SAPIENZA
UNIVERSITÀ DI ROMA

Micro-Threading: Effective Management of Tasks in Parallel Applications

Department of Computer, Control and Management Engineering

Dottorato di Ricerca in Engineering in Computer Science – XXXIII Ciclo

Candidate

Emiliano Silvestri
ID number 1101255

Thesis Advisor

Prof. Francesco Quaglia

Co-Advisor

Prof. Leonardo Querzoni

27 January 2021

Thesis not yet defended

Micro-Threading: Effective Management of Tasks in Parallel Applications
Ph.D. thesis. Sapienza – University of Rome

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

Version: January 27, 2021

Author's email: silvestri@diag.uniroma1.it

Abstract

Modern parallel applications, to be run on top of multi-core systems, are ever more characterized by the presence of many differentiated activities, which can be (re-)dispatched on different elaboration units—at disparate wall-clock-times—in order to make parallelism effective. Consequently, several programming models (or environments) for multi-core shared-memory architectures give to developers the capability to indicate what portions of the application must be treated as tasks, which can then be run in parallel. However, once these tasks have been CPU-dispatched they are executed in a non-preemptible manner until they spontaneously re-interact with the specific runtime layer hosting the application, a possibility for which there is no guarantee on when it will occur again at run-time. In the meantime, sudden program state changes can always arise as a consequence of the effects that each of these tasks can have on the program evolution throughout its execution. Operating System classical preemption does not fully cope with this problem—given that the timeline for this preemption procedure is general purpose and not necessarily suited for a given application context—and may be attempted to be exploited limited to scenarios where multiple tasks are assigned for being processed to different threads, which is not the optimal case for most application domains. Also, in some application fields we also have the exploitation of speculation (giving rise to potential causality inconsistencies), which is something not dealt with by the Operating System. As a result, whenever one of such (speculative) changes in the application state occurs, the application's current execution dynamics may already be far from the optimal ones, which is a condition that can affect its performance in a significant manner. This must be addressed through timely re-assessments of the work assigned to the underlying computing resources regardless of whether the application tasks were intended to release the CPU at that time. To cope with this problem, we propose a new execution model for tasks, called Micro-Threading model, which provides for application-transparent task interruptions at arbitrary points of their executions. These interruptions are aimed at re-evaluating the current task-to-CPU assignments, in a manner fully alternative with respect to the thread-to-CPU assignment established by the Operating System. Also, we provide an implementation of this model to support task-preemptive execution in a wide range of applications contexts deployed on top of x86 machines with Unix-like Operating Systems. These include Transactional Memory, OpenMP and speculative Parallel Discrete Event Simulation. Clearly, the exploitation of micro-threads in these contexts has also led us to introduce new algorithms and solutions suited for optimizing the linkage of the micro-thread based execution to application level specific features. These proposals form a kind of reference to be considered for the exploitation of micro-threads in application scenarios related to the specific ones considered in this thesis. The results obtained by the experiments we carried out confirm the capability of our proposal to provide better run-time dynamics thanks to higher reactivity to program state changes, which is reflected in promptly renewing the overall scheduling of tasks to CPU-cores and/or in re-assessing the execution trajectory of each single task whenever is deemed counterproductive for the performance of the application as a whole.

Contents

List of Tables

List of Figures

List of Algorithms

Listings

Chapter 1

Introduction

Over the past three decades, CPU architectures have evolved rapidly by gaining an ever increasing capacity to carry out jobs in parallel. These changes took place at different levels of the hardware design with effects on both the amount of data that can be processed at a time and the number of instructions that is possible to simultaneously accommodate inside a single computing unit. We are referring to *bit-level parallelism* (BLP), widely exploited thanks to the doubling of the number of bits that a single unit can handle, and *instruction-level parallelism* (ILP), achieved with the implementation of more complex techniques such as instruction pipelining and redundant functional units, which are typical of superscalar architectures. The main goals of these hardware implementations is to achieve higher performance by keeping the number of committed instructions per cycle (IPC) as close as possible to one, which, along with other techniques (*e.g.*, out-of-order execution and branch prediction), attempt to hide the excessive latencies that can arise from read and write operations from/to memory. Nevertheless, it was immediately clear how data dependencies, affecting subsequent instructions belonging to the same execution flow, imposed strict limitations to the exploitation of ILP—a phenomenon known as the *ILP Wall*—so that chip manufacturers have started to move their attention to other solutions. This was further exacerbated by the impossibility of facing the problem of power dissipation—also known as the *Power Wall*—which has emerged when the continuous scale-up of clock frequencies, supplied voltages and transistor densities, characterizing the traditional processor architecture improvements in the first decade of this century, finally appeared to have broken down. This break in the trend of improvements has led vendors to focus the hardware design on multi-core processors as an alternative way to earn more in performance. In this regard, they started to produce CPU architectures equipped with more processing units named CPU-cores, each one capable to carry on an independent execution flow. This has definitively gave way to the possibility of exploiting *thread-level parallelism* (TLP).

This architectural shift signed in its turn a radical change in the world of software design and implementation, in that new ways of structuring programs have appeared in order to exploit multi-core hardware resources—an immediate practice has been to identify which parts of the application can be executed in parallel and then give to the program a structure consistent with this possibility of parallelism exploitation. As a consequence, this technological improvement also imposed changes in the design

of the software components that form the environment within which the applications actually live. Figure 1.1 shows a simplified scheme of a software environment deployed on top of a multi-core shared-memory architecture.

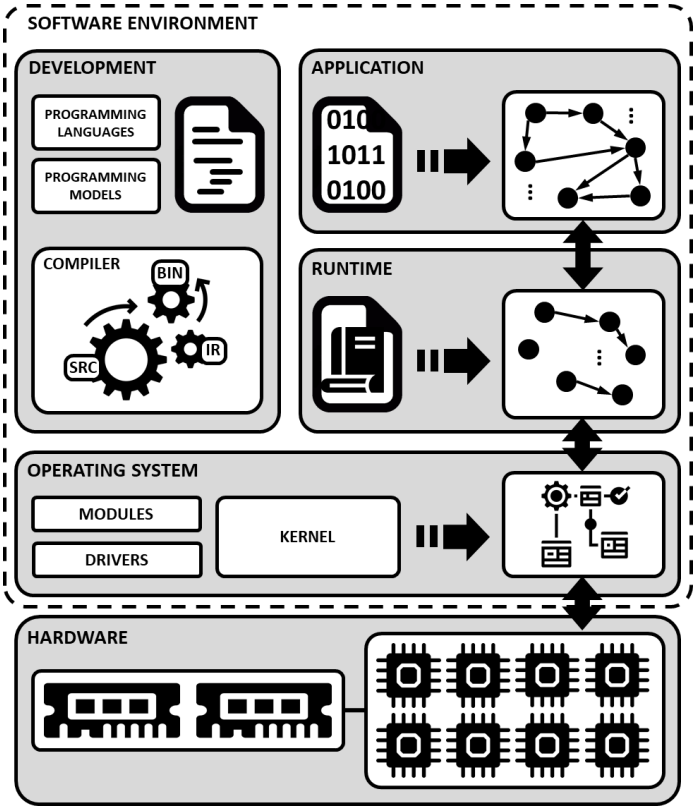

Figure 1.1. Software environment.

At the lowest level of the stack we find the operating system (OS), which is the most important software that acts as an intermediary between the application and the underlying resources. It provides the most basic level of control over all of the hardware components and offers common services to the programs. As for the latter, the OS simplifies the execution of programs by creating processes or threads and by performing their runtime control, it makes the setup of the address spaces and links the applications' code with other software components (*i.e.*, shared libraries that will be part of the runtime) which ultimately provide the interfaces through which real activation of the OS services can take place—these are the so-called *system-calls*, whose invocation occurs by following conventions that differ from those provided for pure application level software. This is the reason for which the aforementioned interfaces are commonly implemented in the form of assembly stubs, whose implementations clearly depend on the target system, hence they must be compatible with the underlying OS and hardware architecture. Also, the OS is in charge of supporting the execution of multiple independent programs (multitasking), or threads related to one or more processes (multithreading), by slicing the CPU time and dedicating a slot to each of them—also taking care of the peculiarities of multi-core machines when implementing CPU-scheduling policies

like load balancing [80] or memory management policies like NUMA (Non-Uniform Memory Access) oriented ones [28]. In this regard, the action of suspending one process/thread execution in favor of another is accomplished through the activation of the OS scheduler which, among all the other things, takes decisions about the next process/thread to dispatch on CPU. Such an activation can take place either upon direct invocation of a system-call by the application, or more likely at time intervals. The latter is made possible by exploiting programmable registers hardwired to each CPU-core which implement timer functionalities. Upon the expiration of these timers, the control is immediately given to the OS by following the same procedure that is used for handling all the other interrupts—common configurations in modern OS provide these registers be initialized with time periods that range from a few to several milliseconds. Lastly, since multiple programs and services are allowed to execute in parallel into the same multi-core system, both performance and security aspects need to be addressed in order to avoid that the sharing of hardware resources will lead processes to interfere with each other. In this regard, the OS implements an efficient management of physical frames in main memory together with virtualization techniques that ultimately provide these programs with separate views of the environment.

Immediately above the OS, we find the user-space runtime system which is responsible of performing activities not directly coded into the application itself. Despite some definitions also include the set of instructions inserted by the compiler to implement the execution model of the language (*e.g.*, creation of the stack, the copy of function-call parameters, etc.), we mainly want to highlight those pieces of software that are included into the environment at link time. These modules provide the applications with interfaces that transparently lead to the activation of environmental services (which may ultimately make use of system-calls), along with additional facilities that efficiently mediate the interactions with, *e.g.*, the OS. The latter are intended to address a number of issues including efficient management of virtual memory, improved I/O solutions for the exchange of data between user- and kernel-space, creation and control of threads, and others.

It is clear therefore how the application programmers that want to exploit hardware parallelism need to rely on the abovementioned facilities in order to achieve the desired behaviour. On the other hand, the programmer is completely unaware of when and where the execution of processes and threads will take place, this decision is left to the OS. The programmer only has to worry about whether the application's behaviour is correct, even when different parts of it are allowed to execute in parallel. It is expected indeed that the execution of parallel programs provides the same outcomes of the original (non-parallel) versions. As for this aspect, it is common that several activities, which are candidates for parallel execution, might not be really independent because of data dependencies that ultimately impose strict ordering constraints. Furthermore, there could be sub-parts of parallel activities that mandatorily need to be performed atomically in order not to incur race conditions. To this end, the programmer is in charge of employing the proper synchronization techniques, and more generally the management techniques, offered either by the runtime system or at the OS level in order to be sure that the application execution does not deviate from the expected behaviour. This explains why, over the years, parallel applications have become ever more sophisticated, so much that it is deemed

to be a hard task to develop and to deploy them on a computer system without the support of a well designed software environment. At the same time, such an articulated stack of software components offers many advantages. In the first place, it simplifies the programmer's life who is made definitively free from the need of implementing many complex functionalities already provided by the underlying (environmental) software layers. Secondly, it makes viable some other crucial aspects such as portability and reusability of parallel application software—just like when relying on traditional standard programming libraries in order to deal with OS or hardware heterogeneity. Lastly, it makes the execution of all programs secure, in that they have no possibilities to interfere with the activities of the others and with those of the OS.

In this broad overview we discussed the advantages provided by software environments in terms of developing and deploying parallel applications onto whatever hardware architecture. On the other hand, we must notice the strong differences existing between the goals that the application and the OS are intended to achieve. On the one hand, the OS provides fair exploitation of hardware resources on behalf of all programs that run into the system, according to a general purpose mode. Conversely, it does not take any scheduling decision aimed to optimize per-process parallel execution depending on the current program state or structure, nor it has the capability to infer the application semantics in order to pursue this goal. On the other hand, the applications are expected to provide all the logic required to control their execution dynamics in the most suitable way, which is usually implemented in the form of specific function-calls appositely placed at fixed points in the code, upon the execution of which program state changes are detected and the execution trajectory of threads is revaluated accordingly. This is a fundamental aspect also considering that modern parallel applications are ever more characterized by the presence of differentiated and fine-grained activities whose executions (and scheduling) are completely under the control of the application logic. Indeed, the latest trends in the design of parallel applications evidence that these applications are designed according to a scheme where threads are not devoted to specific activities, rather they can take care of running whatever activity is required to reach the application completion (or specific application goals). Hence threads are essentially workers that can take care of carrying out the actual work to be done. The latter comes in the form of the so-called *tasks*, which are units of work that are no longer statically tied to the execution of a specific thread, rather they can be taken in charge by any one of them [74]. These tasks are usually associated with different functions or code blocks, and can have therefore different execution profiles. Precisely for this reason, the workload profile of these applications is continuously subject to variations as there does not exist a specific pattern characterizing their execution over time (unless for very regular applications). This is further exacerbated by the fact that, in almost all cases, tasks dynamically materialize at runtime (*e.g.*, because of interactions with an application-external source or of non-deterministic evolution of the application itself), which definitively does not allow to provide a precise characterization/expectation of the timing of their execution, as well as to predict the next time at which the activation of the logic for controlling their execution dynamics will take place.

Under this uncertainty and dynamism, we have no guarantees of the existence of temporal bounds within which threads can react to sudden program state changes

(as noted, the OS does not fully cope with this need). The latter can occur so quickly that most of the time could be spent performing sub-optimally before the application can again have the chance to reassess the execution trajectory of threads (with respect to pending tasks to be processed). In other words, the execution performance of a parallel program relying of several threads to carry on the execution of its tasks is ultimately determined by the execution conditions of each single thread participating to the evolution of the application itself, which in turn are determined by the current execution state of the program as a whole intended as the entire set of activities that are currently ready to be executed and those that have been already assigned to other threads, as well as their characteristics. In this regard, the execution state of a parallel program may constantly be subject to sudden changes as the threads participating to the parallel execution may always affect it by, for instance, generating new tasks with characteristics that differ from those characterizing the currently performed ones, thus requiring the actual schedule to be renewed in order to follow better execution dynamics than those that have suddenly materialized upon the occurrence of such changes, which clearly does not implies a re-evaluation of the assignment of work to the only thread that has caused them. Sometimes, program state changes resulting from operations performed by a certain thread, hence by the task it has currently taken in charge, can also affect the usefulness of the work currently carried on by other threads which mandatory need to timely detect them in order not to incur the risk of pursuing ahead work that will reveal unfruitful for the whole execution of the application.

There exists a wide range of parallel applications for which the program state and the execution state of threads rapidly evolve due to causes that are not predictable. Typical examples are those parallel applications whose activities have been assigned different priority levels. This because some of them are identified by the programmer to be more critical than others, such that a timely start of their execution also means positive performance implications. By the way, priorities are probably the most powerful tool provided to programmers in order to indicate that a subset of tasks deserves more attention and immediate cooperation of threads in carrying out certain activities as soon as possible. In this regard, the programmer would like the computing power to be immediately lent to serve the highest priority tasks upon their creation, whose reactivity to start the execution determines the performance level (or the application effectiveness level) that can be achieved. Differently, any occurrence of priority inversion will lead the application to perform under non-optimal conditions. In addition, tasks may also be subject to waiting conditions deriving from dependency constraints, for which threads that carry on their execution are allowed to suspend them in favor of other tasks so as not to incur in blocking conditions. However, suspended high priority tasks are required to promptly resume as soon as their dependencies are satisfied, otherwise priority inversions may still adversely affect the execution of the application. Therefore, it is clear that, in order to achieve the most efficient management of task priorities the last two points need to be solved through solutions that allow prompt activation of the logic used to control the assignment of tasks to threads, as any additional delay in starting newly created high priority tasks, or resuming the ones blocked because of dependencies, may lead to sub-optimal execution scenarios.

Other examples of applications where the timeliness of suspending/resuming

the execution of tasks, or of the assignment of tasks to threads, is critical are those allowed to carry out tasks (and more generally computation) speculatively. This is an attractive paradigm for parallel computations—especially in irregular parallel applications with data dependencies—thanks to its high potential for actual parallelism exploitation and scalability, as compared to parallel execution methods not entailing speculation [42]. In these contexts, tasks are usually associated with functions or code blocks whose execution is just an attempt to make permanent the effects generated by the performed operations. If successful, any committed operation is an actual update to the program state, otherwise all the effects must be undone as if the operations never took place. Somehow, any change in the program state can also affect the validity of one or more tasks which are running speculatively along the execution path of concurrent threads. Just for this reason, prompt activation of the logic that is used to check the execution validity—hopefully, as soon as an invalidation has occurred—would be fundamental in order not to incur the risk of wasting computing power, in terms of both time and energy. It is instead possible that tasks have to execute for long time before threads have again the chance to reach the point where to call the routine conceived to reassess the task validity—this just depends on the application logic structure and on how the calls to these routines are nested by the programmer—while the causes that make tasks no longer valid can occur at any time as an effect of thread concurrency plus speculation. The timing according to which these checks take place clearly depends on the granularity of tasks, but also on the occurrence of certain events that force threads to perform validation activities. Nevertheless, in generic application contexts we have no guarantees on how long it will take before these controls will be triggered, even if we demand this facility (*e.g.*, the activation of a handler targeted at the control) to the general purpose support by the OS—in fact the activation of signal handlers in the Posix domain are delayed up to the next kernel to user mode switch along the thread. Clearly, this is an intrinsic aspect of speculative executions, for which it would be unreasonable to ask the programmer to explicitly include routine-calls to perform a validation of the execution state synchronously with respect to the execution of concurrent tasks. In fact, this would hamper speculation and actual parallelism exploitation. Furthermore, any synchronous invocation to these routines would anyhow require the thread to reach the point of their calls, which might give rise to unpredictable delays in general application contexts, also because of the fact that the actual path of machine instructions to be executed before reaching that point is typically unknown to a programmer who mostly rely on higher level languages. Differently, it would be desirable to trigger these checks transparently to the application logic—asynchronously—and hopefully only when (or right after) certain events have occurred, those that can have effects on the validity of current execution or those that have introduced a causality violation. A similar behaviour does not only allow to limit the task management overhead, spending cost just at the points of execution where it is likely or certain that an invalidation has occurred, but it also avoids that the execution of tasks already doomed to rollback goes on for non-minimal CPU-time (just depending on the execution profile of the application logic).

