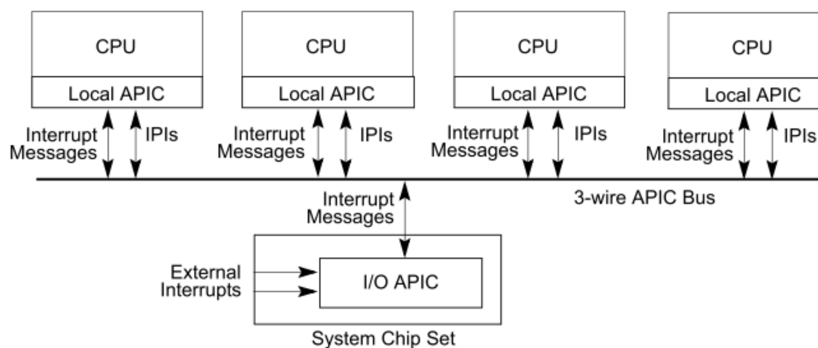

Figure 3.1. x86 interrupt system.

Linux handles timer-interrupt in the same way it does with other kind of interrupts, following the *top/bottom-half* paradigm. Indeed, for certain interrupts, there's the pressing need to timely handle the most urgent section of the events, and it is just the top-half part of the interrupt handler to fulfil this task which commonly includes very simple operations without risking to delay possible critical activities, plus the registration of the associated bottom-half part into an ad-hoc data structure that will be queried at certain reconciliation points, when will be sure there are no sensitive structures we are operating on. While the top-half part of the timer-interrupt handler simply increases the `jiffies` counter and marks the task-queue `tq_timer` as ready to be queried, the bottom-half checks whether the `need_resched` variable in the current PCB structure has been flagged, and in case invokes the `schedule()` function for performing actual context switches, actuated right prior to leave kernel mode, when no kernel-level critical task is being executed by the thread.

That said, it's clear that the module should be aware of which threads

need to be managed according to the lightweight extra-tick scheme, and which do not. Threads which are not involved in TM operations should not deliver extra-tick, but they have to be only subject to the basic scheduler's strategy that, as already mentioned, is absolutely not modified by the module activation. To this end, the module adopts a *dynamic patching* approach that rewrites parts of the executable image of the kernel upon being loaded and avoids in this way a kernel recompilation. The portions of the whole kernel architecture that need to know whether some thread is registered for delivering extra-tick are the kernel scheduler and the top-half part of the timer-interrupt handler. The `schedule()` function is patched by injecting an execution flow variation such that control goes to a `schedule_hook()` routine offered by the module right before `schedule()` would execute its finalization part, when the decision about what thread needs to take control of the CPU-core is already finalized. As a consequence, the module is able to check whether the thread is a registered one, and in case it changes the LAPIC-timer period according to a scale value. Further, it records in a per CPU-core entry a value to remember that the CPU-core is actually working in extra-tick mode, thus simplifying the reverse process when a non-registered thread shall take control on that CPU-core. The top-half hook, instead, is in charge of executing the same identical basic actions as those executed by the original top-half procedure with the difference that it is able to discriminate whether the interrupted thread is a registered one, and in positive case:

1. It decreases the extra-tick counter associated to the thread.

2. If the counter reaches zero, then the original period has expired and kernel-level timing information must be updated, as it was the original handler to carry out this task.

3. It exploits the CPU-context saved by the top-half part of the timer-interrupt handler to retrieve and update the user-level stack-pointer (SP)

address, as well as the original instruction-pointer (IP) address, in order to reflect the insertion of a new element, which is just an IP address different from that in which the thread was interrupted, so that when the timer-interrupt handler returns the user-level thread may asynchronously start executing a code block by not referring to any function call, but at the end of which a simple `ret` instruction will bring it to run up where it left off.

4. Finally, if the extra-tick counter reached the value zero, the thread is again filled with the number of extra-ticks it is allowed to receive in the next period.

The Figure 3.2 below gives an idea on how the top-half part of the extra-tick timer-interrupt handler recovers all it needs from the CPU state's snapshot presents in the kernel-stack area immediately after the CPU-core firmware accepts the interrupt. Once the user-level stack has been altered according to the extra-tick logic, such a snapshot is altered too in order to start running the code block mentioned above.

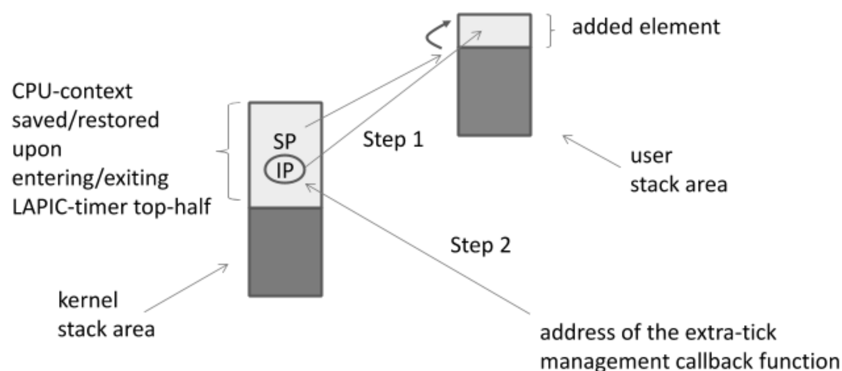

Figure 3.2. Stack and CPU context management by the LAPIC-timer top-half hook.

Last but not least, a thread can register itself to deliver extra-ticks by issuing a `ioctl` call towards a special device file called `dev_extra_tick` offered

by the module, which in turn exposes several facilities such as the possibility to register a callback function as entry-point for the code block we want to be performed at the completion of the interrupt handler. The behaviour of the top-half hook for the LAPIC-timer interrupt is schematized below.

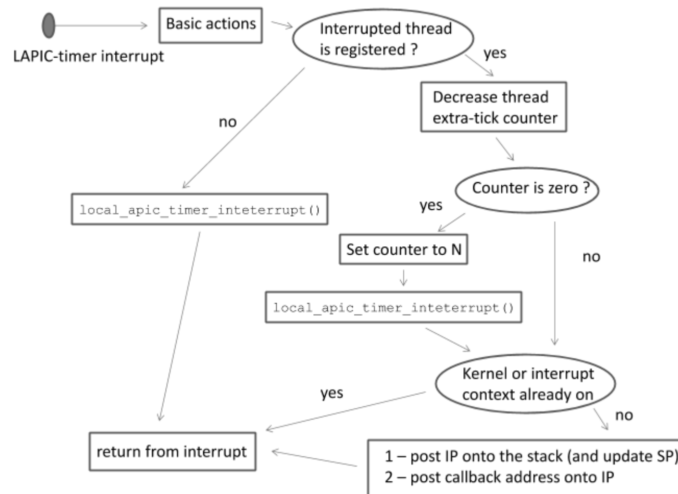

Figure 3.3. Behaviour of the top-half hook for LAPIC-timer interrupt.

By the way, this approach to modify the execution flow of a registered thread by altering the user space stack just above the current stack-pointer address, requires that the application is compiled by passing the `-mno-red-zone` directive to GCC, to indicate that we do not allow the conventional red zone displacement in the stack for leaf functions provided by some compilation tool-chains.