Since the occurrence of program state changes is clearly unavoidable, the run-time environment should provide the applications with an adequate support that

transparently allows their executions to timely realign with suited dynamics (*e.g.*, task suspend/resume/squash or task to thread dynamic re-association). This is a critical performance aspect, which deserves to be addressed with lightweight and fast techniques in order to achieve the desired application behaviour at runtime.

Overall, parallel programs running on multi-core machines need to be supported with environmental solutions that enable the possibility to control the execution dynamics of tasks at fine grain, and that must be decoupled from the application logic and from the timing according to which this logic is activated and evolves along time. Starting from these last two observations, we can draw three main conclusions about the actual limitations that currently prevent parallel task-based applications from reaching the aforementioned objectives:

- all tasks take place along the execution path of one or more threads, whose flows are constrained to follow the branches admitted by the *control flow graph* (CFG) generated for the application at compilation time—CFG is a representation of all paths that might be synchronously traversed through a program during its execution;
- execution of one thread is not allowed to asynchronously slide out from the CFG in order to start the execution of different code without incurring the risk of compromising the execution state of the abandoned path—there are functions (*e.g.*, `setjmp/longjmp`) that provide the implementation of synchronous non-local jumps that also reestablish thread state but for which the execution is maintained correct only because the points where their invocations occur are known at compilation time. In any case, their exploitation for the purpose of task management and control is still subject to the fact that the application logic needs to reach the point of a call to these functions, which might be unpredictably delayed just depending on the application execution profile and its input data, among others;
- there not yet exist transparent techniques for triggering the execution of code not directly known to be placed along the execution path of threads at compilation time regardless of the threads' current execution point—signaling mechanisms (like Posix signals) allow to achieve this behaviour but still require an effort by the programmer and more important the intervention by the OS with time constraints that would result ineffective for our purposes[1].

The research work carried during my Ph.D. career has lead to solutions overcoming these limits. This has been achieved by designing and implementing innovative software components that we have installed at different levels of common software environments used to support parallel applications in differentiated contexts. This is achieved without altering the software stack structure proper of the target context, in terms of programming model offered to the developers, as we have already underlined its importance in terms of easiness of applications' design and deployment. With the aid of these solutions, the applications receive transparent support for the achievement of the objectives we discussed so far, namely:

[1]The actual processing of a Posix signal is delayed up to the next system-call return or the next user-space return after a CPU-reschedule of the target thread.

- very fine grain control (and possible preemption) of the execution of tasks, including asynchronous control;
- very prompt reaction, in terms of tasks to thread assignment or task suspension/resume, depending on program state changes;
- very prompt reactions to causal inconsistencies among concurrent tasks with data dependencies, such as early squash of doomed to abort tasks.

The aforementioned capabilities have been included in differentiated runtime environments, ranging from transactional memory systems, to discrete event simulation systems, to pure task-parallelism environments. In more detail, we devised our solutions for integration with the TinySTM open source package [21], a well known enviroment largely used for the assessment of innovations by the side of management of in-memory transactions, the USE (Ultimate-Share-Everyting) Paralle Discrete Event Simulation (PDES) open source package [36], a last generation environment supporting extremely efficient speculative parallel execution of Discrete Event Simulation (DES) models on multi-core machines, and the GNU OpenMP package [27], a renown software platform hosting task-parallel applications developed according to the OpenMP specification [62]. All these environments have been devised for C-based applications—hence being naturally oriented to high runtime effectiveness. This intrinsically leads to scenarios where optimized solutions for the runtime management of applications to be run on multi-core machines, like the one we propose in this thesis, play a central role. Overall, a common framework has been designed and developed for all the target application contexts, which is based on the notion of *micro-threads*. These are work units corresponding to portions of the whole execution trace of an OS managed thread, which are in their turn controlled (in terms of, *e.g.*, actual execution along the thread) by mechanisms that stand aside of the ones used by the OS. On the other hand, these mechanisms do not interfere with the ones adopted by the OS software for handling resources, such as the assignment of CPU to threads. This leads to minimal intrusiveness and high versatility. Micro-threads can be driven according to differentiated policies, which are explored in this thesis. Furthermore, although our design has been targeted at the Linux OS, its underlying principles can be easily ported to other OS flavours. Beyond such common framework, the thesis also provides innovative architectural solutions and algorithms suited for the exploitation of micro-threads in the target applications contexts. On the other hand, these contexts can be seen as archetypal (*e.g.*, in terms of employment of differentiated task priorities, or data dependencies among concurrent tasks). Hence, our proposals can be anyhow considered as solutions possibly exploitable in other contexts showing similar features, in terms of rules and policies for the management of tasks.

The rest of this thesis is structured as follows. In Chapter 2 we discuss about differences existing between our solution and the OS-level CPU scheduling of threads by also providing references to literature. In Chapter 3 we introduce the concept of Micro-Threading and its characterization for application-specific scenarios. In Chapters 4 and 5 we discuss about two different approaches for accomplishing the commissioning of the Micro-Threading model, which involve the use of different hardware to achieve our objectives. As for the latter two chapters, the inner Sections

4.1, 4.2 and 5.1 are devoted to argument specific performance issues which can always arise if not assisted by adopting the approaches discussed in the relative chapters, thus the importance of employing such solutions in order to optimize the performance and/or satisfy specific requirements. Finally, in Subsections 4.1.1, 4.1.2, 4.2.1 and 5.1.1 we present the implementation of software architectures, integrating the proposed approaches, specifically designed to address the performance drawbacks discussed in the upper sections and the results obtained by the experimental evaluations carried out.

The work presented has given rise to the following original publications:

- **E. Silvestri**, S. Economo, P. di Sanzo, A. Pellegrini and F. Quaglia (2017). Preemptive Software Transactional Memory. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017* (pp. 294–303). IEEE Computer Society / ACM.

- S. Economo, **E. Silvestri**, P. di Sanzo, A. Pellegrini and F. Quaglia (2017). Prompt application-transparent transaction revalidation in software transactional memory. In *16th IEEE International Symposium on Network Computing and Applications, NCA 2017, Cambridge, MA, USA, October 30 - November 1, 2017* (pp. 157–162). IEEE Computer Society.

- **E. Silvestri**, C. Milia, R. Marotta, A. Pellegrini and F. Quaglia (2020). Exploiting Inter-Processor-Interrupts for Virtual-Time Coordination in Speculative Parallel Discrete Event Simulation. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS 2020, Miami, FL, USA, June 15-17, 2020* (pp. 49–59). ACM.

- **E. Silvestri**, A. Pellegrini, P. Di Sanzo and F. Quaglia (2020). Effective Runtime Management of Tasks and Priorities in OpenMP Applications. *Ready for submission to an international journal.*

The research work that lead to this thesis has also produced the following additional publication:

- S. Economo, **E. Silvestri**, P. di Sanzo, A. Pellegrini and F. Quaglia (2018). Model-Based Proactive Read-Validation in Transaction Processing Systems. In *24th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2018, Singapore, December 11-13, 2018* (pp. 481–488). IEEE.

Chapter 2

Relations with the Literature

Explaining what is the relation between the contribution presented in this Ph.D. thesis and the literature is an articulated activity. This is because the thesis has an initial part where new solutions and mechanisms are provided for the management of the execution flow of conventional threads, which are then exploited in specific application fields thanks to Micro-Threading. This exploitation is presented by also discussing why it oversteps results that are published in the literature and deal with the specific fields we are considering. Hence, a very large part of the literature, which is related to this thesis content, is actually discussed in subsequent chapters.

The initial contributions of the thesis, which are related to the instantiation and management of micro-threads, are clearly related to the state-of-the-art in the field of Operating Systems (OS), where the problem of how (and when) to reassign a CPU-core to a given activity that needs to be carried out has been studied for long time. Checking what has been done in the OS field, in particular by the side of linkage between the concept of thread and task, shows there are various solutions which are aimed at improving the effectiveness of software execution when there are both parallelism and concurrency.

A first approach, which we can observe in several of the OS instances we daily use, is related to the management of critical activities within the software, such as those to be carried out by higher priority threads. This is the case of things to be done according to real-time scheduling, like it happens when exploiting the Posix-style real-time priorities [41, 72] or the Windows real-time priority class [58]. In this scenario, what really happens is that a lower priority thread needs to promptly release the CPU-core it is running on to another thread (the one marked with higher priority, *e.g.*, the real-time thread) as soon as this becomes ready again for the execution. However, the actual thing that occurs at the thread that needs to be context switched off the CPU is not the one of being actually forced to leave the CPU along its execution by some interrupt. The realease of the CPU is in fact a kind of OS-level critical activity that is carried out by the thread only under spontaneous decisions, for example when checking that its time slices have been completed or that there is a standing higher priority thread. Hence, what we have is that timer interrupts—or interrupts coming from other devices—do not lead to run a handler that brings immediately control to some other thread. The handler effect is the one of changing metadata kept at the level of the OS so that, along its

execution flow, the hit thread can (as soon as possible) check the metadata value along the execution of some kernel-level code block, such as the one that brings back control to user-space software after a previous passage to the kernel mode, in order to synchronously run the kernel-level schedule function, if needed. We note that the approach based on having the handler that does a partial part of the work—such as the change of metadata to be later checked by some thread—has also been exploited as the building block for the construction of the so called SoftIRQ architecture [52]. This architecture has been oriented to scalability in the management of the activities associated with the interrupts, a topic that is clearly important in the context of multi-core hardware (see, *e.g.*, [19]). In this solution the interrupt handler takes care of marking a so-called SoftIRQ demon as ready again for processing some specific tasks, so that it will actually process them as soon as the thread running on the CPU-core will decide to check the metadata that indicate that the demon is currently waiting for the CPU.

Our Micro-Threading approach is clearly different from this type of solutions, since a micro-thread is actually switched-off the CPU when the interrupt that indicates the need for a reschedule occurs. It can be either a timer-based interrupt or, as we will discuss towards the final part of the thesis, an interrupt coming from another processor—given that we target parallel application that can exploit multiple CPU-cores for running multiple OS threads. Hence, what we can say about the relation with OS-level CPU scheduling is that we allow something to occur—the context switch between micro-threads takes place exactly upon the interrupt arrival—which is instead not actually allowed by the OS software. The motivation that stands in our solution is that, at the OS infrastructure level, we have a clear view of whether we are trying to directly interrupt user-space code (namely a micro-thread), which is a thing that an OS observes, even though with a less directive perspective, only when running the software block that brings threads back to user-space code after their access to the kernel mode. This difference makes our approach much more fine-grain in the possibility to change the execution flow of tasks, since it leads to bring control back to the same interrupted thread, but in a different micro-thread context. In the essence, this is what the user-level thread (ULT) technology [80] is aimed at, but with no linkage to asynchronous (namely, upon the interrupt arrivals) control flow variations. In fact, under ULT, each thread can switch its execution flow (thus switching the current task) only upon the explicit arrival of the released task to an API call that yields control to a different task (*e.g.*, `longjmp`). On the other hand, we still have in our micro-thread architecture the possibility to avoid the immediate interruption, thanks to mechanisms that have linkage with the preemption counters used in classical OS solutions.

Still dealing with the correlation between threads (to be CPU dispatched) and tasks (to be processed) in a parallel hardware architecture, there has been a long term evolution of solutions aimed at avoiding that the relation between the number of concurrent threads and the number of standing tasks, becomes devastating (too many threads) for the actual effectiveness of the OS. In particular, a big work has been carried out in methods and technologies in order to avoid that the deferred work architecture, which is in the essence what we discussed before (*i.e.*, an interrupt does not lead control to a specific task directly) is in the end based on spawning threads when creating repositories of tasks to be processed at some later time. This evolution

has introduced the so called concurrency-managed approach for handling tasks—like the Linux work queues [6]—which is based on dynamically controlled thread pools whose number is not selected by the user of the deferred work architecture. Rather, it is setup by a control component that decides if new threads in the pool need to be launched, based on the overall OS state of the other threads in the same pool (say, blocked or ready for their execution). This architecture is however still based on tasks (*e.g.*, functions of the OS kernel) that, once taken control, will leave the CPU just upon their end. Overall, these solutions do not include mechanisms for immediate release of the CPU-core by some task because of the event caused by an external entity, as instead we permit in the proposed approach based on micro-threads.

As it will be clear when we will present innovative solutions for the specific application fields we target, a few works exist which aim at immediate change of the execution flow of a thread upon the arrival of an interrupt in order to support a task switch in the CPU (not a thread switch). However, in order to achieve this objective at a relatively fine grain, these solutions are based on the need to change the management of hardware timers that are suited for the OS—in particular for time-sharing. This induces an overhead which instead we avoid in our solution, which does not need to interfere with the actual time passage and tick timeline as observed by the OS software. At the same time, our solution is not prone to be unusable in scenarios where dynamic priorities and priority inversion are required [77], like in context where some currently active task has become a bottleneck for the execution of a higher priority one. In fact, our approach of keeping the CPU assigned to a given task can still be possible upon the interruption from the task external source, as we have also discussed in a previous paragraph.

Overall, beyond over stepping the state-of-the-art thanks to the exploitation of micro-threads in specific application scenarios, we believe that the micro-thread approach we present has important innovations with respect to what current infrastructure-level solutions provide. Further, it actually gives the possibility to devise the application specific innovative solutions we will discuss a head in this thesis.

Chapter 3

Micro-Threading Model

A (parallel) programming model specifies the programmer view on the parallel computer by defining how the programmer can code an algorithm. This view is influenced by the architectural design and the software environment within which a parallel application will live. Thus, there exist many different parallel programming models even for the same architecture that are distinguished by several criteria. Among these criteria we want to mention the implicit or explicit specification of parallelism, the execution mode of parallel units, the communication mode for the exchange of information and the synchronization mechanisms to organize computation. For the purposes of this thesis, we focus on those models aimed to achieve parallelism in multi-core shared-memory architectures through the creation and the assignment of one or multiple control flows to each CPU-core, and for which communication is accomplished by reading from and writing to global shared memory. These control flows are commonly referred as *threads*, whose creation and control is charged to specific libraries of the runtime system that, depending on the particular execution model they implement, may also require interaction with the OS through system-call invocations. By the way, threads can be created by user-space libraries as *user-level* threads (ULTs) or by the OS as *kernel-level* threads (KLTs). Multithreading that relies on creation of kernel threads to exploit hardware parallelism is probably the most widespread execution model, which allows multiple threads to exist within the context of one process at the level of the OS. The OS has indeed complete knowledge about the existence of these threads, hence they undergo to the scheduling decisions that the OS implements system-wide. Precisely for this reason, it is also known as *preemptive* multi-threading as a thread can be temporarily suspended by the OS without requiring its cooperation, with the intention of resuming it at a later time. Differently, user threads are implemented in user-space libraries without necessarily having the support of the OS. Involvement of the latter depends on the kind of mapping established between user and kernel threads, which is functional for the execution model that one threading library is intended to provide to programmers.

One-to-one (1:1) mapping is probably the most intuitive model (see Figure 3.1) as it implies for each user thread the generation of its counterpart at kernel-side. Scheduling activities and assignment of threads to CPU-cores are charged to the OS that also reserves space for accommodating their control blocks—thread control

block (TCB) is a data structure maintained by the OS which contains specific information needed to manage one thread—and CPU snapshots—it is the set of all general purpose registers and control flow registers including program counter. A typical example of (1:1) mapping is the Posix Threads library for Unix-like systems, which provides the programmers with facilities to assist thread creation and control by mean of calls to Posix Threads API. If on the one hand this kind of mapping immediately gives the advantage of exploiting the hardware parallelism offered by multi-core processors and multi-processor computers, on the other, threads creation is an expensive operation as well as their context-switch too. Moreover, applications have no direct control over the scheduling of threads, reason for which they lack any capability to lent computational power to target threads that more deserve it when changes in the program state require changes in the current schedule.

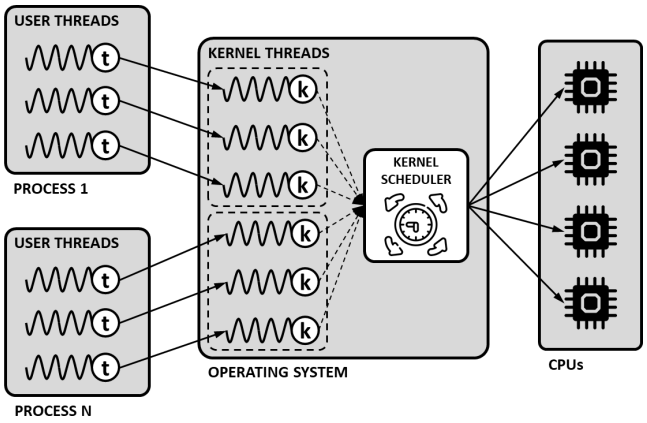

Figure 3.1. (1:1) thread mapping model.

Many-to-one (M:1) mapping is a model for which multiple user threads map to the same kernel thread (see Figure 3.2). Implementation of this model does not require any interaction with the OS as all the application-level control flows are handled by the threading library. The library itself is charged of the creation of user threads and to manage CPU-contexts for each of them. It also implements the scheduling logic that determines the order and the way according to which user threads get executed. In this regard, these scheduling activities are normally performed only upon completion of thread executions, unless threads spontaneously yield control either periodically or as the result of checks which decree a context-switch must occur. The latter takes place in the form of a *cooperative* scheduling and requires the programmers to implement apposite statements to accomplish this behaviour. Some definitions also refer to these threads as *fibers* due to the particular execution model they implement, which is definitely non-transparent. This model clearly provides the advantage of having fast and lightweight switches of user threads. Moreover, it can be implemented on simple OSs that do not support kernel level threads living in a same process. On the contrary, it misses the opportunity to exploit the parallelism offered by the hardware at least for user threads that map to the same kernel thread.

A hybrid solution is then provided with many-to-many (M:N) mapping (see Figure 3.3), which is a compromise between the (1:1) and (M:1) threading models. Systems employing this model are generally more complex to implement, in that

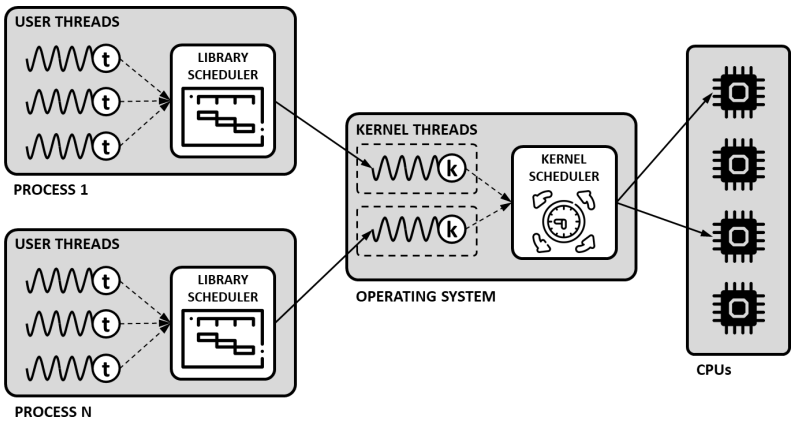

Figure 3.2. (M:1) thread mapping model.

threading libraries are responsible for the scheduling of user threads on the available schedulable entities handled at kernel-side. This requires the scheduling logic to be supported with some form of synchronization in that all these schedulable entities are admitted to take care of the work associated with any user thread. However, the scheduling of user threads is still limited to fixed points in the execution of kernel threads, as it is for the (M:1) model, which can take place at most in a cooperative form as long as the programmers are willing to spend additional effort for achieving such behaviour. Nevertheless, even with the aid of programmers, there exist application scenarios for which program state changes are not predictable at development time and can occur so quickly that any attempt to tackle them through the activation of the scheduling logic by mean of predetermined calls (to user level threads' switch functionalities) inserted into the code at the application development time is not adequate.

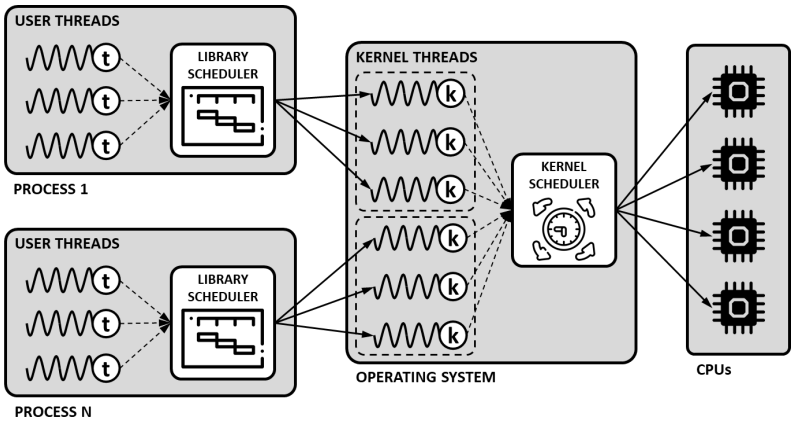

Figure 3.3. (M:N) thread mapping model.

The micro-threading model is an innovative parallel execution model that we have designed to address the abovementioned shortcomings. It is based on a ULT technology that implements the (M:N) mapping of user threads to kernel threads. However, it differs from similar execution models in that it overcomes the limits of

a cooperative form of scheduling of user threads, which currently characterizes all the ULT solutions offered to the application programmers. Differently, the micro-threading model relies on a new technology that we have appositely designed and implemented to transparently support the execution of user threads with a preemptive form of scheduling. We called this technology *user-level micro-thread* (ULMT), which is composed of user-space facilities implemented at the level of runtime environments hosting the specific application, plus a kernel module installed at the level of the OS—particularly, the Linux OS. Working in synergy, these components make it possible to put in place synchronous (*i.e.*, embedded in the compile/link-time defined CFG) and asynchronous (*i.e.*, not embedded in the compile/link-time defined CFG) activation of the scheduler of user threads.

Specifically, the ULMT technology allows user threads to asynchronously (and promptly) suspend their execution upon the occurrence of events generated by external sources, like application-specific interrupts standing aside of the ones the OS exploits for implementing its general purpose housekeeping operations. How these interrupts can be instantiated on off-the-shelf computing systems, according to different solutions, will be thoroughly discussed. In any case, when these interruptions arise, the control is transparently given to the function that implements the application-specific scheduling logic of user threads. Clearly, different application contexts may have different needs in terms of implementation of the scheduling logic. This is not an impairment for ULMT given the modular construction of the overall architecture supporting this paradigm.

Since this mode of switching user threads (which can arise due to causes that are unrelated from the application semantics) gives rise to a preemptive form of scheduling, we started to refer to these threads as *micro-threads* in that they can always be made pausing at arbitrary points of their execution in a safe manner, that is, without incurring the risk of compromising their execution state and with the aim of resuming them at a later time either along the execution path of the same kernel thread that was carrying on the activities of the involved micro-thread or along the execution path of another one. This solution is somewhat reminiscent of the way through which the OS performs the scheduling of kernel threads, which can take place upon the expiration of the LAPIC timer of CPU-cores, but with the difference that the aforementioned interruptions will lead to the activation of the user-space scheduling logic that can be either provided by the micro-threading library or implemented by the programmer whereas it is desired to employ custom scheduling policies. We want to point out that this scheduling logic, commonly implemented in the form of a function, is equivalent to code provided for classical ULT-based applications that gets activated after every user thread completes its execution or upon direct invocation by mean of calls included by the programmer. The ULMT technology also extends the activation of this scheduling logic to points of the execution where it was not coded to occur, therefore independently of what the application CFG admits. This is the core advantage the micro-threading model provides to drive parallel applications and to enable them to follow optimized execution dynamics, as it makes them capable to catch sudden program state changes and to very timely react with renewed decisions about the assignment of micro-threads (hence real application activities) to kernel threads. In fact, execution of the scheduler logic is no longer confined to a few predefined nodes composing the

CFG of the application, and to the actual timings or reaching these nodes along an OS thread. By the way, a micro-thread can be split into several sub-parts that form together its own execution trace, whose size and their number will only be known at run-time depending on the timing according to which such interruptions will occur—the micro-threading model does not impose any particular constraint on the execution of these sub-parts but they must remain sequentially consistent with each other even when dispatched on different kernel threads, which is a requirement that the ULMT technology fulfils. It is therefore clear that the micro-threading model, more precisely the ULMT technology, provides a solution to the limitations that we have listed at the end of the introductory chapter.

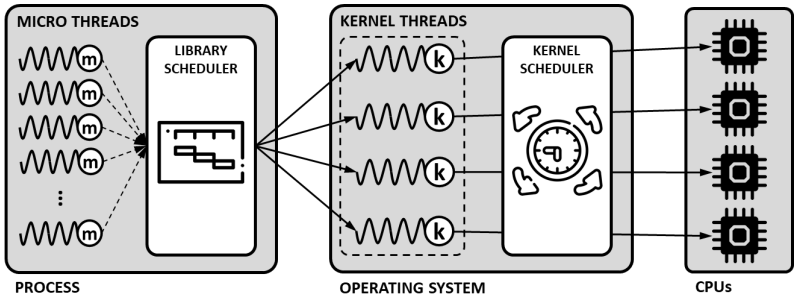

Figure 3.4. Micro-thread mapping model.

Moreover, in order to mitigate the likelihood of incurring sub-optimal scheduling conditions, which can always arise in absence of extensive and expensive coordination between the user-space scheduler (which handles micro-threads) and the OS scheduler, a number of kernel threads that does not overcome the number of CPU-cores can be taken as a good reference when exploiting the micro-threading model, especially in dedicated environments hosting HPC and time-sensitive applications. This way, the probability of having a kernel thread switched off the CPU while it was performing a critical activity on behalf of a micro-thread is definitively reduced. As an additional aspect, even if it is not indispensable for correctness, kernel threads can be pinned to different CPU-cores, as it is shown in Figure 3.4, which is an approach that makes the micro-threading model adhering to the paradigm of strong separation of the concepts of *work* and *workers*. The latter are just the schedulable entities handled at kernel-side that act as engines for the execution of the former and which can be seen as a sort of intermediate layer between the work and a particular CPU-core. In this regard, dispatching a micro-thread to a kernel thread will correspond to assigning the corresponding work to a processing unit. The latter is an operation over which the program can have complete control. Just for this reason, the micro-threading model is probably the execution model that best applies to the needs of those parallel applications designed according to a scheme where worker threads are not devoted to the execution of a single activity (*e.g.*, task parallelism). Indeed, a certain task to be performed can be associated with the execution context of a specific micro-thread which will represent the work that one or more kernel threads will have assigned along time in the most flexible way (and the most suitable for handling preemptible tasks) in order to progressively give rise to newer schedules, those that are deemed to lead the application evolve according to better execution dynamics.

3.1 Runtime Facilities

Since the switch of a micro-thread with another may always arise asynchronously at arbitrary points in the execution of a kernel thread, the entire execution context of a micro-thread must be preserved in order to correctly resume it at a later time. In this regard, when a micro-thread is created, the micro-threading library also reserves enough space to maintain the values of all CPU registers which form its execution context. It comprises all the general purpose registers plus floating-point and SIMD registers since there in no guarantee that the content of any one of them is preserved across subsequent micro-thread switches—in fact the switch does not correspond to any control flow variation governed by an Application Binary Interface (ABI)—nor it is possible to determine that the resumption of a micro-thread will occur again along the execution path of the same kernel thread. Conversely, resuming a micro-thread requires that the whole set of previously flushed registers is restored again into one of the available CPU-cores (or hyperthreads), hence accomplishing the correct restart of its execution. Figure 3.5 shows an example of saving/restoring the values of all CPU registers to/from main memory on a `x86_64` processor. The rightmost box reported in this figure is the data structure that the ULMT runtime uses to maintain the values of CPU registers when the micro-thread execution undergoes suspension, one for each micro-thread, and that we have called CPU-snapshot. Additionally, in order to ensure all micro-threads correctly perform their operations (they do not deviate from the expected behaviour), each micro-thread must also execute by relying on a different stack memory area so as to avoid that, in presence of interleaved executions of distinct micro-threads, the effects that some operations performed by any one of these can have into the stack do not invalidate those produced by the others.

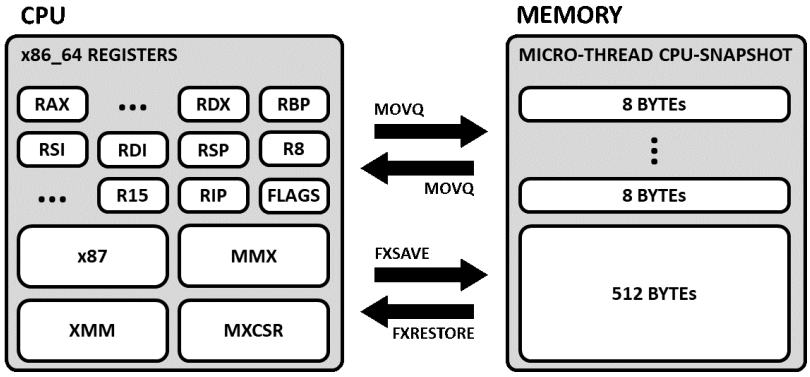

Figure 3.5. Saving of the execution state of a micro-thread into the relative CPU-snapshot data structure.

Thus, the micro-threading library has the responsibility of allocating memory for both the stack area and the CPU-snapshot data structure for each micro-thread that will be spawned at run-time. These last two objects together form what we call the micro-thread context, which is essentially the set of all information that enables a micro-thread to correctly start running, possibly suspending and then resuming, along the execution path of any kernel thread. By the way, the micro-threading library is also in charge of properly initializing the micro-thread context, which

is a necessary operation in order to allow a micro-thread to start performing the assigned task. In this regard, there exist several solutions to make the setup of fresh execution contexts, among which we mention the one proposed in [17] which is aimed to provide a portable approach for creating contexts of multiple threads of execution on Unix systems by exploiting a signal stack trick based on standard facilities and ANSI-C language features. This is clearly an approach that would make the ULMT technology portable to a number of CPU architectures. Nevertheless, it is not a performing solution when making dynamic setup of contexts at run-time as the completion time required for the initialization of each context is bounded by the time required to send, to deliver and to process a signal onto the target system. Thus, for the achievement of the results presented in this thesis, we implemented a solution compatible with those systems that follow the System V ABI specification for compiled application programs on x86_64 architectures. However, if needed, the proposed solution can be extended to different architectures with reduced effort.

Algorithm 1 Creation of a micro-thread context.

```
 1: procedure CONTEXT_CREATE(OriSnap, MtSnap, MtStackPtr, TaskFunc, TaskArgs):
 2:     if CONTEXT_SAVE(MtSnap) == 0 then                ▷ first time MtSnap is saved
 3:         FrameSize ← (MtSnap.BP − MtSnap.SP)
 4:         COPY(MtStackPtr − FrameSize, MtSnap.SP, FrameSize)
 5:         MtSnap.BP ← MtStackPtr
 6:         MtSnap.SP ← MtStackPtr − FrameSize
 7:         return
 8:     else                                             ▷ when MtSnap is restored
 9:         TaskFunc(TaskArgs)
10:         CONTEXT_RESTORE(OriSnap)                     ▷ no-return statement
```

In our solution, the initialization of each context is accomplished by generating a fresh stack frame at the bottom of the space reserved for the stack, in such a way that the micro-thread operates on that frame as if it were activated via a normal function call. The latter operation takes place along with the initialization of the CPU-snapshot data structure with values that match those that would have characterized the micro-thread execution state after the activation of that frame via function call. The final result is an initial configuration for the micro-thread context which is consistent with the aforementioned ABI specification followed by the compiler. Although this initial frame has nothing to do with the operations that the involved task is intended to perform, it is linked to the execution of an intermediate code block that will finally invoke the task function by passing the related arguments. This also implies the automatic creation of a new stack frame on top of the initial one as soon as the micro-thread begins to perform the associated task. Moreover, once a micro-thread has finished the execution of its task, the code residing into the aforementioned intermediate block makes use of the information stored into the lowest (artificial) stack frame to resume the execution context that has previously let control to the current one (this is usually a platform context within which the scheduling operations and some others included into the runtime are performed). Algorithm 1 shows the procedure we have discussed so far which is, however, a pseudo-code version of the real implementation. The actual implementation is indeed a function coded in x86 assembly language because we must avoid that any form of

compiled code may alter the shape or the content of current stack frame just prior to complete its copy (operation at line 4) into the space reserved for the micro-thread stack. Once the original stack has been copied completely, the previously saved CPU snapshot can be updated by overwriting the fields intended to accommodate the base pointer and the stack pointer (at line 5 and 6) in such a way that, when this micro-thread will later resume, it will restart execution using its own stack. We want to make notice that, for the same reasons discussed above, the procedures CONTEXT_SAVE and CONTEXT_RESTORE (invoked respectively at line 2 and 10) are functions coded in x86 assembly language. More in detail, the CONTEXT_SAVE function has to perform a complete flush of CPU registers into the CPU snapshot data structure by preserving the values that these registers maintained before this procedure was called, that is, the execution state of a micro-thread before and after a call to CONTEXT_SAVE must be the same (with the exception of the content of rax register that, still according to the ABI specification, is a caller-save register used to return a value to the caller function). This last requirement is clearly needed for the correct resume of a micro-thread in that, as soon as a call to CONTEXT_RESTORE function completes restoring the involved CPU-snapshot, its execution resumes as if it had never been interrupted before. In this regard, CONTEXT_SAVE function recovers the return address value to insert into the CPU snapshot data structure from the bottom of current stack frame, where it has been automatically stored when this function was called by means of call instruction (in Listing 3.1 is reported a portion of the assembly code included in CONTEXT_SAVE that is used to obtain the return address).