3.3 TPC-C Benchmark

Transaction Processing Performance Council (TPC) is a non-profit organization to define transaction processing and database benchmarks. The TPC-C benchmark measures the performance of online transaction processing systems (OLTP) and it is based on a complex database and a number of different transaction types that are executed on it. TPC-C simulates an environment

where the central elements are typical transactions of a wholesale company concerning order entries and includes 5 different transaction profiles (new order, payment, order status, delivery, and stock level) to supply items from a set of warehouses to customers within sales districts.

Chapter 4

The Preemptive STM Architecture

Our preemptive STM architecture is based on a few standard facilities offered by Linux, plus additional kernel and user space facilities we discussed in the previous chapter. In Figure 4.1 it's shown a high level schematization of the whole architecture in order to give to the reader a first idea of how it is structured and try to understand how it works. Our preemptive STM architecture is targeted at back-end STM environments that, differently from the front-ends which are mainly the presentation layers to interface with users, have the responsibility to correctly process incoming requests issued by users through more than one front-end system. In fact, our goal is primarily intended for the optimization of the performance of back-end systems in terms of overall reduction of turnaround time of high-priority transactions so as to spend less time as possible within the system, without damaging too much the lower priority ones.

From the moment a transaction arrives into the system until it is completed, it will be associated with one and only one data structure we name `transaction_context`, representative of the status of the transaction itself, including the CPU-context (all CPU registers) and the user-level stack area,

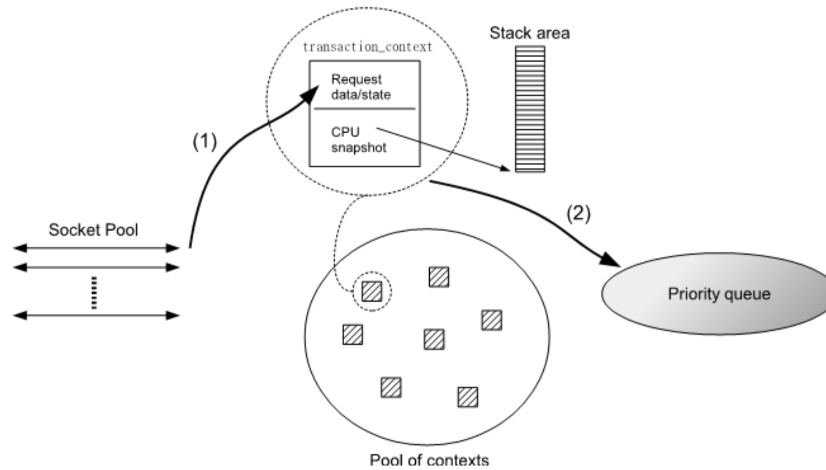

Figure 4.1. Basic architectural organization.

which is different for each `transaction_context` instance. Moreover, in order to avoid too many frequent calls to *malloc/free* functions offered by the GNU C standard library, we rely on an ad-hoc memory management system which draws from a pool of `NUM_CONTEXTS` data structures of type `transaction_context`. Once a transaction completes, the associated structure is released into the pool of contexts and again made available for reuse by the incoming transactions. Since our architecture aims to manage more contexts than worker threads processing transactional requests, we let `NUM_CONTEXTS` be a configurable parameter that should be set to a value significantly greater than the number of worker threads selected for running the STM application, enough for the purpose we have set ourselves.

A classical socket pool is handled in order to receive requests for executing data manipulations transactionally, which come in from some front-end system as hinted before. The job of receiving requests from sockets is done via dedicated server threads, whose execution profile is clearly I/O bound, thus not interfering in significant manner with the worker threads even in scenarios where the total amount of threads exceeds the number of available CPU-cores. Once a server thread receives a new request it queries the memory management system described before in order to get back a free `transaction_context`

structure, which will be initially filled with all values needed to successfully complete an execution such as the fields related to the specified transactional profile and the in-birth priority, then it proceeds to insert such a context into a *priority queue* structure, at the corresponding priority level. Overall, request insertion into the priority queue takes place off the critical path of the worker threads running the STM based application logic, which are therefore enabled to find the most up-to-date state of the priority queue every time a fine-grain periodical control flow variation occurs, so as to verify the need to pass control to some pending higher-priority request.

Such periodical control flow variation is based on the fine-grain timer-interrupts managed by the Linux extra-tick module discussed in section 3.2. These interrupts are issued exclusively towards worker threads, and lead to the activation of a user space module we refer to as `preemption_check()` registered by the worker thread itself as callback function during the setup phase. As soon as this function starts executing, the first thing it does is to check whether we are currently running in preemptable mode, verifiable by reading a thread-local-storage (TLS) variable we simply name `PREEMPTABLE`, in order to avoid the execution of the code block that implements the preemption management policies at the core of our STM environment when not needed. Conversely, if the `PREEMPTABLE` variable is set, then the code block implementing those policies must be executed. Let's recall that this callback function violates any *calling convention* dictated by the System V AMD64 ABI for architecture x86 and followed by the Linux systems, and the execution is actually at a point in which the same function has not been called by anyone, so that any operation performed before the `ret` instruction will dirty the registers' content, invalidating in this way the CPU-state to which the worker thread has undergone preemption. This is the reason why the `preemption_check()` function is completely coded in *assembly* programming language, so as to get full control on the registers we are going to use. The content of any

used register plus the flags' one is temporary saved in the stack and restored only at the end, immediately before the `ret` instruction is executed. The `preemption_check()` source code is reported in Appendix C for completeness. If a different transaction needs to take control of the CPU-core, the context of the currently processed transaction must be saved in its `transaction_context` data structure and enqueued again within the right priority queue, so that any worker thread will be able to restore it according to a *many-to-many user-level-thread* model. In the meanwhile, the context of the higher-priority transactional request (new or already suspended) is installed so that the worker thread can start processing it.

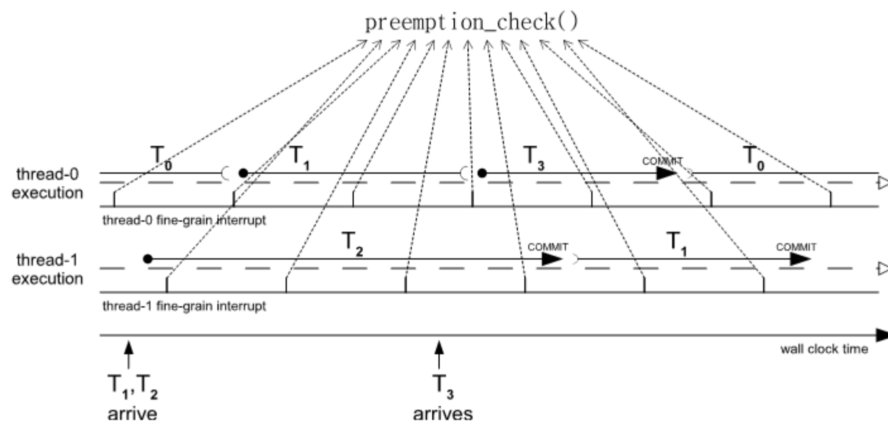

Figure 4.2. Fine-grain interrupts timeline.