Listing 3.1. Saving SP and IP into CPU snapshot (CONTEXT_SAVE).

```
. . .                          // rax is address of CPU snapshot
movq    %rsp,    32(%rax)      // move stack pointer into CPU snapshot
movq    8(%rsp), %r11          // 8(%rsp) is the return address
movq    %r11,    128(%rax)    // move return address into CPU snapshot
. . .
```

As we already pointed out, the only thing that differs when resuming a suspended micro-thread context, with respect to when CONTEXT_SAVE was executed the first time, is the value returned to the caller function. This value is moved into the rax register just prior to returning from the CONTEXT_SAVE and CONTEXT_RESTORE functions, and it is used to determine if the involved CPU-snapshot has been just saved for later resume or it is currently restoring as a result of a context-switch (we use value 0 for the first case, and any value different from 0 for the second one).

Listing 3.2. Restoring SP and IP from CPU snapshot (CONTEXT_RESTORE).

```
. . .                          // rax is address of CPU snapshot
movq    32(%rax),  %rsp        // restore the old stack pointer
movq    128(%rax), %r11        // 128(%rax) is the old return address
movq    %r11,      8(%rsp)     // restore the old return address
. . .
```

Furthermore, since the return address stored at the botton of the stack frame created upon the activation of the CONTEXT_SAVE function is consumed by the ret instruction when the execution returns control to the caller one (the value pointed

by the stack pointer register, which is the operand of ret instruction, is moved into the instruction pointer register while the stack pointer value is incremented accordingly), the CONTEXT_RESTORE function must also rearm the stack frame being restored with the correct return address value (in Listing 3.2 is reported a portion of the assembly code included in CONTEXT_RESTORE that is used to rewrite the return address into the stack frame being restored) in order to recreate the same state that characterized the execution when the interested context was saved. Just to clarify, the return address value is maintained within the CPU-snapshot data structure by the field reserved for the instruction pointer, which is then moved by the CONTEXT_RESTORE function to the apposite position into the stack frame before the execution of ret instruction. This operation of moving the return address value into the stack is probably the most significant one among those that characterize the ULMT technology as it provides a double final effect, that of recreating an execution state which is an exact copy of the one that was previously saved by CONTEXT_SAVE, and that of making possible a jump to the correct address, which completes the context-switch procedure started by CONTEXT_RESTORE.

This way of restarting the execution from a generic address results very powerful because it does not require to engage any register to maintain the return address value at the time when all CPU registers have been already filled with the content of the CPU-snapshot data structure, nor it requires to rely on immediate operands which are clearly not known at compilation time. As we will see in the following, the latter also turns out as the key capability to resume the execution of micro-threads that have been subject to asynchronous preemption.

3.2 Kernel Module

As already mentioned at the beginning of this chapter, the other software component that is part of the ULMT technology is a Linux kernel module appositely devised to complete the commissioning of the micro-threading model. The main purpose of this module is to handle the reception of specific interrupts with the final goal of splitting the execution flow of micro-threads so as to allow the kernel threads to slide out from the paths to which these flows were constrained in order to give control to a distinct procedure, the one that eventually leads to abandoning the current micro-thread context.

These interruptions can have different origins as the possibility to trigger them derives from the exploitation of several hardware capabilities of modern CPUs. For example, x86_64 processors are equipped with model specific registers (MSR) that can be programmed to generate interrupts after a certain number of instructions are fetched from cache, or after a certain number of operations (an instruction may decode to more than one operation) are dispatched for execution, or again, after a number of core clock cycles have elapsed. Although these registers have been designed to perform specific activities, such as program execution tracing and computer performance monitoring, they normally remain unused when running applications in production and nothing prohibits their use for other purposes.

In addition to these registers, all modern processors are also equipped with the advanced programmable interrupt controller (APIC) which is composed of as many

hardware components called Local-APIC (LAPIC) as there are CPU-cores. Each LAPIC manages all external and internal interrupts locally to a single CPU-core and it is also provided with apposite registers that allow to trigger inter-processor interrupts (IPI) to remote CPU-cores of SMP systems. The latter are delivered to target CPU-cores with almost zero-delay as soon as a source CPU-core writes on these registers. By the way, they are already used by the OS kernel to force all CPU-cores in performing the same action system-wide (*e.g.*, flushing of TLB caches when memory mappings are changed). However, they are free to be programmed by any kernel thread that wants a certain activity (*i.e.*, handler of a registered interrupt vector) to be performed on a remote CPU-core or multiple ones.

Anyhow, all the abovementioned registers can be accessed only when threads are executing in kernel mode, that is, user-space code runs with a privilege level that does not permit to perform such kind of operations. This is why ULMT technology mandatory requires the cooperation of kernel-space code to put in place the micro-threading model. In this regard, once the kernel module has been installed it provides a number of services needed to support the execution of micro-threads for those applications that are intended to follow the micro-threading model. These include a number of file-operations for a special device-file appositely created and mounted in kernel-space to keep track of registered threads, that is, those kernel threads that want to exploit the aforementioned hardware capabilities to put in place the execution model of interest. Only registered threads are allowed to receive this kind of support for accomplishing to the preemptive execution of micro-threads, with the goal of giving the application the capability to timely react to program state changes and to respond with renewed assignments of micro-threads to kernel threads as soon as these changes materialize. Moreover, in order not to waste the occurrence of interrupts whenever a registered thread has been scheduled off the CPU by the OS, we rely on a hook installed at the tail of the OS schedule function via the Linux kprobe service. This hook to the OS scheduler simply checks if a per-CPU flag has been set to indicate this scenario has occurred, and in positive case it gives control to the same function devised to handle this kind of interrupts.

The interrupt handler implements the most important service that this kernel module provides. It is designed to alter the control flow of kernel threads that are carrying out micro-threads to start executing something outside their current execution traces which ultimately leads to revaluating the assignment of work. We refer to this operation as *control flow variation* (CFV) which involves updating the content of some CPU registers of the interrupted micro-thread in a way that, when the execution will switch back again in user mode, the micro-thread restarts running from the beginning of another function as if it had voluntarily called that procedure. In this regard, the handler first checks if the thread that was subject to interrupt is among those registered to undergo CFV, then it performs CFV for the involved micro-thread. In Figure 3.6 is reported a graphical representation of the steps that, once performed in interrupt context yet, lead the execution of micro-threads to slide out from their CFGs so as to perform the operations included in a different trace. They mainly consist of a copy of the instruction pointer and stack pointer register values at the bottom of a small memory area that acts as an intermediate stack, which is followed by the updating of the same registers in order to point, respectively, to the function to which it is wanted to give control

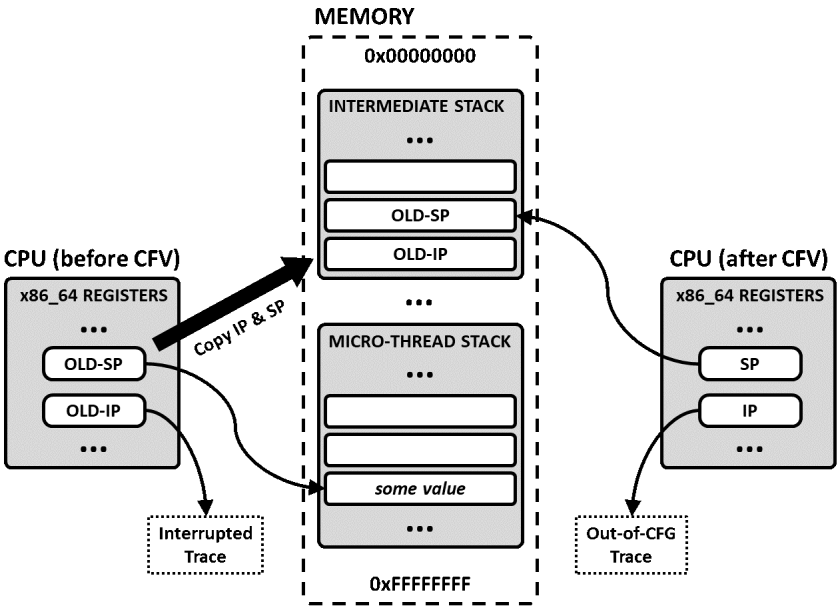

Figure 3.6. Variation of the control flow.

downstream of CFV, and at the top of the intermediate stack where their old values were previously placed. The use of the intermediate stack in place of the micro-thread one is needed not to incur the risk of accessing a virtual page whose frame is not present in main memory (never accessed before or the associated frame swapped out from memory) which, consequently, would give rise to a page-fault whereas the kernel thread is running in interrupt context. The latter is indeed an unwanted behaviour as the involved boundary conditions are not those expected when handling this kind of fault. Differently, the intermediate stack is a single memory-mapped virtual page whose frame has been locked in main memory, one for each registered kernel thread, for which we can be sure no page-faults will occur while performing CFV. As soon as the execution switches back in user mode we can stop worrying about page-faults, as it is an exception that normally occurs during the execution of user-space applications, and the micro-thread stack can be safely restored as shown in Figure 3.7. This is accomplished by copying first the old instruction pointer on top of the micro-thread stack memory area (whose address is known since it was stored within the intermediate stack) and by updating then the stack pointer register to point to the memory address where the old instruction pointer was just placed. The latter is the address of the instruction immediately following the last committed one before the execution was interrupted, and it will be used later to restore the abandoned control flow (*i.e.*, performing a `ret` instruction on it).

However, the same procedure leading to a variation of the control flow also leaves the micro-thread execution in a state for which the compiled code gives no guarantees regarding the content of CPU registers since no `call` instruction was actually included in the execution trace at compilation time, hence the code does not follow the normal calling conventions designed to preserve the execution state across subsequent function calls. To cope with this problem, the function to which

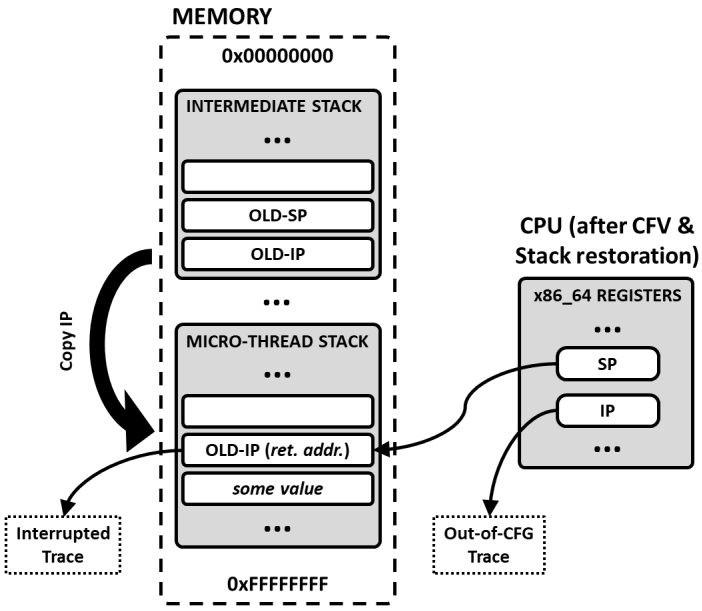

Figure 3.7. Restoring micro-thread stack.

control is given downstream of CFV is a function coded in x86 assembly language that we have called CFV_TRAMPOLINE (the pseudo-code is reported in Algorithm 2). This function indeed, after having restored the original stack and after having made some checks about the preemptibility of the involved micro-thread (at line 2 and 4), immediately invokes the CONTEXT_SAVE function (at line 6) to save the micro-thread context, so that it becomes possible to finally start the execution of the scheduler function (at line 7) without affecting the consistency of the interrupted execution state. The scheduler is in charge of evaluating if the assignment of a different micro-thread to the current kernel thread would allow the application to follow better execution dynamics with positive effects on performance. If so, the scheduler is also in charge of resuming the execution context of the selected micro-thread by means of a call to CONTEXT_RESTORE function. The currently suspended micro-thread can then be resumed either in the immediate or at a later time by invoking the same function, which will lead the involved execution to perform the ret instruction that finally restores the original control flow as described before (operation at line 8).

Algorithm 2 CFV trampoline.

```
1: procedure CFV_TRAMPOLINE:
2:     SWITCH_STACK()                          ▷ reported in Figure 3.7
3:     pc ← READ_PREEMPTION_COUNTER()
4:     if pc == 0 then                         ▷ see Section 3.3
5:         MtSnap ← GET_CPU_SNAPSHOT()
6:         if CONTEXT_SAVE(MtSnap) == 0 then   ▷ when MtSnap is saved
7:             MT_SCHEDULER()
8:     return                                  ▷ when MtSnap is restored
```

This is precisely the key capability we highlighted at the end of the previous

section. When the ret instruction is being executed, all the CPU registers except the instruction pointer already maintain their original values, those that characterized the execution state at the moment before the interrupt was delivered. Once the ret instruction has committed, also the instruction pointer register contains the correct value while the stack pointer one is updated accordingly to restore the original stack frame. This way we have resumed the execution of a micro-thread that has undergone CFV asynchronously without having compromised its execution state. This also means that it is always possible to perform CFV at arbitrary points of the execution of micro-threads, but it does not mean that any micro-thread can always be preempted in favour of another regardless from what it is currently performing. As we will see in Section 3.3, there could be conditions under which preemptibility of micro-threads is voluntarily inhibited by the micro-threading runtime before starting the execution of critical code regions, that is, the scheduler of micro-threads is no longer invoked as long as these conditions are met (at line 4 of Algorithm 2).

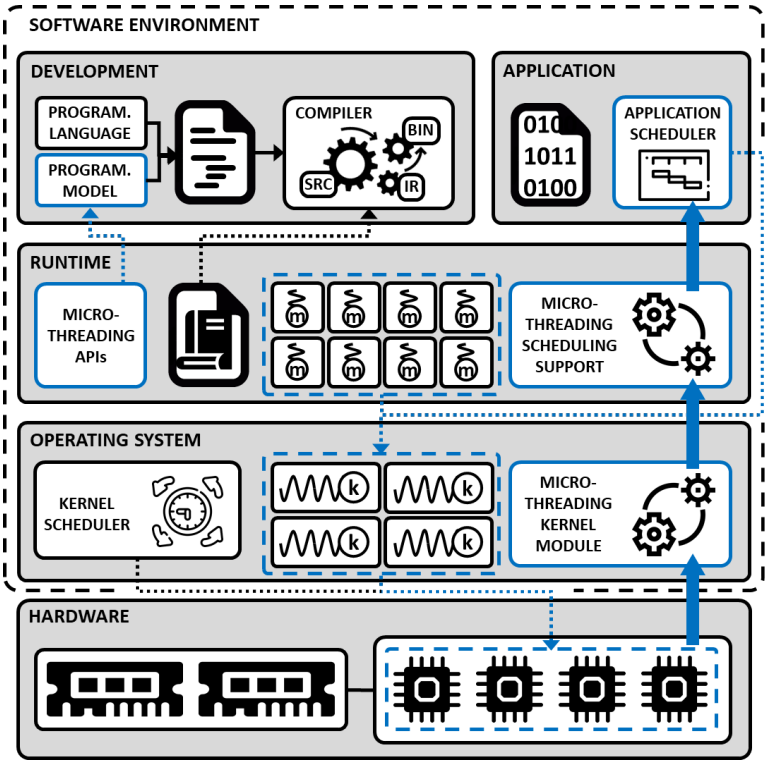

Figure 3.8. ULMT technology within the software environment.

For the sake of completeness, in Figure 3.8 is shown the software environment within which ULMT-based application actually live. It is basically the same figure shown in the introductory chapter with the difference that highlighted in blue are the software parts that together form the ULMT technology. The runtime system is enriched with facilities, as those presented in Section 3.1, to support the execution model of micro-threads. The latter are provided by the micro-threading library, which is linked to the application whenever the application itself is coded by following the proper parallel programming model. A kernel module is installed at the OS level to handle specific interrupts whose handler has been designed to perform CFV

of threads instantiated for executing tasks of ULMT-based applications. Such an altered control flow gives to threads the capability to execute code not included in the interrupted execution trace without losing the possibility to resume the original ones at a later time. In the end, this procedure leads to the activation of the scheduler of micro-threads which eventually renews the task assignment to the current worker thread. It is clear therefore that all these software parts cooperate with each other to transparently accomplish the preemptive execution of micro-threads, which is definitively the only approach that can guarantee time bounds for reacting to sudden program state changes with renewed schedules.

3.3 Micro-Threads Preemptibility

Till now we assumed the whole execution of a micro-thread to be always preemptible regardless of what it is actually performing. However, there are code regions for which it would be preferable not to defer the involved execution but leave it continue up to the end of these regions. In some cases, this choice is unavoidable in order not to compromise the correctness of the micro-thread execution, also considering that in the end this execution corresponds to a part of the execution of a classical kernel thread. Kernel threads have therefore these tasks tied to them as long as the execution of these regions does not complete. The latter include all non-reentrant functions which, by definition, should not be suspended as their interleaved execution by distinct micro-threads may lead to corrupt the execution state of these functions. The programmer who wants to rely on the micro-threading model to put in place parallelism is generally aware of this fact and he has the responsibility of implementing code which is consistent with the chosen parallel programming model. Nevertheless, it is common that programs make frequent use of external library functions to accomplish the completion of a number of tasks, including memory management and I/O activities, and for which there are no guarantees about reentrancy of their implementation. Also, there are micro-threads that need to obtain the exclusive access to one or more critical sections to complete a set of operations atomically. This is achieved by acquiring locks or mutexes that will be released only when these operations have terminated. Therefore, in order not to incur the risk of generating deadlocks, as well as to introduce excessive delays for completion of a critical section, we need to prevent the micro-threads from being scheduled-off while they are performing these sections. It is clear that, both these last two aspects deserve to be tackled with adequate solutions which do not require the intervention of the programmer, rather they must be transparently included into the execution trace thanks to compilation/linking time operations.

We addressed both the abovementioned scenarios by using per-thread atomic counters that increment by one each time a non-preemptible code region is encountered during execution of a micro-thread, and which will decrement accordingly when exiting these regions. The value of these counters, that we have called `preemption_counters`, are updated by means of operations whose execution does not take place while performing activities specific to the micro-threading library, rather they are injected into the application code at compilation time or belong to intermediate blocks placed between the code that calls the library function and its actual

Algorithm 3 Wrapping for resource acquisition and release procedures.

```
1: procedure WRAP_RESOURCE_ACQUISITION_FUNC:
2:     ATOMIC_INCREMENT()
3:     ret ← RESOURCE_ACQUISITION_FUNC()
4:     if ret == FAILURE then
5:         ATOMIC_DECREMENT()
6:     return ret
```

```
1: procedure WRAP_RESOURCE_RELEASE_FUNC:
2:     ret ← RESOURCE_RELEASE_FUNC()
3:     if ret == SUCCESS then
4:         ATOMIC_DECREMENT()
5:     return ret
```

implementation. Thus, their execution is fully part of the ULMT technology as they contribute to the correct commissioning of the micro-threading model. To this end, we rely on a wrapping strategy applied at compilation time that allows to execute these operations exactly where they are required to take place. It is carried out fully transparently by the compile/link infrastructure, with no need for any intervention by the programmer. With the aid of wrappers surrounding calls to library functions or to lock acquisition/release facilities it is possible to transparently update the value of these counters when entering and exiting non-preemptible code. Algorithms 4 and 3 show the pseudo-code of wrappers used to protect the micro-threads from preemption when they are executing non-preemptible code regions.

Algorithm 4 Wrapping for library function calls.

```
1: procedure WRAP_NON_REENTRANT_LIBRARY_FUNC:
2:     ATOMIC_INCREMENT()
3:     NON_REENTRANT_LIBRARY_FUNC()
4:     ATOMIC_DECREMENT()
```

However, since the interrupts arise at arbitrary points in the execution regardless from what micro-threads are executing, to avoid that the occurrence of CFV leads to abandon the current execution context whenever the preemption_counter value is not zero, an additional check must be performed by the CFV_TRAMPOLINE function before giving control to the scheduler of micro-threads (operation at line 4 of Algorithm 2). If this check succeeds, it means the micro-thread is performing no critical activities and the involved kernel thread can have eventually assigned a different task (*i.e.*, micro-thread). Otherwise, the check has failed and the original control flow is immediately restored to continue the execution of the interrupted trace.

Chapter 4

Time-based Micro-Thread Scheduling

In Chapter 3 we mentioned the existence of model specific registers (MSR) in `x86_64` processor architectures allowing to generate interrupts upon the occurrence of specific events. More in detail, we are referring to the *instruction-based sampling* (IBS) [1] and the *processor event-based sampling* (PEBS) [39] hardware facilities offered, respectively, by AMD and Intel processor architectures. Both these facilities can be enabled in their relative architectures to gather specific metrics related to processor instruction execution to support computer performance monitoring and program execution tracing activities, for which data capture is performed by hardware at a sampling interval specified by values programmed in the MSRs. As soon as the programmed interval has expired and the data collection is completed as well, the hardware signals an interrupt to the involved CPU-core which in turn leads to the activation of a specific handler installed in the OS kernel. Although these hardware facilities were originally designed for supporting the abovementioned activities, the capability they can potentially provide to threads to immediately switch to kernel mode regardless of what CPU-cores were actually executing and by preserving the state of the interrupted execution flow—it is result of a finite number of operations performed cooperatively by the firmware and the software at the OS level—can ideally be exploited to give control to an interrupt handler devised to serve other purposes such as that of performing CFV of micro-threads. The latter becomes possible when we replace the default handler with the one provided by the kernel module making part of the ULMT technology (Section 3.2), in such a way to let it be the function that gets activated upon the occurrence of such kind of interrupts. Since the arrival of these interrupts depends on the specified sampling interval, we refer to this approach of supporting preemptive execution of micro-threads as the *time-based* approach, in that the activation of the scheduler of micro-threads, which will eventually lead to renew the assignment of a micro-thread to the interrupted kernel thread, can potentially take place at the end of each interval.

The results presented in this chapter are the outcome of experiments carried out on AMD processors. For this reason, from now on, we will focus on details of the IBS implementation. Nevertheless, all the considerations made in the following would still be valid even if we had employed the PEBS implementation of Intel processors

as it is only a means for actually realizing time-based micro-threads scheduling.

We already mentioned that the IBS facility can be used by software to perform code profiling based on statistical sampling. It comprises two independent data gathering components which provide respectively information about instruction address translation look-aside buffer plus cache behaviour for randomly selected cache blocks (fetch sampling), and information about instruction execution behaviour by tracking the execution of a single micro-operation that is randomly selected (operation sampling). When the programmed interval for fetch or operation sampling has expired, and data collection for tagged fetch block or dispatched operation is completed (data is placed in specific IBS registers), the hardware triggers an interrupt on the involved CPU-core. For the purposes of our work, we considered only the IBS facility that enables to receive interrupts upon the expiration of operation sampling intervals as it can be programmed to count core clock cycles in place of dispatched micro-operations, which is a feature that allows to specify the time interval as a function of the core speed. In Figure 4.1 we show the IBS execution control (IbsOpCtl) register used to enable (when the 17-th bit—IbsOpEn field—is set to one) and to configure the operation sampling functionality of the IBS facility. It is a 64-bits register of which 27 bits are devoted to represent the current value of the operation sampling counter (IbsOpCurCnt field), which is a value incremented by one at each clock cycle (when the 19-th bit—IbsOpCntCtl field—is set to zero) until it reaches the value specified within other 23 bits of the same register that represent the most significant bits of the operation sampling maximum count value (IbsOpMaxCnt field). As soon as the counter value equals the maximum count value, the hardware signals an interrupt to the involved CPU-core. Also, it sets to one the 18-th bit of the IBS execution control register (IbsOpVal field), which is the bit devoted to represent the operation sample validity. Once the operation sample gets valid and ready to be retrieved, the sampling counter stops counting and no other interrupts will be delivered until the interrupt handler resets again this bit.

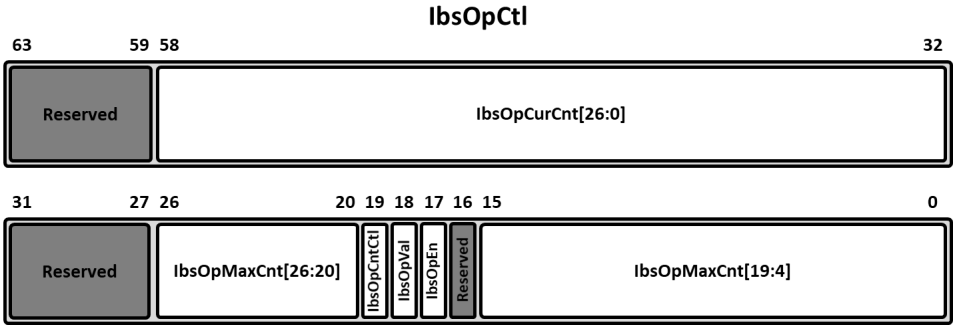

Figure 4.1. IBS execution control register.

The IBS execution control register can be read and written by means of the x86 instructions rdmsr and wrmsr if and only if current execution has the necessary privilege level to do these operations. This means that threads can access this register only when they are running in kernel mode. Thus, the programming of the IBS execution control register can only be done by software included in the kernel module which, in addition to implementing the interrupt handler, also exposes a

number of file-operations (*e.g.*, ioctl) for a special device-file that is used to keep track of threads registered to undergo CFV. All these software components are involved in programming this register at different points of program execution. More in detail:

- when a kernel thread asks to be registered in the device-file, the latter enables the IBS execution control register to start counting clock cycles on current CPU-core;
- when a kernel thread wants to be removed from the set of those registered in the device-file, the latter disables the IBS execution control register to stop counting clock cycles on the current CPU-core;
- upon the reception of the IBS interrupt, if the involved thread is registered in the device-file then the interrupt handler resets the operation sampling validity bit in the IBS execution control register to restart counting the core clock cycles of the next sampling interval, otherwise it sets a per-CPU flag that will be read by the hook installed at the tail of the OS schedule function;
- when the OS schedule function gets executed, the installed hook does first a check to verify whether the scheduled thread is registered in the device-file, then it controls if the aforementioned per-CPU flag has been set by the interrupt handler. If both checks succeed the control is immediately given to the interrupt handler that works as described in the previous point.

In Listing 4.1 we report the function used to reprogram the IBS execution control register. It takes a single argument called clocks which is used to specify the operation sampling maximum count value (IbsOpMaxCnt field), while all the bits dedicated to the operation sampling counter (IbsOpCurCnt field) are set to zero. The operation sampling bit (IbsOpEn field) is set again to value one so as to maintain this functionality enabled. The operation sample validity bit (IbsOpVal field) is set to zero in order to immediately restart counting core clock cycles.

Listing 4.1. Programming of the IBS Execution Control register.

```
static void enable_ibs_op(u32 clocks) {
    // address of IbsOpCtl on AMD Family 10h Processors
    static const u32 msr = 0xC0011033U;
    // 32 MSb of IbsOpCtl
    u32 high = 0x0U;
    // 32 LSb of IbsOpCtl
    u32 low = (clocks & 0xFFFF0U) >> 4;  // IbsOpMaxCnt[19:4]
    low |= clocks & 0x7F00000U;          // IbsOpMaxCnt[26:20]
    low |= 0x1UL << 17;                  // IbsOpEn
    // writes High and Low on MSR register
    asm volatile(
        "wrmsr" : : "c" (msr), "a"(low), "d" (high) : "memory"
    );
}
```

The argument clocks of the function enable_ibs_op indicates the number of core clock cycles that must pass before the next IBS interrupt will be triggered. It can

be either a static value or a function of the core speed so as to match a desired time interval. Furthermore, it is not constrained to assume the same value throughout the execution, but it can be a value adaptively updated depending on the effectiveness of the time-based micro-threads scheduling. The latter feature has not yet been included in ULMT technology, however it can be easily implemented by relying on sporadic and low overhead user- to kernel-space communication.

At this point it is clear how the IBS facility can be used to periodically generate interrupts at the involved CPU-core. We also discussed which are the software components belonging to the kernel module that deal with programming the MSR of interest, as well as the conditions according to which this operation must either take place immediately or be delayed depending on if the interrupted thread is a registered one or not. However, we have not yet introduced the procedure by which our IBS interrupt handler is installed into the OS. To this end, we have to recall some concepts about the advanced programmable interrupt controller (APIC) and the operations that are cooperatively performed by the firmware and by the software at the OS level to accomplish the correct management of interrupts.

In Section 3.2 we introduced the Local-APIC (LAPIC) as the hardware that controls the delivery of interrupts to the relative CPU-core. It comprises a set of APIC registers which provide both information about the LAPIC itself and the mode of handling interrupts coming from internal and external sources. All APIC registers are memory-mapped into the 4KB APIC register space, each one aligned to 16B offsets, and can be accessed with memory read and write operations by specifying an address composed as follows

$$\mathsf{APIC\ Register\ address} = \mathsf{APIC\ Base\ address} + (\mathsf{Offset} \cdot 16)$$

Among the available sources of interrupts there are local ones whose delivery is redirected by the LAPIC to the CPU-core using an interrupt delivery protocol that has been set up through a group of APIC registers called the local vector table (LVT), which is a table that allows the software to specify the manner through which the local interrupts must be delivered to the CPU-core. There are various information that can be specified in the registers of the LVT table, among which we mention:

- the interrupt vector number, which is used to displace within the interrupt descriptor table (IDT)—it is a data structure used to implement a software-side interrupt vector table that the processor consults to determine the correct response to interrupts and exceptions—to retrieve the interrupt-gate entry that provides the address of the entry-point function where the handling of the involved interrupt begins;

- the delivery mode, which is used to indicate if the interrupt is specified by the vector number or it is of a different type (*e.g.*, SMI, NMI, INIT or ExtINT);

- the delivery status, which is used to discover if there is currently no activity for this interrupt source or it is delivered to the CPU-core but has not yet been accepted;

- the mask field, which is used to enable or to inhibit the reception of the interrupt.

Interrupts generated by the IBS facility are local ones. Indeed, there is an APIC register within the LVT table to specify how they must be delivered to the CPU-core. By the way, its offset in the LVT table can be retrieved from the 4 LSb of the MSR named IBS control register (IbsCtl), which can be used as shown in the expression above to produce the address where the memory-mapped APIC register resides. The latter is commonly not enabled at boot of the Linux OS, so that the exploitation of the IBS facility first requires to update this register. We do this by writing the APIC register with a 32-bit bitmap for which the mask field has been set to zero to enable reception of IBS interrupts. Also, the delivery mode has been set to zero (*i.e.*, Fixed) to indicate that IBS interrupts must be handled by passing through an interrupt-gate whose descriptor offset within IDT table is specified by the interrupt vector number. Finally, the interrupt vector number has been set to the same value stored into the LVT register devoted to the management of spurious interrupts—the latter are invalid signals on interrupt inputs which are usually caused by glitches resulting from electrical interference or malfunctioning devices—which allows to displace to the IDT entry that holds the address of the entry-point function named spurious_interrupt.

There are three main reasons why we use the spurious interrupt vector number instead of packing a new interrupt-gate descriptor to insert into the IDT table. The first one is because in the more recent versions of the Linux kernel, all IDT entries are made reserved while both data and function symbols used to update the IDT table are no longer exported to inhibit this possibility, hence they are not visible by kernel modules. The second one is that under normal functioning conditions spurious interrupts never occur throughout the execution. In addition, their handling does nothing but collecting statistics about the occurrence of this kind of interrupts, which enables us to alter the default behaviour without compromising the operation of the OS. The third reason is the most important and derives from the fact that this solution is compatible with the Page Table Isolation (PTI) patch included in all recent versions of Linux kernel to contrast security attacks based on hardware-level speculation, like Meltdown [50] and Spectre [45]. PTI provides that only few kernel pages are left mapped in the virtual page table (pointed by the cr3 register) when threads run in user mode, those that include the entry-point functions responsible for preserving the execution state of interrupted threads before the control is given to the actual interrupt handlers. So that, when an interrupt arises and the thread switches to kernel mode, its execution restarts from the beginning of an entry-point function that surely resides in mapped pages. The latter is then in charge of switching the cr3 register to point to the whole kernel page table before calling any other function or accessing global variables. Conversely, any function implemented in the kernel module does not belong to mapped pages at the time when an interrupt occurs. This means that, if we had used one of such functions as an entry-point then the execution would have ended with the arising of a segmentation-fault while running in interrupt context.

By using the same interrupt vector number used for spurious interrupts, the CPU-core that queries the IDT table after having received an IBS interrupt also retrieves

the same interrupt-gate descriptor that holds the address of the spurious_interrupt function. The latter is an entry-point function for interrupt handling that always belongs to mapped pages, even when threads are running in user mode. It is a stub written in assembly code which has been replicated for all APIC interrupts, but with the difference that each one of these entry-points will yield control to a different interrupt handler. By the way, the spurious_interrupt function invokes the so-called smp_spurious_interrupt one, which is a function coded in C language that, as we already pointed out, only collects statistics about the occurrence of spurious interrupts.

Overall, if on the one hand we mandatory need to rely on an already implemented entry-point function to be compatible with the PTI patch, on the other hand we have to put in place an hacking procedure to replace the call to smp_spurious_interrupt function with that to the handler provided by our kernel module. This is achieved by performing a binary inspection of the kernel code at the time at which the module is installed, intercepting first each call instruction present within the spurious_interrupt function (starting address stored in the IDT entry) and by comparing then the relative operands with the address of the smp_spurious_interrupt one. The latter can be retrieved either via the Linux kprobe service or by invoking the kallsyms_lookup_name kernel function. Anyhow, the final result is the same address value. Once the desired call instruction has been found, its operand is updated to point to the interrupt handler implemented by us, which concludes the hacking procedure pursued ahead during module installation. As a final aspect, we want to point out that this procedure is compatible with all the Linux kernel versions, regardless from the fact that PTI patch is included in the kernel image and enabled at OS boot. Thus, our solution is portable across distinct Linux distros and different versions of the installed kernel.

4.1 Effective Management of Task Priorities

Modern parallel applications are characterized by the presence of numerous, differentiated and fine-grained activity instances (*i.e.*, tasks) that may or may not be subject to data dependencies, whose dynamic materialization and activation at run-time is not predictable due to the non-deterministic evolution of the applications' execution state. It is clear that, these tasks are not all ready to run the same time throughout the execution but different subsets of them are allowed to execute in different instants of time. Furthermore, tasks can have assigned different priority values to indicate that some of them are much more critical than others. Whatever the set of tasks currently admitted to execute, some of them have higher priority values, which means that selecting these tasks to run before lower priority ones will lead to certainly obtain better performance results. In this regard, we have already pointed out how powerful can be the assignment of priority values to tasks by the programmer as it is the best tool he has to hint to the scheduler which are the tasks that much more than others deserve to be immediately executed. Thus, the reactivity of the system in starting tasks with the highest priority values determines the performance level that can be achieved.