We shall now discuss how it is possible to explicitly switch among two different transactional contexts. The main problem we had to face is related again to the already mentioned *calling conventions*, according to which the compilers are in charge of dividing general purpose registers between *caller-save* and *callee-save* registers. Unfortunately we cannot rely on the compiler at this point of the execution, which it reflects on the impossibility to use traditional implementations of `setjmp/longjmp` Posix API functions. In particular `setjmp` is regarded by the compiler as a function call, therefore any required caller-save register is pushed before issuing that call, allowing the

`setjmp` to store only callee-save registers. So, we have to relay on a different implementation of these two functions in order to still permit a correct control flow variation by saving all registers in the `exec_context_t` data structure which is part of the `transaction_context` data structure, and preserving in this way the transactional state that is about to undergo preemption. These functions, named `_set_jmp` and `_long_jmp` to recall the original ones, are introduced in [14] and we report the source code in the Appendix A.

In Figure 4.2 is shown a possible example of the fine-grain time-shared execution of in-memory transactions with only two worker threads. Here the two threads start executing the highest-priority transactional request found in the priority queue at that time, which will be either completed or suspended to give way to a possible pending higher-priority transactional request upon interrupt arrival. The example also shows how the transaction T_1 is in effect suspended while running along the thread-0 workflow and restored along the thread-1 workflow with a minimal delay. This lucky behaviour avoids the *vulnerability window* (the time elapsed between the begin and the abort/commit phases, during which the transaction would incur in possible conflicts) of the transaction T_1 to grow a paltry amount, and the *abort probability* (which is function of the number of data accesses, the read/write percentage, both the read/write and write/write affinities among concurrent transactions, and obviously the vulnerability window length) remains almost the same.

The remainder of this chapter is structured as follows. In Section 4.1 we discuss the setup of the contexts. The memory management is presented in Section 4.2. In Section 4.3 we show the priority queue in more detail.

4.1 ULT

We refer to the *execution context* as the program state at a certain point of the lifetime of the application. This state comprises both the *CPU image* represented by all the CPU registers, and the *program variables* currently residing in the data section, in the heap, and in the stack. It's therefore clear that the *execution context* is the abstraction of the `transaction_context` data structure included in our implementation that we introduced before. At each instant of time, the whole set of these values determines the state of the execution, from which it's deterministically known what is the next instruction to be executed, as well as the values on which the instruction performs. If we want to be able to restore a suspended transactional state, then all this information must be stored in special data structures that do not suffer side-effect due to control flow variation and consequent execution of a further transactional request. Rather, it must be possible to restore the whole context in a later time and resume execution as if it had never been suspended. In the Listing 4.1 is shown the data structure used to keep saved the state containing the aforementioned `exec_context_t` and `stack` fields plus an `stm_tx_t` address value which points to the structure dynamically allocated by the TinySTM to keep track of the TM-operations associated to the transaction. Since we want to handle a number of transactional contexts that is a value much greater than the number of CPU-cores available, but at the same time we cannot instantiate as many worker threads as there are contexts due to the issues discussed in Chapter 1, we cannot rely on the Pthread POSIX library to create new transactional contexts as well as *kernel-level thread* instances about which the Linux kernel is aware and therefore subject to the kernel schedule mechanisms that is not under our control. Rather we want to rely on a *user-level thread* (ULT) implementation on which our priority-based user-level scheduling policies have full control, so as to allow the user-level application to decide which is the next context to be scheduled. The model to which we

refers is the *many-to-many* one enabling any worker thread to take in charge of any transactional context, which may migrate from one workflow to another (example shown in Figure 4.2).

The initialization of the ULTs occurs during the setup phase of the entire application and relies on a revised procedure introduced in [6], which has its hub in the `context_create()` function reported in Appendix B. The goal of this function is to create an image of the initial state living in a different stack, which will be awakened only later, when the setup phase will be over and the system will be ready to serve transactional requests. This image finds its start at a predefined address within the `.text` section of the executable, an entry-point (function pointer) for the transactional routine.

```
1 typedef struct state {
2       exec_context_t      context;
3       void*               stack;
4       stm_tx_t*           stmtx;
5 } state_t;
```

Listing 4.1. State Data Structure

To do this, we rely on the POSIX-compliant `sigaltstack()` API, which asks the underlying operating system to run a signal handler named `context_create-_trampoline()` within a separate stack. Then, a call to the POSIX `raise()` API, with the parameter `SIGUSR1`, will give control to the signal handler mentioned before, whose first task is that of saving the current context, the new one living in a different stack, after which it gives back control again to the previous context from which we threw the signal. Saving the context is possible thanks to the revised `set_jmp()` function we already introduced. At this point, `context_create()` relies again on `sigaltstack()` to restore the previous setting, to indicate that the signal `SIGUSR1` is no more handled by the trampoline function. A subsequent call to the `context_switch()` macro (a

set_jmp() followed by a conditioned long_jmp()) makes sure that the control is given to the context saved before within the context_create_trampoline() procedure, resuming in this way the execution from the point at which it was suspended, that is a call to the ((noreturn)) labelled function named context_create_boot(). This function can thus create the starting conditions for the transactional routine, when this context will be resumed again later. Given that we are initializing an ULT, the job of allocating and reserving user-level memory space such as the stack is on us, and it is performed by the get_ult_stack() function through a malloc() invocation.

Furthermore, we have to emphasize the fact that contexts' creation is performed only during the setup phase so as to avoid the heavy process of constructing a new context on-demand, whenever a transactional request arrives into the system, which would clearly slow down the whole execution and impacting in this way on the performances. Rather, a transactional context that serves a transactional request during all its life-cycle, from the time at which it's accepted by one server thread to the time it will commit, it can be used again to serve another transactional request. In order to make this possible, the while-loop routine living within the function called for the first time by the context_create_boot() function must be structured to operate on renewed data structures at each loop (code block containing the body of the transactional profile that wants to access data such as the arguments needed by the profile itself). This is accomplished by relying on TLS variables, such as the running_task variable used to point to the task_t data structure (discussed in Section 4.2) associated to the ready-to-run transactional request, and which are updated each time by the worker thread right before to switch the context, from the *platform* context (the original context of the worker thread) to the one associated to the transactional request that is going to be served.

4.2 Memory Management

We already hinted how the architecture we developed is equipped with a memory management system to speed up the process of acquiring the resources when needed as well as releasing them when they are no longer so. A pool of `task_t` (shown in Listing 4.2) data structures is built by calling the function `TaskPoolInit()` during setup phase.

```
1  typedef struct task {
2      struct task*        next;
3      int                 conn;
4      int                 txid;
5      int                 prio;
6      char                args[256];
7      state_t             state;
8      int                 num_susp;
9      int                 susp_prio;
10     struct task*        next_free;
11     struct task*        next_gc;
12     int                 free_gc;
13     int                 aborts;
14     int                 commits;
15 } task_t;
```

Listing 4.2. Transaction Data Structure

This function takes only one argument as input, an integer value representing the pool size, and by mean of a `malloc()` invocation it allocates an array of `task_t` structures, whose fields are all initially set to the default value, such as the `next_free` pointer initialized to point to the next `task_t` structure in the array in order to make them all connected in a *free linked-list* which is protected by mean of a `pthread_spinlock_t` lock to guarantee that the *insert*

and *free* operations have a cost of O(1). These two operations are performed by mean of the GetTask() and FreeTask() functions and are relatively invoked by the server threads upon the arrival of a transactional request and once a worker thread has finished its task. To avoid synchronization tasks at the worker threads side, we allow a further list of task_t structures for each server thread, a sort of *garbage collection* connected with the next_gc pointers, each one initialized only once the server thread has removed the structure from the free list. The zero-ed free_gc integer field indicates to the server thread that this structure is still utilized by some worker thread and should not be considered for the re-insertion into the free list. Differently, when the worker thread will have finished its work, it will set the free_gc field to 1 so as to indicate that it is ready to be inserted into the free list.

```
1 struct task_pool {
2       int                      size;
3       task_t*                  array;
4       task_t*                  head;
5       task_t*                  tail;
6       pthread_spinlock_t       lock;
7 };
```

Listing 4.3. Task-pool Data Structure

Since the server threads are in charge of accepting and inserting the transactional requests into the priority queue which we will discuss in more detail in Section 4.3, they perform these tasks much more fastly than what worker threads can do. Thus, continuously exploring the *garbage collection* at each step could result useless in terms of resource utilization, and this task may also be performed after a predefined number of steps. We allow this feature be configurable at compilation-time by updating a define value with the number of desired steps. Even if the server threads are mainly characterized

by I/O-bound computation, this feature may result good enough when the preemptive STM environment is running on top of machines with limited resources, and the server/worker threads share the CPU-cores available.

The `state` field is a structure of type `state_t`, and it is initialized by the same function used to initialize the pool of `task_t` structures, which is therefore in charge of the setup of the transactional contexts by calling the `context_create()` function discussed in Section 4.1. For each `task_t` structure in the pool the functions `get_ult_stack()`, `context_create()` and `stm_init_thread()` are invoked in sequence to initialize the fields belonging to the `state_t` structure, such that when the setup phase is over and the system starts working as it should, nothing needs to be dynamically allocated and configured. In the Listing 4.3 is shown the data structure who keeps all the metadata needed to manage the pool and the free list.

4.3 The Priority Queue

We can finally explain in detail the Priority Queue we introduced in the previous sections. The priority queue comprises two `task_list_t` structures' array of size `NUM_PRIORITIES`, so that we have a couple of `task_list_t` structure for each priority level. A `task_list_t` structure by itself represents a linked list of `task_t` data structures, each one managed as a FIFO queue, on which it's possible to perform the insertion in the tail and the removal from the head by using the fields `tail` and `head` belonging to the `task_list_t` structure. The reason why the priority queue uses two array is because it has to provide a couple of lists <*active, standing*> for each of the managed priority levels. While the *standing* lists maintain the contexts of the transactional requests accepted and inserted just before by the server threads but not yet started, the *active* lists keep track of the contexts associated to those transactions already started but switched off the CPU in favour of an higher-priority transactional request.

The latter ones are also defined as *hot* contexts due to the fact that they could have already accessed shared data with high probability and therefore subject to possible conflicts with other transactions currently running in the system, in contrast to the former ones defined *cold* contexts, which are not yet started and consequently they cannot have accessed any shared data. This is the reason why we give priority to *hot* contexts into the *active* list with respect to the *cold* ones into the *standing* list when they are at the same priority level. By the way, every other higher-priority context still takes precedence over lower-priority ones, both *hot* and *cold*. In Listing 4.4 is reported a code snippet of the data structures used to implement the priority queue. As it can be easily seen in that snippet, two bitmaps teamed together with the lists, where every bit is used to indicate whether at least one context is appended into the related list or not.

```
1  typedef struct task_list {
2          task_t*                  head;
3          task_t*                  tail;
4          pthread_spinlock_t       lock;
5          int                      count;
6  } task_list_t;
7
8  struct prio_task_array {
9          int                      num_prio;
10         int                      num_bytes;
11         byte_t*                  stdn_bitmap;
12         byte_t*                  actv_bitmap;
13         task_list_t*             standing;
14         task_list_t*             active;
15 };
```

Listing 4.4. Priority Queue Data Structures

To perform as fast as possible the check on each bit of the bitmap, we rely on a bitwise operation that, once the byte has been loaded in a register, involves only a `shift` followed by an `and` machine instruction. Being the insertion and remove operations on a list two critical sections, these operations are protected with a `pthread_spinlock_t` lock associated to each list, while any needed update to the bitmap is performed through a CAS (compare-and-swap) operation on the byte. Once the lock has been acquired, it is possible to append a `task_t` structure in tail paying O(1), as well as it is possible to remove the `task_t` structure from the head of the list paying again O(1), which in turn make it possible to release the lock as soon as possible and available for further concurrent insertion/removal operations. Let's say that the bitmap has to undergo renovation in two cases, when insertion occurs into an empty list, and when the removal is performed on a list with only one element. With the growth in the number of concurrent operations on the tail, the likelihood that the CAS operation will fail increases as well. On the other hand, the increasing concurrency may depend by the growing arrival rate, which in turn it reflects in a filling of the lists and, as we said before, insertion/removal operations on lists containing more than one element does not require performing CAS, and therefore the need to update the bitmap proportionally shrinks.

Upon *commit* of a transaction, the worker thread that was in charge of the related context firstly marks the `free_gc` field so as to indicate this context is used no more and can be freed by the server thread, then it starts to search in the priority queue structure for another transactional context to be executed, the highest-priority one it finds and that it is able to remove. This task is accomplished by the `GetHighestPriorityContext()` function which reads the bitmap starting from the most-significant-bit representing the highest priority level and descending towards the least-significant-bit, but always giving precedence to the bit associated to the *active* list. Once a bit has been found set to 1 a subroutine called `try_remove_active_context()`/

`try_remove_standing_context()`, depending on the list type associated to that bit, is invoked so as to try to remove a context from that list. The meaning of the prefix *try* is that, even if we found the bit set to 1 it's not ensured we'll remove the element for sure because of the concurrency with other worker threads, and in negative case the routine returns and we continue to check the subsequent less significant bits. This immediately reflects on the way of acquiring the lock we chose, the `pthread_spin_trylock()` one, in such a way to either acquire the lock or check whether the list counter is expired, and in case leave the lock-loop and return without engaging the lock so as to let it free for possible concurrent insertions. However we can be sure that we'll eventually remove a different priority context by spinning on the priority queue as described so far. A possible and not lucky evolution could occur in case the worker thread does not find any context at the highest-priority levels and removes a lower-priority one, while a server thread is inserting a new higher-priority transactional request in the meantime. However, thanks to the fine-grain timer-interrupt provided by the extra-tick module we can revise the selection with an expected delay of half of the timer-interrupt period, so that we are still advantaged once by the use of extra-tick module in terms of reactivity to the event.