Assigning priority values to tasks does not only provide practical evidence of

better performance when performing experimental evaluations of certain task-based applications, but it is also particularly relevant in the field of scheduling theory as the formulation of algorithms that are enabled to dynamically assign priorities to tasks, according to predefined rules, brings out important theoretical results about the schedulability of model-specific task systems (*i.e.*, periodic, aperiodic and sporadic models), as well as to provide provable bounds on the number of resources required to successfully schedule these task systems. By the way, dynamic priority assignment plays an important role in formulating scheduling algorithms in the context of real-time system. Here, tasks have assigned possibly different deadlines and are characterized by arrival time periods that can be fully, partially or not known a-priori depending on the task model of interest. Also, all tasks are preemptive in the sense that their execution can be made pausing whenever an higher priority task arrives in the system. The challenge is to design an efficient priority-driven scheduling algorithm that, given a feasible task system, is guaranteed to schedule the system to meet all deadlines.

One of the most popular algorithms used for priority assignment in run-time scheduling of tasks with deadline is the *earliest deadline first* (EDF) scheduling algorithm [51] that falls within the framework of priority-driven algorithms—jobs have assigned priorities in inverse proportion to their deadline—for which, despite it is known to be a not optimal scheduling algorithm for multiprocessor platforms, it is guaranteed to successfully schedule any periodic task system (*i.e.*, each job has its period as deadline) as long as two main constraints imposed on both the task system utilization factor (*i.e.*, the capacity bound) and each single task utilization factor are satisfied (we remind the reader to the original work for more details). Always remaining in the context of the scheduling of periodic tasks upon multiprocessor platforms, the authors of [81] have presented a priority-driven scheduling algorithm based on EDF that further assigns the highest possible priority to those jobs of a periodic task system whose utilization factor is beyond a particular threshold (*i.e.*, per task utilization factor introduced before), otherwise they have assigned priorities according to the original EDF. By relying on this additional priority-based scheduling logic, the authors were able to relax the restriction on the utilization factor of each individual task and to prove that the algorithm correctly schedules any periodic and feasible task system that satisfied the only constraint on the capacity bound introduced above. In another work [2], the authors extended the algorithm presented in [81] to the scheduling of aperiodic tasks (*i.e.*, future arrivals are unknown) upon multiprocessor platforms, for which they first claim and then prove that if the system utilization is, at every time, less than or equal to the capacity bound discussed before, then all deadlines are met. The authors of [30] have proposed instead a variant of the EDF scheduling algorithm which is provably superior to EDF in the sense that it schedules all periodic task systems that EDF can schedule, and in addition schedules some periodic task systems for which EDF may miss some deadlines. In this work, the authors have shown how, by assigning the highest possible priority to a certain number of tasks of an EDF-schedulable system (the first `k` tasks in the EDF-ordered sequence, where the parameter `k` is object of a minimization function), it could be possible to successfully schedule a periodic task system to meet all deadlines on fewer processors, a value that never overcomes that required by the original EDF algorithm to ensure schedulability.

Even if these theoretical works are ideally based on the assumptions that maximum execution times are known and task switches have no overhead costs, hence being poorly representative of the actual context within which parallel programs of our interest execute, we hope their presentation was useful to make the reader aware of the fact that the choice of a priority-based strategy for scheduling preemptive tasks instead of another can also give guarantees on a theoretical level, such as the schedulability of certain task systems (as a function of the associated environmental constraints), and therefore the capability of the algorithm in achieving its goal at best and in the most cases. Thus, in order to pursue (or try to get close to) these objectives also on a practical level, the execution of tasks with priority must be supported with adequate and lightweight solutions in order to realize a preemptive execution scheme, as any occurrence of priority inversion phenomenon—it is a scenario in which the execution of a higher priority task is delayed by a lower priority one, effectively inverting the relative priorities of the two—is adverse to the achievement of the desired performance levels as well as to the possibility to satisfy any constraints (or try to be penalized the least possible) for which fulfilment the algorithm may have been appositely designed.

Therefore, regardless of what is the application context and regardless of what measure is used to evaluate the goodness of the scheduling algorithm, applications consisting of tasks with assigned priorities are designed in this way to indicate that certain activities must have computing power immediately available for their execution, because from the system ability of promptly giving this power to the higher priority tasks also derives the capability of achieving the highest possible performance levels and that of satisfying certain requirements.

The task priority management aspect is dealt with in this chapter of the thesis, where we present how time-based micro-threads scheduling has been exploited in differentiated contexts.

4.1.1 Preemptive Software Transactional Memory

In this work we tackled the problem of efficiently handling the execution of tasks with assigned priority. More in detail, we refer to the running of transactional tasks for which the involved (concurrent) operations are carried out speculatively by the available worker threads that the transactional layer has appositely instantiated to achieve a higher level of parallelism than that achievable through conventional synchronization techniques with same number of threads.

Transactional Memory (TM) is a paradigm for the management of shared-data accesses on multi-core machines that enables the programmers to mark groups of in-memory operations as transactions. It attempts to simplify concurrent programming by allowing a group of load and store instructions to execute in an atomic way, that is, according to all-or-nothing execution semantic. Its formal definition was first introduced in [34], within which the authors presented a new multiprocessor architecture intended to make lock-free synchronization as efficient as conventional techniques based on mutual exclusion. In this regard, they defined a transaction as a finite sequence of machine instructions, executed by a single process, satisfying *serializability*—transactions appear to execute serially, meaning that the steps of one transaction never appear to be interleaved with the steps of another, or seen in

different orders by distinct threads—and *atomicity*—after having made a number of tentative changes to shared memory, the transaction either commits making its change visible to other threads instantaneously, or it aborts causing its changes to be discarded. When met, these last two properties ensure that the commitment of any transactional (lock-free) execution always leaves the program in a consistent state. By the way, this paradigm is inspired by the concurrency control schemes originally implemented within database management systems (DBMS) that were in charge of ensuring serializability of transaction schedules.

Under high-contention conditions, TMs have been shown to achieve better level of performance than fine-grain hand-made locking. In fact, conventional synchronization techniques, based on the concept of critical section, are clearly unsuitable on modern multiprocessors since they limit parallelism, increase contention for memory, and make the system vulnerable to timing anomalies and processor failures. Conversely, the key to highly concurrent programming is to construct classes of implementations that are *non-blocking*—a TM implementation is said to be non-blocking if the repeated attempt to execute some transaction by a thread implies that some thread (not necessarily the same one) will terminate successfully after a finite number of machine steps in the whole system.

Software Transactional Memory (STM) was then introduced in [79] to overcome the limits of the hardware implemented transactions proposed in [34], which are claimed to be not non-blocking by the authors of this work since a single thread running alone and being repeatability swapped out during the execution of a transaction will never terminate successfully. STM implementations provide instead the advantage of not requiring any specific hardware technology, by extending in this way their portability to the most of computer systems. This means that they also bypass the limitations imposed by the impossibility to commit a transaction that undergoes repeatability a mode switch, or by the L1 cache size that can be large not enough to accommodate wider write set—TM implementations based on hardware support require the write set of a transaction to be temporarily buffered at the cache level before it can be flushed to the lower levels in the memory hierarchy when the transaction successfully commits. For all these reasons, nowadays the TM implementations based on software support are still the most diffused ones.

However, despite the offered advantages, STM environments are still doomed to improvements, particularly for what concerns the management of differentiated transaction priority levels, which is an aspect of particular interest in the context of online transaction processing (OLTP) applications. OLTP is a class of applications for which users send requests to the system in charge of processing them as transactions. Such applications are characterized by a high throughput workload profile, whose transactions are mainly insert- or update-intensive. These systems are commonly implemented in the form of back-end tier servers in charge of serving different priority requests coming from some front-end systems, each one gathering same priority requests from clients, or directly from clients—a classical example of commercial OLTP systems is that of the automated teller machines (ATM) for banks.

In this context, the difficulty in handling transactions with differentiated priority levels originates from the fact that threads, employed by the transactional layer, carry out the execution of transactions as non-interruptible tasks, hence any thread can react to the materialization of a higher priority transactional task and take care of

its processing only at the end of the currently executed transaction. In fact, in their common implementations, TM systems simply delay the processing of an incoming high priority request up to the point in time where some thread ends its last started transaction and runs the routine devised to verify the presence of new requests. It should also be noted that generating additional threads just to carry out higher priority transactions is not an option, as it is generally not convenient to run STM applications with a number of threads exceeding the number of available CPU-cores, mostly because they show a CPU-bound execution profile which (unlike I/O-bound) is a condition that would lead to a scenario where multiple threads compete for CPU-usage, and which has been already shown to be likely adverse to this kind of applications [18]. Any attempt to alleviate such a problem by assigning higher CPU scheduling priority at the OS level to threads running higher priority transactions is not a definitive solution for managing threads that compete for CPU usage, as it might lead to starvation of lower priority transactions uncontrollable by the STM layer. Also, it would require expensive coordination between user- and kernel-space to keep a consistent mapping between CPU scheduling and transaction priorities in a scenario where requests with arbitrarily assigned priority values can arrive at the system. In addition to this, there might be other distinctive characteristics besides the priority value that can play an important role in determining the scheduling of same priority transactions, of which the OS cannot be aware nor it can take care of. As a final aspect, the dynamic spawning of a new thread as a reaction to the arrival of some higher priority request to run an in-memory transaction would also result in paying excessive overhead costs for the spawn operation.

It is clear therefore that a performing solution cannot rely on dynamic spawning of threads at run-time, as well as the transactional layer should never engage a number of threads that exceeds the number of available CPU-cores in order not to incur all the abovementioned issues. A solution to the problem of efficiently handling the execution of transactions with assigned priority values must instead be sought elsewhere. More precisely, we have to start from the reasons why current execution models of transactions do not provide the possibility to promptly cede control to higher priority transactions as soon as they arrive at the system, making in this way the transactional layer unable to react to the occurrence of priority inversion. One of the causes is certainly the OS-thread centric nature that currently characterizes the technology at the base of the execution of transactions in all modern STM frameworks, which places some immediate constraints on the scheduling of transactions along the execution path of one or more threads. Compounding this inability, there is the total absence of any hardware and software support aimed to help threads to switch in executing a different transaction at arbitrary points of their execution without compromising the execution state of the ones being suspended.

Conversely, the implementation design of these frameworks should be based on a technology that differs from classical ones in that it must provide worker threads with the ability to periodically suspend the execution of current transactions in order to check if higher priority transactional requests are present in the system, and if so, switch to executing one of them. This is ultimately represented by the ULMT technology that we have presented in Chapter 3, which provides the STM systems with facilities and supports from the underlying layers to accomplish the commissioning of the micro-threading model for the execution of transactions. With

the aid of the innovative ULMT technology, we were able to pursue a paradigm shift where the execution of an in-memory transaction is carried out as a preemptible task, so that a thread can start processing a higher priority transactional task before finalizing its current transaction. This is made possible since the execution context of each transaction is that of a distinct micro-thread, which is appositely initialized for each transactional task every time a new request arrives at the system, making sure that the execution of each transaction can adhere to the micro-threading model. Our STM solution is therefore based on a fully new management scheme of differentiated execution contexts within the STM layer, so that any transaction context-switched off the CPU is not aborted, rather, it will be eventually resumed so that its outcome will be only determined by possible data conflicts, as typical of the TM paradigm. Overall, in our approach we promptly nest the execution of a higher priority transaction along the execution path of an already active thread with no need to spawn additional ones, thus preventing all the aforementioned problems related to CPU competition by multiple threads in STM systems. Additionally, by still relying on the ULMT technology, we avoid CPU under-utilization that would be caused either by statically devoting specific threads to process higher priority requests or by binding transactional tasks to a single thread throughout their execution, given that in our architecture each thread can be in charge of processing whichever in-memory transaction at any time instant, regardless of whether it is a new standing one or a previously suspended one. The resulting implementation is a preemptive STM environment which is able to exploit the IBS hardware facilities that we have presented at the beginning of this chapter in order to program the receipt of periodical interrupts, which in turn allow to enact fine-grain periodical control flow variations along any running thread with minimal run-time overhead.

We want to make notice that, if a signaling mechanism were used to notify the materialization of a new request, such as Posix user-defined signals, the timeliness of the signal delivery to the destination thread would be bound to the conventional OS timer-interrupt interval which, as we already pointed out in previous chapters, is a value in the order of few milliseconds (common setups range from 1 to 4 μsec) that, compared to the grain of TM transactions, would delay too much the activation of the signal handler able to detect the presence of the standing high priority transactional requests, hence not fully adequate for promptly dispatching the latter. It is equally important to point out that, if we had chosen the historical user level thread technology (ULT) to enable the time interleaved execution of different code blocks along a same thread, we would not have obtained an application transparent implementation of the preemptive STM environment since the programmer would have been responsible for injecting calls to ULT API functions at specific points of the STM application code.

Our approach is instead fully transparent, so that the programmer of the STM application does not need to care about the management of transaction priorities and control flow variations. He only needs to code the data access logic, while the actual passage of control to higher priority transactional requests is achieved in our architecture via actions performed by the runtime environment with the aid of services offered by the ULMT technology. Moreover, given that preemption of lower priority transactions takes place on the basis of fine-grain hardware interrupts, we also avoid at all context-switch delays that would be potentially experienced in

some hypothetical architecture based on signaling mechanisms or on classical ULT technology, for which a possible scenario is where the lower priority transaction currently running along a thread does not timely reach the point of the call to the ULT API functions, which is a possibility enhanced by the fact that a transaction can repeatedly abort at arbitrary points of its execution depending on the actual concurrency level that affects the program progress.

The only solution we are aware of which discriminates between transaction priorities in STM systems is the one in [54]. In this work the authors cope with quality of service in STM applications and propose an approach where transactions that are subject to deadlines, and experience abort retries due to conflicts, tend to execute more conservatively (*e.g.*, by eager locking data) while getting closer to their deadlines, since eager locks will lead these transactions not to be aborted because of data conflicts. In any case, this work does not make systematic use of preemption in order to enable the timely processing of higher priority transactions along the execution path of the threads running the STM application, which is instead the fulcrum of our proposal.

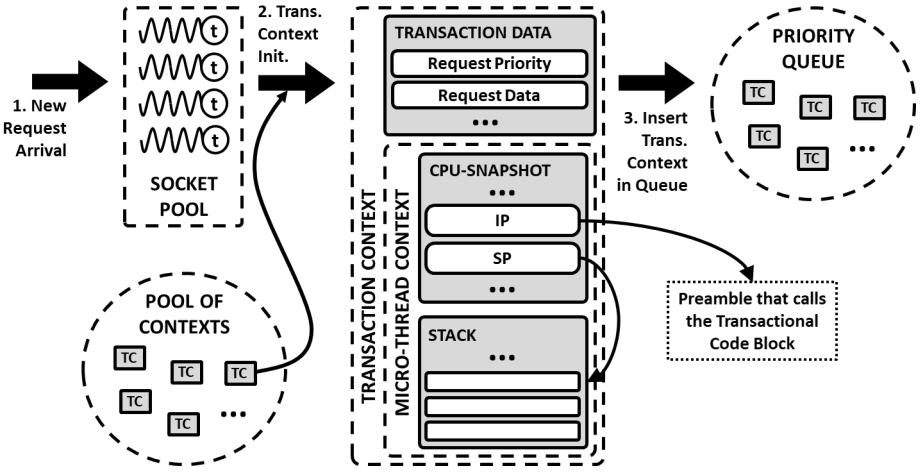

Figure 4.2. Basic architectural organization of the preemptive STM environment.

In Figure 4.2 we show a high level schematization of our preemptive STM architecture, which is targeted at back-end STM environments. A classical socket pool is handled in order to receive requests for executing data manipulations transactionally, which come in from some front-end system. Upon its receipt, a request is first associated with a transaction context data structure—it is in effect a micro-thread context plus additional information of interest for the execution of the involved transaction—retrieved from a pool of already initialized contexts and then is placed into a priority queue, by associating it with the corresponding priority level. With no loss of generality we assume the priority level is explicitly marked within the transaction request, together with the function to be run by the STM environment for serving the request, and its input parameters.

When associating a transaction context to an incoming request, the same data structure is also used to keep track of the priority information for the transaction,

the function to be run and its parameters, as well as information on what happened along the transaction lifetime, such as the number of times it has been preempted and context-switched off the CPU in favour of a higher priority transactional task. We exploited this information in order to dynamically change the actual priority of a transaction according to a feedback scheme aimed at improving performance. When a transaction ends its processing phase, the context it is using is released to the pool in order to be reused from a subsequent incoming transactional request, in a fresh incarnation of its content. In this regard, the value of NUM_CONTEXTS is a configurable parameter and determines the maximum number of transaction contexts that are admitted to the processing stage simultaneously. When no context is available from the pool, incoming transactional requests are not migrated to the priority queue. This migration is resumed as soon as the termination of already active transactions will lead to releasing contexts to the pool. On the one hand, fixing the maximum number of contexts that can be added to the priority queue prevents the latter from being persistently accessed for the insertion of new transaction contexts as the average rate with which these data structures are inserted into the priority queue does not overcomes the average throughput. On the other hand, our architecture is intentionally devised in order to manage more transaction contexts than worker threads, since transactions can be preempted, hence suspended and eventually resumed later. Therefore, the parameter NUM_CONTEXTS should be set to a value significantly greater than the number of worker threads selected for running the STM application, but not too large to prevent the overloading of priority queue. Nonetheless, as long as a reasonable arrival rate is configured for the experiments (values that well represent the workload of real world application scenarios), the pool of contexts is unlikely to get empty throughout the execution.