Chapter 5

Policies for Priority Management

We already discussed how the <*active, standing*> contexts are basically handled when no further policies are enabled. Higher-priority contexts take always precedence on the lower-priority ones, and the already started *hot* contexts have priority on the yet not started *cold* contexts at the same priority level. While this implicit policy tends to help already started transactions in order to avoid an excessive enlargement of the related vulnerability windows which directly reflects in a growing abort probability, it cannot help the stretch of the vulnerability windows caused by repeated context switches caused by the presence of higher priority requests within the priority queue. Whenever a low priority transaction undergoes extra-tick timer interrupt, it will mandatory leave the CPU in favour of a higher priority transaction even if one or more than one of these are present in the priority queue. This behaviour does not only give rise to an increasing vulnerability window, but also yields to a longer real execution time (time in CPU to complete) due to the growing abort probability, which is obviously the factor that mainly characterizes the number of aborts and rollbacks that a transaction undergoes before it can really commit. It's clear that there are a lot of inter-dependencies between all these

aspects which may drastically affect performances, especially for the lower priority levels, whereas they are not well handled by mean of good scheduling policies.

To cope with such a problem we have devised a feedback mechanism such that the actual priority level of an already started transaction is dynamically modified at run-time. This is possible by maintaining a counter $\mathtt{C_T}$ for each transactional context representing the number of times the transaction $\mathtt{T}$ has been context switched off the CPU. As soon as the value of $\mathtt{C_T}$ reaches a threshold that we denote as $\mathtt{C_{max}}$, then the transaction is migrated to the highest-priority level, so that no further delays caused by preemptions will be introduced on it. Clearly the benefits brought by this policy are different for different values of the threshold $\mathtt{C_{max}}$, as well as each of these values affects different priority levels in a different way. An high value of $\mathtt{C_{max}}$ obviously increases the aforementioned delays for the very low priority transactions more subject to context switches and therefore they are the ones who take mostly advantage from this policy, while the other priority transactions are able to complete before reaching the threshold. On the other hand, a low value of $\mathtt{C_{max}}$ tends to help all that transactions at a certain priority level that undergo context switch for few times, not only the lowest priority ones. The results obtained with experimental studies show us how to different values of $\mathtt{C_{max}}$ correspond different speed-up values for the various priorities. Furthermore, we also devised and implemented a variant by augmenting the same policy with a *lazy promotion*, an increase by 1 of the current priority level $\mathtt{P_T}$ whenever a transaction is context switched off the CPU, until the threshold has been reached

$$\mathtt{P_T} = \begin{cases} \mathtt{min(P_T + 1, P_{max})}, & \mathtt{if\ C_T \leq C_{max}} \\ \mathtt{P_{max}}, & \mathtt{otherwise} \end{cases} \tag{5.1}$$

where we denote with $\mathtt{P_{max}}$ the maximum admitted priority level within the

priority management scheme. All these policies have been integrated within the scheduler source code and shown in Appendix D.

One important final aspect to consider relates to how the extra-ticks delivered to threads needs to be handled in case they are received while the target thread is currently executing some function offered by the STM environment or the standard library, rather than native application code. We already mentioned in Chapter 4 how a TLS variable named `PREEMPTABLE` is kept by each worker thread in order to be able to know if we are currently executing application code or not. By reading this variable the extra-tick timer-interrupt handler may decide if it is the case of either operating a control flow variation and leave perform the user-level `preemption_check()` function, or setting a second per-thread TLS variable named `STANDING_TICK`, so as to delay the execution of the `preemption_check()` function at the time to which the non-preemptable code is ended. All this work to protect those

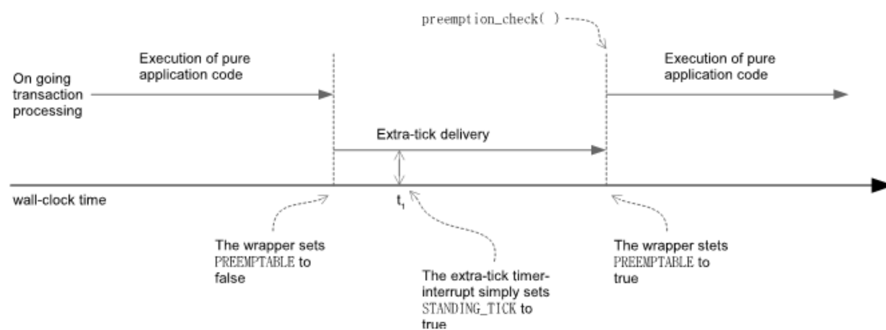

Figure 5.1. Standing ticks and time shift of preemptions.

functions that execute critical actions, such as the `TM_read` and `TM_write` services or functions provided by the standard library, and which should not be interrupted while executing their critical actions.

Lastly, the careful reader might express concern about the fact that a lower-priority transaction could never be served for the first time, and missing in this way the possibility to benefit of the policy mechanism. However, this behaviour may occur only in case the higher-priority arrival rate is so high to

avoid the lower-priority requests be eventually served, which finds the cause in an under-sizing of system capacity. In fact, in the process of performing our experimental study, once the hardware resources have been fixed, we have also chosen the right arrival rate in order to avoid such thrashing phenomena, by maintaining the system utilization at a value not so much high, but enough to simulate an high load of transactional requests. The per-scenario capacity planning task is however not under our investigation scope and we leave this job to the interested parties who want to get benefit from the preemptive STM architecture that we presented so far.

Chapter 6

Experimental Study

We run our preemptive STM environment on top of a 64-bit NUMA HP ProLiant server, equipped with four 2GHz AMD Opteron 6128 processors and 64 GB of RAM. Each processor has 8 cores, for a total of 32 CPU-cores, which share a 12 MB L3 cache (6 MB for each 4-cores set), and each CPU-core has a 512 KB private L2 cache. The operating system is OpenSuse 13.2 (Harlequin) (x86_64), with Linux kernel 3.16.7.

As hinted, our STM environment has been implemented by using a TinySTM [8], [15] as the baseline TM layer, and the whole package we developed is available for free download.[1] In our experiments, we used 16 worker threads in charge of processing transactions, and 5 server threads in charge of managing I/O operations on the socket pool and inserting incoming transactional requests into the priority queue. In this conditions we reserve no more than the 65% of the overall CPU-core capacity, in such a way to leave enough resources to the operating system to perform classical housekeeping operations without interfering with performance measurement of our STM environment. The client threads issuing transactional requests run on a different multi-core machine with the same technical specifications of the one hosting the STM environment, and connected via a switched 100Mb ethernet. The extra-tick interval in our

[1] https://github.com/HPDCS/PRESTO

preemptive STM system has been configured to 100 microseconds, a value definitely lower than the timer-interrupt period originally used in the configuration of the Linux kernel adopted, which was set to 1 millisecond. This value results to be the right trade-off between the non-excessive overhead cost associated to the extra-tick management logic, and responsiveness in detecting the arrival of higher-priority transactional contexts into the priority queue. Finally, the size of the context pool has been set to 1024, a value that enables keeping active a number of transactions highly larger than the number of worker threads processing them.

Table 6.1. Transaction profiles and associated priority levels.

transaction profile	CPU demand	priority level (the higher the better)
delivery	$\approx$ 5 msec	1
stock level	$\approx$ 650 μsec	2
new order	$\approx$ 350 μsec	3
order status	$\approx$ 10 μsec	4
payment	$<$ 10 μsec	5

Furthermore, it must be noted that transactions belonging to different profiles exhibit very different CPU demands, especially the ones provided by the TPC-C benchmark which range from ten of microseconds to milliseconds. This peculiarity has been exploited in our experiments in order to determine a transaction priority scheme where shorter running transactions are given higher priority as shown in Table 6.1.

We setup the workload generator to inject 25.000 transactional requests per second, issuing a total number of 6 millions of transactional requests along the experiment lifetime. Actually, this peak-load phase, evidenced by having the pool of contexts highly busy (above the 90%), is suitable for assessing the potential of an optimized preemptive CPU-dispatching scheme. The reported performance results have been computed as the average over three repetitions

of the experiment.

The actual performance results we want to show are represented by the whole time that a transactional request spends into the system, the *turnaround time* that goes from the time when the request is received and enqueued into the priority queue to the time at which it is really committed, and resulting therefore analogous to the sum of the execution time and in-queue time (both *active* and *standing*). Even if a transaction is aborted and then retried, any aborted run contributes to the turnaround time for the transaction.

In Figure 6.1 is shown the turnaround time for different configurations. The *baseline* plot refers to the execution without the extra-tick module active which implies no preemption for the transactional contexts that are able to give control to a standing higher-priority transaction only at the end of their executions, after the commit operation successfully completes and the worker thread may take in charge either a *hot* or a *cold* transactional context. For completeness of the analysis we also plot a second configuration that provides the extra-tick module enabled, but no-preemption is ever actuated, so as to underline the effective overhead cost compared to the baseline case. Finally, all the subsequent plots refer to the preemptive STM architecture assessed by considering different setting of the value C_{max}, and by either including or excluding the lazy priority promoting scheme for the management of the dynamic priority of the transactions. By the results we see how, compared to the baseline case, the preemptive approach reduces the average turnaround time of transactions born at higher priority levels (say levels 4 and 5) by around 60%-65%, while at the middle priority (say level 3), transactions exhibits an average turnaround latency essentially not penalized by preemption. Transactions born at lower priority levels (say levels 1 and 2) show a penalization of their average turnaround which is mostly limited to less than 5%, and no more than 15% in the worst case. As we discussed before, for higher values of C_{max} the majority of the transactions who benefit from the policy are those ones

born at the lowest-priority levels which are subject to a greater number of context switches, thus interfering less with the higher-priority transactions compared to the case with lower values of C_{max}. In order to better outline

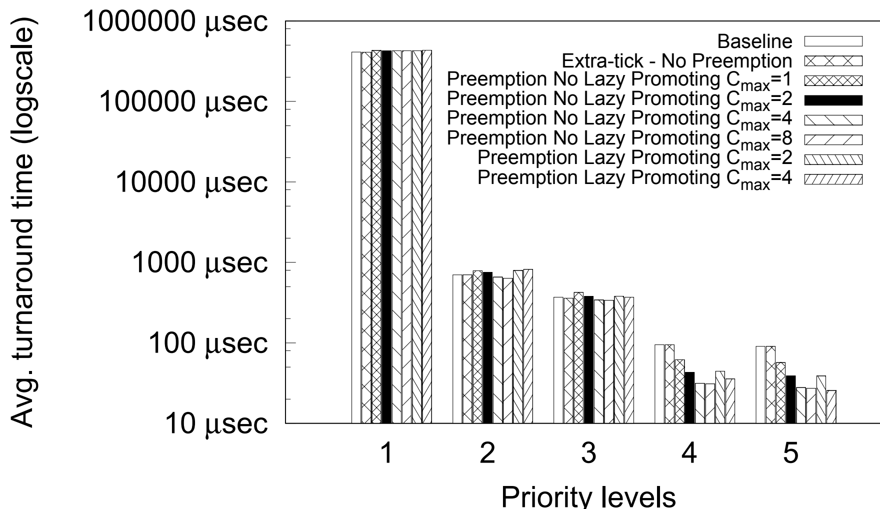

Figure 6.1. Average turnaround time for transactions born at different priority levels (log-scale on the y-axis).

the effects by the preemptive approach, we report in Figure 6.2 the ratio between the average turnaround latency provided by the baseline and the one provided by the preemptive approach. For this plot we selected the most promising configurations of the preemptive solution, based on the results shown in Figure 6.1. According to what we was discussing before we found the best solution in the configuration assessed with larger values of C_{max} (namely 4 or 8) for both approaches with no-lazy promoting and lazy promoting policies. In particular, the configuration with lazy promoting and C_{max} set to the value 4 is even able to provide higher speed-up (vs. the baseline) for the highest-priority level 5, with respect to the one not employing lazy promoting, that is possible due to the fact that, transactions born at priority level 1 dynamically acquire higher priority right after the first preemption and obtaining in this way a greater chance to complete before reaching the threshold C_{max}. Finally, in

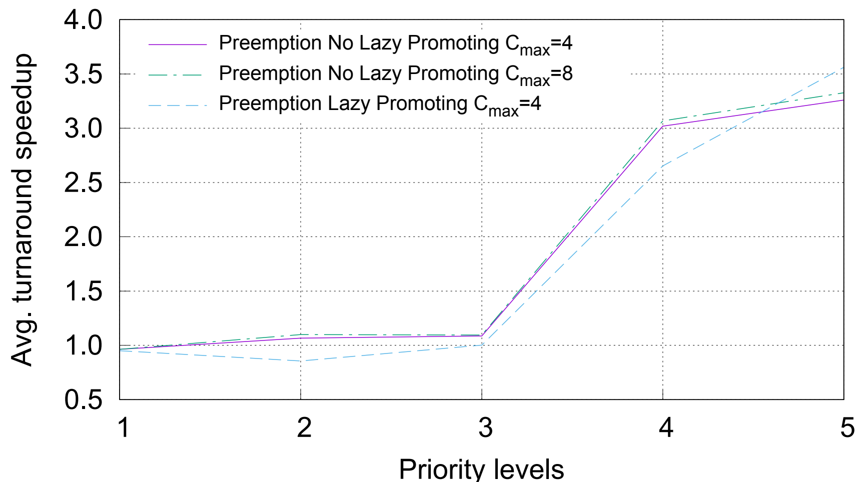

Figure 6.2. Speedup - Ratio between the turnaround time of the baseline configuration and the turnaround time of the preemptive configuration.

Figure 6.3 we report data indicating how the probability of abort varies in the different configurations. As hinted, this variation can be caused by the effects of preemptions on the length of the vulnerability window of the transactions.

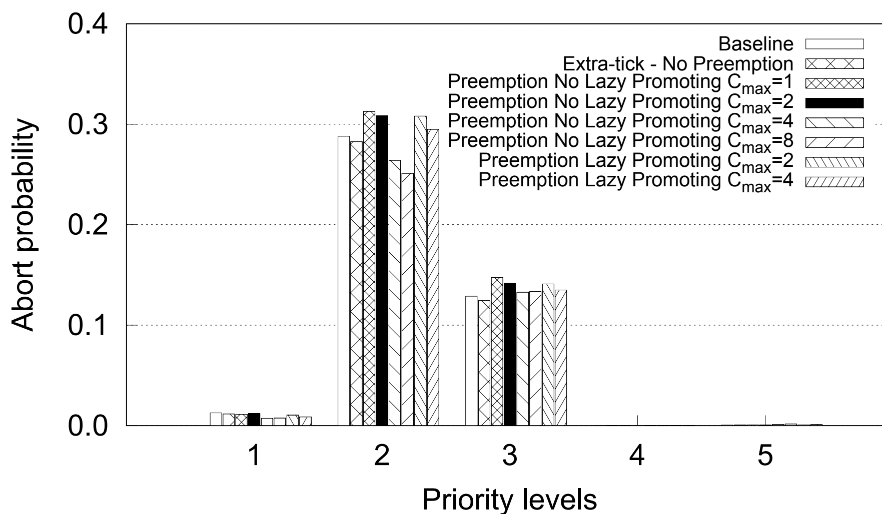

Figure 6.3. Variation of the transaction abort probability.