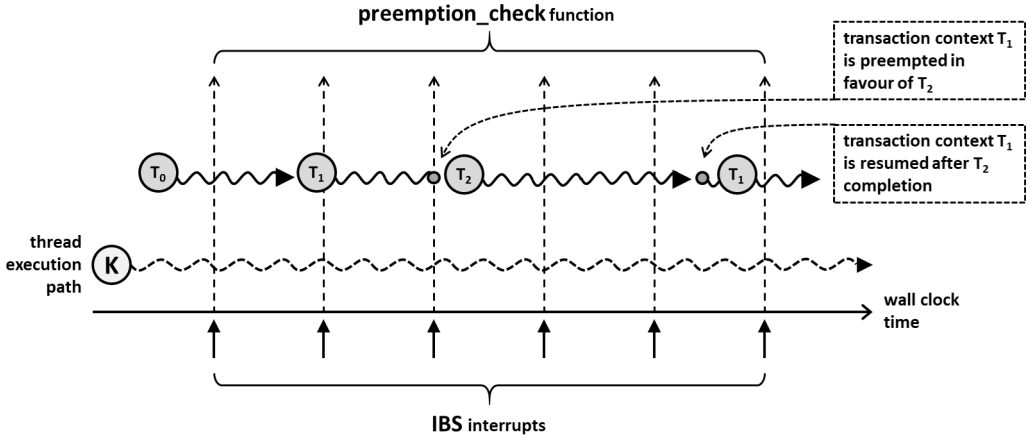

Figure 4.3. Fine-grain interrupt timeline.

The job of receiving requests from sockets and inserting them into the priority queue is done via dedicated threads, whose execution profile is clearly I/O bound. Hence, request insertion into the priority queue takes place off the critical path of the worker threads running the STM application. This enables to find the most up-to-date state of the priority queue every time a fine-grain periodical control flow variation occurs along any worker thread to verify the need to pass control to

some standing higher priority request. In their turn, worker threads that undergo control flow variation due to the arrival of IBS interrupts are shifted to execute a user-space function, called PREEMPTION_CHECK, whose code was not directly placed along the interrupted execution trace (*i.e.*, out-of-the CGF provided for the involved transaction), and which implements the preemption management policies at the core of our STM environment. If PREEMPTION_CHECK determines that a different transaction needs to take control of the CPU-core, the currently processed transaction is preempted, and its context is enqueued again within the priority queue, while the transaction context of the higher priority transactional request is installed so that the worker thread can start processing it. As soon as a worker thread ends the processing phase of its current transaction, it releases the no-more-in-use context to the pool, and then queries the queuing data structure in order to take care of activating, or resuming in case of a previous preemption, the transaction that currently stands at the highest level of priority, if any. Figure 4.3 shows an execution example of transactional tasks in our preemptive STM environment, in which the transaction T_1 is context-switched off the CPU-core in favour of the higher priority transaction T_2, whose context is installed along the execution path of the involved worker thread in order to promptly carry out its operations. Anyhow, once T_2 has successfully committed, transaction T_1 can be resumed to continue its speculative execution up to the commit. We want to make notice that, in this particular example, transaction T_1 resumes along the execution path of the same thread on which it has been previously suspended, but in general nothing could prevent it from being resumed by any other thread that was idling, as admitted and provided by the micro-threading model.

However, we have not yet gotten into the details of the priority queue, whose structure design mostly depends on the employed management policies that we have appositely designed to cope with those problems that are typical of speculative computation and that characterize the execution of all transactions. As a first aspect, the priority queue we use in our preemptive STM environment includes a couple of lists <ACTIVE,STANDING> for each of the managed priority levels. The STANDING list keeps all the contexts associated with transactions having a given priority, whose execution has not yet been started—these transactional requests have been delivered but have not yet been admitted to the processing phase along any worker thread. Conversely, the ACTIVE list keeps track of all the contexts associated with transactional requests at that priority level, which have already been started by some worker threads, and have then been context-switched off the CPU—the involved transactional tasks have already been preempted at least once after being admitted to the processing phase. As a second aspect, a compact bitmap is used to determine whether any given priority level has at least one element within the corresponding <ACTIVE,STANDING> lists. Hence, as soon as a worker thread accesses a priority queue for determining what is the highest priority level that is currently keeping some request to be started or resumed, such determination takes place via fast bitwise instructions. Also, depending on the number of bits returned by an in-memory read operation, the total number of memory accesses to find a queued transactional request decreases accordingly since no sequential search on all list pointers is required. In Figure 4.4 is reported a graphical representation of the priority queue with some transactional requests linked to it as an example.

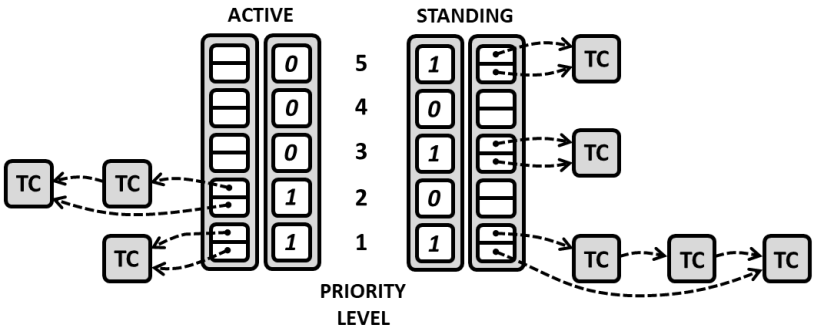

Figure 4.4. The priority queue.

For what it concerns the assignment of transaction contexts to worker threads, among those belonging to the same priority level, the logic provides to always favour the transactions that are currently residing within the ACTIVE list—called *hot* transactions. In this way, elements within the STANDING list—called *cold* transactions—are considered for CPU-dispatch only if the ACTIVE list is currently empty. In addition to this, the policy for managing each of the two lists is First-In-First-Out (FIFO), so that the oldest transaction in the list is always selected for CPU-dispatch before the others. This separation between hot and cold requests within a given priority level, with hot requests favoured over cold ones, as well as the FIFO policy for managing each single list, have been exploited precisely to keep into account the peculiarities of in-memory transactions handled by common STM layers. More in detail, after the start of a transaction, the longer the length of the time interval for reaching the commit phase, the higher the likelihood of observing a conflict with some concurrent transaction. Specifically, delaying the finalization of an already started transaction—because of context-switches that lead a transaction to wait an unpredictable amount of time after being suspended into one ACTIVE list—leads to a stretch of the so-called *vulnerability window* [54]. This in turn, may lead to an increased likelihood of abort, a phenomenon adverse to performance. In the end, keeping the already started transactions as hot records within the ACTIVE list, and favouring them over the cold transactions kept by the STANDING list, contrast the stretch of the vulnerability window. It is clear that this approach has an impact on the total waiting time of a transaction request to be admitted to the processing phase for the first time, which in turn affects its response time. Nevertheless, the amount of speculative computation that would have been wasted due to excessive stretch of the vulnerability window, as a consequence of the increased abort probability, would have been greater than the initial wait.

On the other hand, the stretch of the vulnerability window of an already started transaction can also be caused by repeated context-switches which can always arise in our preemptive STM environment due to the presence of higher priority requests within the priority queue. To cope with this orthogonal problem, we have designed a feedback mechanism such that the actual priority level of an already started transaction is dynamically updated at run-time. In particular, for each active (*i.e.*, hot) transaction we keep track of the number of times it has been preempted in favour of a higher priority one. We denote the counter of context switches involving

transaction T as C_T. As soon as the value of C_T reaches a threshold that we denote as C_{max}, the transaction is migrated to the highest priority level, so that no further delays caused by preemptions will be induced on it. The responsiveness of such a feedback mechanism clearly depends on the value of C_{max}, since greater values of this parameter will tend not to promote the priority of the transaction along its lifetime. Conversely, setting C_{max} to the minimum value 1 would lead any transaction to reach the maximum priority level right after the first occurrence of preemption. This, in turn, would lead to flatten the actual priorities of active transactions to the same value, with consequent scarce possibility to discriminate among transactional requests born with different priority values.

So, if on the one hand a larger values of C_{max} tends to keep dynamic priorities more aligned to the original ones by preventing their flattening to the maximum priority level, on the other, it does not help a transaction that repeatedly undergoes preemption to avoid being aborted with high probability due the stretch of the vulnerability window. To this end, we devised and implemented a variant of the aforementioned dynamic priority assignment mechanism which provides to promote by 1 the priority of a transaction each time its counter C_T also increments by 1, as long as the inequality $C_T < C_{max}$ holds. This *lazy priority promotion* scheme has the potential to tackle the stretch of the vulnerability window of an active transaction, while still not favouring the flattening of the dynamic priorities to the maximum priority level admitted in the system. Indicating with P_T the current priority of transaction T, which initially corresponds to the priority level originally assigned to the transactional request, the variation of the priority P_T upon preempting transaction T, with consequent increment of the counter C_T, takes place according to the following scheme:

$$P_T = \begin{cases} min(P_T + 1, P_{max}) & \textit{if } C_T < C_{max} \\ P_{max} & \textit{otherwise} \end{cases}$$

where we denote with P_{max} the maximum admitted priority level within the priority management scheme.

One important final aspect to consider relates to how IBS interrupts delivered to threads need to be handled in case they are received while the target thread is currently executing some function offered either by the STM environment or by some library, rather than application code. This might be the case when the thread runs the TM_COMMIT statement for the transaction it is currently processing, as well as classical TM_READ and TM_WRITE services, which map operations on shared data by the application code to those managed by the STM system. Given that these functions might execute critical actions such as locking data—several STM implementations relay on commit-time-locking algorithms, *e.g.*, the one in [11] locks data in the transaction write-set for atomically installing all the newer versions upon a successful finalization—preempting the transaction execution while one of these functions is in progress may hamper both performance and correctness. In other words, we need to leave these functions execute as non-preemptible code regions. In this regard, in Chapter 3 we presented a wrapping-based approach devised to prevent a thread from being preempted when it is executing a non-preemptible code region on behalf of a given micro-thread. Given that in our preemptive STM

environment transactional tasks perform their operations within the context of an assigned micro-thread, the same approach must be used to avoid transactions from being context-switched off the CPU when they are executing the abovementioned functions. A per-thread preemption_counter variable is atomically incremented by 1 whenever a transaction's execution flow enters a non-preemptible region. Analogously, it is atomically decremented by 1 when exiting from these code regions. As long as the value of preemption_counter is not zero, the executing transaction is not allowed to be preempted in favour of any other transaction, not even for the higher priority ones. The CFV_TRAMPOLINE function, executed downstream of the control flow variation performed by the IBS interrupt handler, is in charge of verifying that the aforementioned counter is equal to zero before giving control to the previously discussed PREEMPTION_CHECK function. Otherwise, the latter function in not invoked and the execution resumes exactly from where it left off before.

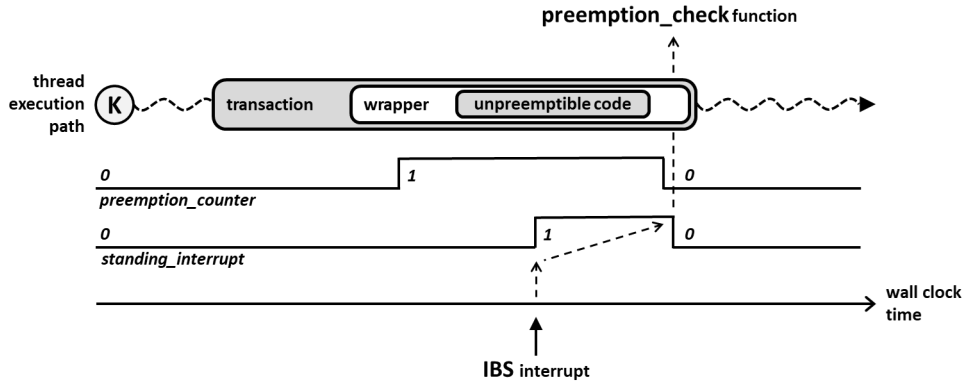

Figure 4.5. Standing IBS interrupt and time shift of preemption.

However, the drawback of this approach is that the delivery of an IBS interrupt to the worker thread is somehow lost, in terms of its potential for promptly passing control to some higher priority transaction. To cope with this aspect, we added a second per-thread variable, named standing_interrupt, which is set to *true* by the CFV_TRAMPOLINE function exactly when an IBS interrupt is delivered to a thread having preemption_counter set to a value greater than zero. In Figure 4.5 is shown an example of setting the standing_interrupt. As soon as the transaction's execution flow leaves the non-preemptible region which finally leads to setting preemption_counter to zero, standing_interrupt is checked by the wrapper. If it is found to be set to *true*, then the wrapper resets it and invokes the PREEMPTION_CHECK function, which this time will actually run the preemption policy. In other words, we shift the management of preemptions along the time axis at the earliest point in time such that no critical action is still in place along the execution path of the thread.

As a final aspect, our preemptive STM environment has been implemented by using TinySTM [21] as the baseline TM layer. We note that TinySTM has the possibility to be configured with either encounter-time-locking (ETL) or commit-time-locking (CTL) of the data accessed by a transaction. Since we have introduced within TinySTM a fully innovative preemption facility, we decided to experiment with CTL configuration as the use of ETL would require the preemptive approach

to be complemented with a suitable contention management scheme for resolving priority inversions, which in turn would require to abort lower priority transactions that can have locked data whenever higher priority ones claim to access the same data, regardless of whether the former have been suspended or are currently running along the execution path of some worker thread.

We tested our environment on top of a 64-bit HP ProLiant server, equipped with four 2.0GHz AMD Opteron 6128 processors and 64GB of RAM (8 NUMA nodes). Each processor has 8 cores, for a total of 32 CPU-cores. The operating system is OpenSuse 13.2, with the version 3.16.7 of Linux kernel installed.

In our experiments, we used 16 worker threads in charge of processing transactions, and 5 threads in charge of managing I/O operations on the socket pool. This configuration leads the STM environment to use no more than 65% of the overall available CPU capacity, hence it allows to assess our proposal in scenarios avoiding interference on the measurement of performance parameters, which could be caused by CPU competition with other processes and services running on the system. The workload generator issuing transactional requests has been run on another multicore machine with the same technical specifications of the one hosting the STM environment, which we described above. The two machines are then connected via a switched 100Mb Ethernet. The IBS operation sampling interval has been configured to match 100 microseconds, a value definitively lower than the LAPIC-timer interrupt period adopted in the configuration of the Linux kernel, which was set to 1 millisecond. Finally, the parameter NUM_CONTEXTS has been set to 1024, a value that enables keeping active a number of transactions definitely larger than the number of worker threads processing them.

Table 4.1. Transaction profiles and associated priority levels.

Transaction Profile	CPU Demand	Priority Level
delivery	$\approx 5\ msec$	1
stock level	$\approx 650\ \mu sec$	2
new order	$\approx 350\ \mu sec$	3
order status	$\approx 10\ \mu sec$	4
payment	$< 10\ \mu sec$	5

In order to evaluate the effectiveness of our preemptive approach, we have performed an experimental evaluation that relies on a port of the TPC-C benchmark [78] to STM. TPC-C is representative of OLTP workloads and includes 5 different transaction profiles that simulate a whole-sale supplying items from a set of warehouses to customers within sales districts. Thus, the benchmark is centered around the principal activities (transactions) of an order-entry environment, which includes entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. In our experiments we instantiated one district, and generated a workload made up by requests equally spanning the whole set of the 5 different transaction profiles specified by the benchmark. Also, it must be noted that transactions belonging to the different profiles exhibit very different CPU demands. In our port to the target STM environment, CPU demands

range from tens of microseconds to milliseconds. This peculiarity has been exploited in our experiments in order to determine a transaction priority scheme where shorter-running transactions have assigned higher priority. In this regard, shortest-job-first scheduling (SJF), with preemption in our case, is a classical strategy for managing priorities in computer systems, which typically allows the optimization of server-side run-time dynamics. Overall, in Table 4.1 we report the list of transactional profiles we have exploited from TPC-C in association with the order of magnitude of the CPU demand for processing them and the corresponding priority level we assigned while testing our preemptive STM environment.

We setup the workload generator to inject 25.000 transactional requests per second, issuing a total number of 6 millions transactional requests along the experiment lifetime. This peak-load phase is suitable for assessing the potential of an optimized preemptive CPU-dispatching scheme, and its actual advantages in the management of differentiated transaction priorities. The indication of peak-load has been evidenced by having the pool of contexts highly busy (above 90%) for most of the duration of experiments. The reported performance results have been computed as the average over three repetitions of the experiment.