By the result we see that transactions born at priority level 2 show an increase of the abort probability for lower values of C_{max} and/or when lazy promoting is employed, and this is caused by the higher interference occurring when

transactions born at priority level 1 dynamically acquire higher priority, so as to lead to an increased concurrency between shorter transactions born at priority level 2 and the definitely longer ones born at priority level 1.

Chapter 7

Conclusions

The Software Transactional Memory (STM) environment we presented is the first attempt to provide preemptive capabilities while processing in-memory transactions, since state of the art STM implementations do not react as fast as possible to incoming higher-priority transactional requests as well as they do not provide transactional contexts management, rather they become aware of transactional requests delivered during the last transaction execution only after this one commits. Moreover we also introduced some policies to dynamically change the priority of transactions in order to give them more chances to take progress and eventually commit in case they undergo preemption too frequently. The results we obtained executing several long runs based on the TPC-C benchmark confirm the advantages, in terms of time spent by higher-priority transactions within the system, we expected from our preemptive STM environment with respect to the baseline implementation. Since in the execution of the baseline implementation we noted very high waiting time in relation to the expected execution time of the shortest transaction profiles, in our experiment we assigned priorities on the basis of CPU demand by the different transaction profiles, with lower demanding ones having higher priorities, a classical approach aiming at favouring shortest jobs.

Appendices

A. Context Save and Restore

Source Code: `jmp.S`

```
1  .file "jmp.S"
2  .text
3
4  .align 4
5  .globl _set_jmp
6  .type _set_jmp, @function
7  _set_jmp:
8         pushfq
9         pushq %rax
10        pushq %r11
11        movq %rdi, %rax
12        movq 8(%rsp), %r11
13        movq %r11, (%rax)
14        movq %rdx, 8(%rax)
15        movq %rcx, 16(%rax)
16        movq %rbx, 24(%rax)
17        movq %rsp, 32(%rax)
18        addq $16, 32(%rax)
19        movq %rbp, 40(%rax)
20        movq %rsi, 48(%rax)
21        movq 32(%rsp), %r11
22        movq %r11, 56(%rax)
23        movq %r8, 64(%rax)
24        movq %r9, 72(%rax)
25        movq %r10, 80(%rax)
26        movq (%rsp), %r11
27        movq %r11, 88(%rax)
28        movq %r12, 96(%rax)
29        movq %r13, 104(%rax)
30        movq %r14, 112(%rax)
31        movq %r15, 120(%rax)
32        movq 16(%rsp), %rdx
33        movq %rdx, 136(%rax)
34        movq 24(%rsp), %r11
35        movq %r11, 128(%rax)
```

```
36         fxsave 144(%rax)
37         addq $24, %rsp
38         xorq %rax, %rax
39         ret
40 .size    _set_jmp, .-_set_jmp
41
42 .align 4
43 .globl _long_jmp
44 .type _long_jmp, @function
45 _long_jmp:
46         movq %rdi, %rax
47         movq 128(%rax), %r10
48         movq 32(%rax), %r11
49         movq %r10, 8(%r11)
50         movq %rsi, (%r11)
51         movq 8(%rax), %rdx
52         movq 16(%rax), %rcx
53         movq 24(%rax), %rbx
54         movq 32(%rax), %rsp
55         movq 40(%rax), %rbp
56         movq 48(%rax), %rsi
57         movq 56(%rax), %rdi
58         movq 64(%rax), %r8
59         movq 72(%rax), %r9
60         movq 80(%rax), %r10
61         movq 88(%rax), %r11
62         movq 96(%rax), %r12
63         movq 104(%rax), %r13
64         movq 112(%rax), %r14
65         movq 120(%rax), %r15
66         pushq 136(%rax)
67         popfq
68         fxrstor 144(%rax)
69         movq 32(%rax), %rsp
70         popq %rax
71         ret
72 .size    _long_jmp, .-_long_jmp
```

Source Code: `jmp.h`

```
long long _set_jmp(exec_context_t* env);
__attribute__ ((__noreturn__)) void
            _long_jmp(exec_context_t* env, long long val);

#define set_jmp(env) ({\
      int _set_ret; \
      __asm__ __volatile__ ("pushq %rdi"); \
      _set_ret = _set_jmp(env); \
      __asm__ __volatile__ ("add $8, %rsp"); \
      _set_ret; \
})

#define long_jmp(env, val)     _long_jmp(env, val)
```

B. Context Creation

Source Code: `ult.c`

```
1  static void context_create_boot(void) __attribute__
2                                         ((noreturn));
3
4  static void context_create_boot(void) {
5      void (*context_start_func)(void*);
6      void* context_start_arg;
7
8      context_start_func = context_creat_func;
9      context_start_arg = context_creat_arg;
10
11     context_switch(context_creat, &context_caller);
12
13     context_start_func(context_start_arg);
14
15     assert(0);
16 }
17
18 static void context_create_trampoline(int sig) {
19     (void) sig;
20
21     if (context_save(context_creat) == 0)
22         return;
23
24     context_create_boot();
25 }
26
27 void context_create(exec_context_t* context, void (*
       entry_point)(void*), void* args, void* stack, size_t
       stack_size) {
28     struct sigaction sa;
29     struct sigaltstack ss;
30     struct sigaltstack oss;
31
32     memset((void*) &sa, 0, sizeof(struct sigaction));
33     sa.sa_handler = context_create_trampoline;
```

```
34         sa.sa_flags = SA_ONSTACK;
35         sigfillset(&sa.sa_mask);
36         sigdelset(&sa.sa_mask, SIGUSR1);
37         sigaction(SIGUSR1, &sa, NULL);
38 
39         ss.ss_sp = stack;
40         ss.ss_size = stack_size;
41         ss.ss_flags = 0;
42         sigaltstack(&ss, &oss);
43 
44         context_creat = context;
45         context_creat_func = entry_point;
46         context_creat_arg = args;
47         context_called = false;
48 
49         raise(SIGUSR1);
50         sigaltstack(&oss, NULL);
51 
52         context_switch(&context_caller, context);
53 }
```

Source Code: `ult.h`

```
1 #define    context_save(context) set_jmp(context)
2 
3 #define    context_restore(context) long_jmp(context, 1)
4 
5 #define    context_switch(context_old, context_new) \
6            if (set_jmp(context_old) == 0) \
7                long_jmp(context_new, (context_new)->rax)
```

C. Preemption Check

Source Code: `preemption_check.S`

```
1  .file      "preemption_check.S"
2  .text
3
4  .globl     preemption_check
5  .type      preemption_check, @function
6  preemption_check:
7          pushq          %rax
8
9          lahf
10         seto           %al
11         pushq          %rax
12
13         lock incl      tick_count(%rip)
14
15         movq           preemptable@gottpoff(%rip), %rax
16         movzwl         %fs:(%rax), %eax
17         cmpl           $1, %eax
18         jne            .L2
19         jmp            .L1
20
21 .L2:
22         movq           preemptable@gottpoff(%rip), %rax
23         movw           $1, %fs:(%rax)
24
25         movq           mode@gottpoff(%rip), %rax
26         movw           $1, %fs:(%rax)
27
28         pushq          %rdi
29
30         movq           running_task@gottpoff(%rip), %rdi
31         movq           %fs:(%rdi), %rdi
32         leaq           272(%rdi), %rdi
33
34         call           __set_jmp
35
```

```
36         testl          %eax, %eax
37         je             .L3
38         jmp            .L4
39
40  .L3:
41         call           schedule
42
43         movq           running_task@gottpoff(%rip), %rdi
44         movq           %fs:(%rdi), %rdi
45         leaq           272(%rdi), %rdi
46
47         movq           $1, %rsi
48
49         call           __long_jmp
50
51  .L4:
52         popq           %rdi
53
54         movq           mode@gottpoff(%rip), %rax
55         movw           $0, %fs:(%rax)
56
57         movq           preemptable@gottpoff(%rip), %rax
58         movw           $0, %fs:(%rax)
59
60  .L1:
61         popq           %rax
62         addb           $0x7f, %al
63         sahf
64
65         popq           %rax
66
67         retq
68  .size      preempt_callback, .-preempt_callback
```

D. Scheduler

Source Code: `scheduler.c`

```
1  void schedule() {
2      int prio;
3      task_t* task;
4      if (running_task == NULL) {
5              /* We are not currently managing a transactional
                   context. We will attempt to pick the highest
                  -priority one actually presents in priority
                  queue. */
6          if ((running_task =
7                  GetHighestPriorityContext(pta)) == NULL)
8              return;
9          thread_tx = running_task->state.stmtx;
10     } else {
11             /* We was managing a transactional context while
                   interrupted by the extra-tick. We will
                  attempt to pick a transactional context at a
                  priority higher than the current one. */
12         task = running_task;
13         prio = (task->susp_prio > -1) ?
14             task->susp_prio : task->prio;
15         if ((running_task = GetHigherPriorityContext(pta,
16             prio)) == NULL) {
17                 /* Come back executing the old transactional
                       context. */
18             running_task = task;
19             return;
20         }
21         thread_tx = running_task->state.stmtx;
22 #ifdef POLICY
23         /* Pre-processor directive to include scheduler
               policies. */
24         if (task->num_susp+1 < MAX_SUSPENSIONS &&
25             prio < pta->num_prio-1) {
26             /* We have reached the threshold Cmax. */
27 #ifdef LAZY_PROMOTING
```

```
28              /* Pre-processor directive to include lazy
                   promoting. */
29              InsertStandingContextAtPriority(pta, task,
30                  prio+1, ((first_tx_operation == FIRST_DONE)
31                      ? 1 : 0));
32 #else
33              /* No lazy promoting. */
34              InsertStandingStandingContextAtPriority(pta,
35                  task, prio, ((first_tx_operation ==
36                      FIRST_DONE) ? 1 : 0));
37 #endif
38          } else {
39              /* We have not yet reached the threshold. */
40              InsertStandingContextAtPriority(pta, task,
41                  pta->num_prio-1, ((first_tx_operation ==
42                      FIRST_DONE) ? 1 : 0));
43          }
44 #else
45          /* No policies employed. */
46          InsertStandingContext(pta, task,
47              ((first_tx_operation == FIRST_DONE) ? 1 : 0));
48 #endif
49      }
50 }
```

Bibliography

[1] Wikipedia: Transactional memory (2016). Available from: `https://en.wikipedia.org/wiki/Transactional_memory`.

[2] Di Sanzo, P., Sannicandro, M., Ciciani, B., and Quaglia, F. Markov chain-based adaptive scheduling in software transactional memory. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016*, pp. 373–382 (2016).

[3] Didona, D., Felber, P., Harmanci, D., Romano, P., and Schenker, J. Identifying the optimal level of parallelism in transactional memory applications. *Computing*, **97** (2015), 939.

[4] Diegues, N., Romano, P., and Garbatov, S. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pp. 224–233. ACM (2015).

[5] Dragojević, A. and Guerraoui, R. Predicting the scalability of a stm: A pragmatic approach. In *Presented at: 5th ACM SIGPLAN Workshop on Transactional Computing, 2010* (2010).

[6] Engelschall, R. S. Portable multithreading: the signal stack trick for user-space thread creation. In *Proceeding ATEC '00 Proceedings of the annual conference on USENIX Annual Technical Conference*, pp. 20–20 (2000).

[7] Ennals, R. Software transactional memory should not be obstruction-free. Technical report, Intel Research Cambridge (2006).

[8] Felber, P., Fetzer, C., and Riegel, T. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pp. 237–246. ACM (2008).

[9] Guerraoui, R. and Kapalka, M. On the correctness of transactional memory. In *Proceeding of 13th ACM SIGPLAN Symposium on Principles and practice of Parallel Programming (PPoPP '08)*, pp. 175–184. ACM (2008).

[10] Harris, T., Larus, J., and Rajwar, R. *Transactional Memory, 2nd Edition*, chap. 1, pp. 1–15. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2 edn. (2010).

[11] Herlihy, M. and Shavit, N. *The Art of Multiprocessor Programming*, chap. 18, pp. 417–433. Morgan Kaufmann, 1 edn. (2012).

[12] Maldonado, W., Marlier, P., Felber, P., Lawall, J., Muller, G., and Rivière, E. Supporting time-based qos requirements in software transactional memory. *ACM Transactions on Parallel Computing, Association for Computing Machinery*, **2** (2015).

[13] Papadimitriou, C. H. The serializability of concurrent database updates. *J. ACM, Association for Computing Machinery*, **26** (1979).

[14] Pellegrini, A. and Quaglia, F. Time-sharing time warp via lightweight operating system support. In *Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation*, pp. 47–58 (2015).

[15] Riegel, T., Felber, P., and Fetzer, C. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th international conference on Distributed Computing (DISC '06)*, pp. 284–298. Springer-Verlag (2006).

[16] Rughetti, D., Di Sanzo, P., Ciciani, B., and Quaglia, F. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 278–285 (2012).

[17] Rughetti, D., Di Sanzo, P., Ciciani, B., and Quaglia, F. Analytical/ml mixed approach for concurrency regulation in software transactional memory. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014*, pp. 81–91 (2014).

[18] Rughetti, D., Romano, P., Quaglia, F., and Ciciani, B. Automatic tuning of the parallelism degree in hardware transactional memory. In *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, pp. 475–486. Springer International Publishing (2014).

[19] TPC Council. *TPC-C Benchmark, Revision 5.11* (2010).

[20] Yoo, R. M. and Lee, H. S. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pp. 169–178. ACM (2008).