In Figure 4.6 we show the average turnaround time for transactions born at the 5 different priority levels. It is computed as the sum of all the times spent by a transaction either for actual processing activities or while being kept within the priority queue—either as a cold or a hot transaction. Also, if a transaction is aborted and then retried, any aborted run contributes to the turnaround time of the transaction. Always referring to the same bar chart, the bar labelled as BASELINE refers to a scenario where the STM system does not follow the micro-threading model for the execution of transactions, that is, the STM layer does not employ the ULMT technology and, as a consequence, does not use the IBS hardware facility. This means that, once the transactions have been assigned to some worker threads, they are executed as non-preemptible tasks with no possibility of nesting the execution of higher priority ones. Therefore, in the BASELINE configuration, a thread passes control to a standing higher priority transactional request only at the end of the processing phase of the currently executed transaction. For completeness of the analysis we also considered a setting, labelled as IBS - NO PREEMPTION, where the IBS hardware facility has been enabled by the STM system, which indeed rely on the ULMT technology to accomplish the commissioning of the micro-threading model for the execution of transactions, but no preemption is ever actuated—once activated the PREEMPTION_CHECK function, as a result of the control flow variation procedure, it returns immediately by restoring the interrupted execution trace. This configuration is useful for the assessment of the overhead caused by the micro-thread execution logic—when applied to the execution of transactions—plus the overhead caused by the handling of periodical IBS interrupts compared to the BASELINE case. Also, the preemptive STM architecture we have presented has been assessed by considering different settings for the value of C_{max}, and by either including or excluding the lazy priority promoting scheme for the management of the dynamic priority of transactions.

By the results we see how, compared to the BASELINE, the preemptive approach reduces the average turnaround time of transactions born at higher priority levels (*i.e.*, levels 4 and 5) by around 60%-65%. Also, transactions born at the middle

priority level (*i.e.*, level 3) exhibit an average turnaround latency essentially not penalized by preemption, or even slightly favoured, while transaction born at lower priority levels (*i.e.*, levels 1 and 2) show a penalization of their average turnaround time which is mostly limited to less than 5%, and no more than 15% in the worst case. As expected, the higher advantages for higher priority transactions are achieved with larger values of C_{max}, which lead to delaying the dynamic increment of the priority of transaction born at the low priority levels. Moreover, the results obtained by the experiments performed under the IBS - NO PREEMPTION configuration show performance essentially aligned with the one of the BASELINE, indicating negligible overhead caused by the micro-thread execution logic and the handling of IBS interrupts together.

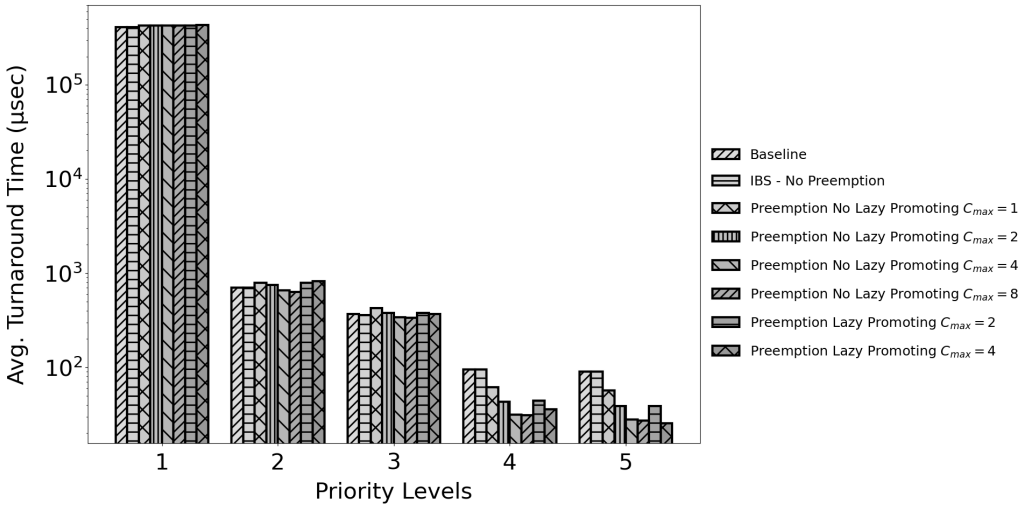

Figure 4.6. Average turnaround time for transactions born at different priority levels.

If we now focus on the results obtained by the experiments performed with the preemptive approach and without having used the lazy priority promoting scheme, we can see how the average turnaround time of transactions born at all the priority levels, but the lowest priority one, decreases as the value of the configured threshold C_{max} increases. This is clearly due to the fact that transactions born at the low priority levels are less likely to be promoted to higher priority levels, thus not affecting the waiting times in the priority queue of both hot and cold high priority transactions, a performance gain that comes at the expense of the average turnaround time of transactions born at the lowest priority level. A similar behaviour can also be observed in the results obtained by the experiments performed with the preemptive approach and with the use of the lazy priority promoting scheme. Anyhow, in this case, transactions born at the low priority levels much more likely will be promoted (incrementally) with the occurrence of repeated preemptions, thus causing a more accentuated imbalance of the workload towards the high priority levels, albeit less flattened on the highest priority one. The latter does not only affects the average turnaround time of transactions born at the middle-high priority levels (especially the cold ones), because they clearly have to compete with long-running transactions

promoted from the lower priority levels (which are hot ones at this point), but it also affects the average turnaround time of transactions born at the low priority levels as their waiting time to be admitted in the processing phase for the first time increases due to the aforementioned workload imbalance which, by involving a larger number of long-running transactions actually promoted with respect to the configuration without lazy promotion of priority, causes the overall execution dynamics to slightly deviate from those expected by the SJF execution scheme, which we found to be the one that would optimize the server-side run-time dynamics.

Just to make an example, let's think to the scenario in which a single low priority (*e.g.*, level 1) long-running transaction is preempted and therefore lazily promoted to the next priority level (*e.g.*, level 2), which is an event much more likely to occur compared to the promotion only for exceeding the threshold C_{max}. Since this active transaction (hot one) has earned one priority level, it will be picked to continue execution before any other standing transaction (cold one) that is born at this next priority level. Thus, these transactions are much more likely delayed in the access path to the processing phase, affecting in their turn also the waiting times for the first processing of low priority transactions. If the threshold C_{max} is set to large values, this behaviour propagates up to a certain priority level because even the lowest priority transactions will successfully commit after being preempted a finite number of times, less interfering therefore with the scheduling of transactions born at the highest priority levels (*i.e.*, level 5). This explains why the average turnaround time of transactions born at the highest priority levels is (slightly) favoured by the configuration that employs the lazy priority promoting scheme with the larger C_{max} value, at the expense of transactions born at the middle and the low priority levels.

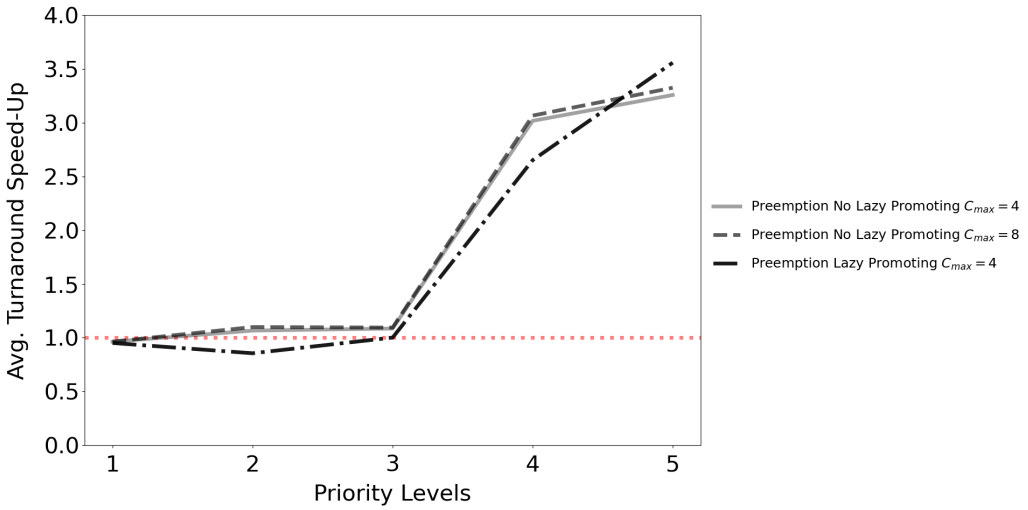

Figure 4.7. Speedup - ratio between the turnaround time of the baseline configuration and the turnaround time of the preemptive configuration.

In order to better outline the effects generated by the preemptive approach, we report in Figure 4.7 the ratio between the average turnaround latency provided by the BASELINE and the one provided by the preemptive approach, namely the speed-up on the turnaround time provided by the preemptive solution. For this

plot, we decided to show only the most promising configurations of the preemptive solution, which have been selected on the basis of the results shown in Figure 4.6. The best configurations are those with larger values of C_{max} (*i.e.*, thresholds 4 and 8), for which the plots show the effectiveness of our preemptive approach in both lazy promoting and no lazy promoting scenarios. The speed-up curves shown in this figure also confirm the reasoning made earlier about the interference effects that middle and low priority transactions can have on the scheduling of transactions born at the highest priority level. It can be noted indeed how the curve corresponding to the configuration that employs the lazy priority promoting scheme never overcomes the other two but for the highest priority level, a speed-up on the turnaround time achieved at the expense of transactions born at all the other priority levels.

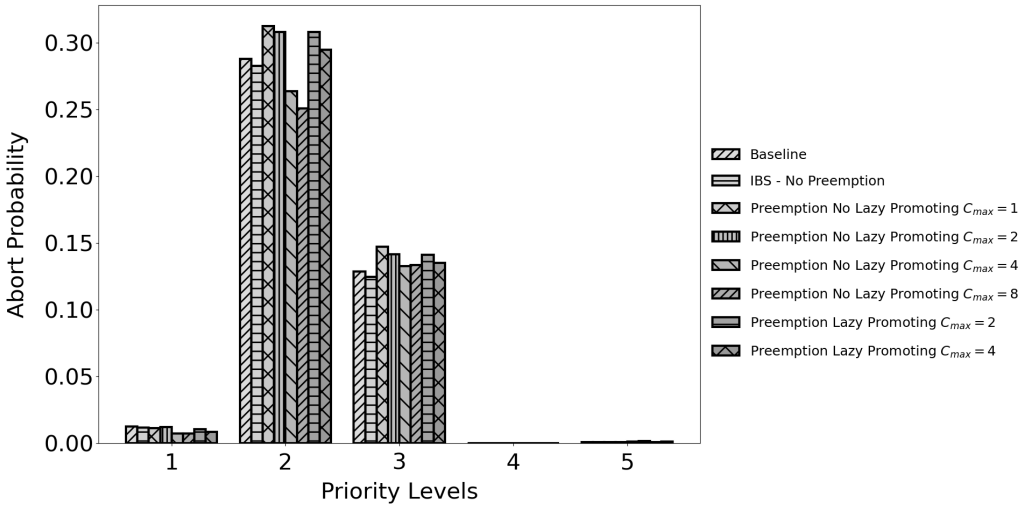

Figure 4.8. Variation of the transaction abort probability.

Finally, in Figure 4.8 we report data indicating how the probability of abort varies in the different configurations. As stated before, this variation can be caused by the effects of preemption on the length of the vulnerability window of the transactions. By the results we see that transactions born at priority level 2 are those more impacted by this phenomenon, immediately followed by those born at priority level 3. In particular, they show an increase of the abort probability—with consequent need for retries that lead to stretch the turnaround latency—for lower values of C_{max} and/or when lazy promoting is employed. Regarding this aspect too, the main causes originate from the high interference introduced by transactions born at priority level 1 which, by dynamically acquiring a higher priority, lead to increased concurrency between these long-running transactions and the shorter ones born at priority level 2. The opposite behaviour, with a reduction of the abort probability of transactions born at priority level 2, is instead noted when running with larger values of C_{max} and lazy promotion has not been included. This is because, in a preemptive STM environment with little propensity to promote transaction priority, the read-only stock level transactions born at priority level 2 are less affected by the effects of concurrency generated by the write-intensive delivery transactions born at priority level 1 compared to what happens when running with the BASELINE

configuration, for which these long-running low priority transactions have shorter turnaround times and much more frequently commit a huge amount of updates in memory that will likely invalidate the execution of the read-only ones.

Overall, the experimental data support the effectiveness of our preemptive approach in favouring the turnaround time of higher priority transactions, compared to a baseline scenario that manages priorities according to a non-preemptive scheme. Also, the hardware and the kernel level supports we have employed for handling preemptions has been shown to induce negligible overhead, which further favours our solution. Such a result also confirms the benefits that the adoption of the micro-threading model, as a new execution model of tasks supported by the innovative ULMT technology, can provide in order to achieve certain performance objectives in application contexts, such as the OLTP one, where the rapidity of the system in renewing the assignment of work to CPU-cores makes the difference.

4.1.2 Task Management in OpenMP Applications

Due to the advent of multi-core machines, parallel programming has become the mainstream approach to develop modern software applications. At the same time, in order to help programmers to structure their parallel applications, several paradigms and runtime environments have emerged, each one providing the programmer with different parallel programming models to follow to take advantage of the parallelism offered by the underlying resources. Such solutions, which follow different paradigms, therefore provide different tools with which the programmer can build parallel programs that target this specific computing environment (*i.e.*, multi-core shared-memory architectures). Over the years, several parallel programming languages and application programming interface (API) frameworks have been presented in the literature with the aim of giving the programmer simple solutions to code parallel applications according to one parallel programming model or another, and with the promise of providing high exploitation of parallel hardware resources. These range from purely functional programming languages, such as Erlang or Clojure, to imperative programming ones, among which we find Unified Parallel C (UPC), Charm++, Chapel, X10 and so on. These in turn differ from each other by the fact that they follow different programming paradigms, such as multi-threaded, agent-based or asynchronous message-driven ones, in such a way as to provide the programmer with a methodology to programmatically follow for dispatching the work to the parallel processing entities. Also, they differ in the way according to which units of works are specified and decomposed at run-time, the communication pattern and the handling of concurrency over data and objects, which are characteristic of a specific parallel programming model.

Some of the parallel programming languages that historically have had greater importance in the context of high performance computing on multi-core shared-memory architectures, with several important results also in the state-ot-the-art, are Cilk [5], Cilk++ [48] and their successor Cilk Plus [38] developed by Intel, which are general-purpose programming languages designed for multi-threaded parallel programming. They are based on the C and C++ programming languages, which they extend with constructs to express parallel loops and parallel sections according to the *fork-join* model—it is a parallel design pattern for which the execution branches

off in parallel at designated points in the program to later rejoin into a single flow, which takes place with the spawning of processes or threads devoted to carrying out distinct execution instances of the same function or code block. The execution model they implement lays its foundations in the capability of spawning child threads for the execution (possibly in parallel) of designated functions to which arguments are passed in a C-like fashion. Execution of threads is then assisted by the activation of stack frames on top of a so-called *cactus* stack, which behaves similar to an ordinary C stack, unless threads generated along different branches of the spawning tree are running in parallel on distinct processors, such that each one of them has its own view of the stack that corresponds to its call history. By the way, the major contribution of Cilk was that of having employed a *work-stealing* scheduling policy according to which processors that have already finished their local work are allowed to act as thieves of the work of others. In fact, by relaying on the concepts of *work* (sum of the execution times of all the threads) and *critical path* (largest sum of thread execution times along any path in the spawning tree), the author were able to analytically and empirically prove that, for well-structured Cilk programs, work-stealing scheduler achieves execution space, time, and communication bounds all within a constant factor of optimal. Nonetheless, the execution pattern that characterizes the threads when they have to synchronize with their children due to data dependencies requires that the handling of return values must take place in separate threads (*i.e.*, *successor* threads). If on the one hand the latter approach simplifies the Cilk runtime system, on the other it is onerous for the programmer who has to explicitly write code to spawn successive threads and to send back to them the return values generated by the child threads. Also, the programmer can only rely on very few keywords to implement the different parallel execution patterns. In 2017, Cilk has been marked as deprecated in the release 7.1 of GNU Compiler Collection (GCC) and then removed in the release 8.1. The next year, Cilk Plus is being deprecated in the 2018 release of Intel Software Development Tools. Anyhow, it has been a reference model for the design and evolution of the specifications of modern and more sophisticated parallel programming ones.

This is OpenMP [63], abbreviation for Open Multi-Processing, which is an API that may be used to explicitly direct multi-threaded shared memory parallelism, and has become the standard de-facto for coding shared-memory parallel applications in C, C++ and Fortran. By the way, it is currently supported by GCC, Intel C++ Compiler (ICC), and the C-Language (CLANG) family frontend of LLVM compiler. OpenMP uses a portable, scalable model that gives to programmers a simple and flexible interface for developing parallel applications for platforms that range from off-the-shelf computers to supercomputers. It consists of three primary API components: compiler directives, library routines, and environment variables, which together influence the run-time behaviour. Compiler directives, in the form of #pragma clauses, are used by the programmer in order to specify that a function (or code block) represents a parallel region, that is, a team of threads is appositely created in order to carry out in parallel the involved region according to the fork-join model. Other directives, namely the work-sharing ones, are then used to diversify the execution along each thread by distributing sub-sections of the entire region among the members of the team, or by assigning subsequent iterations of code within a looping statement, which would have been performed serially in a non-parallel

execution, to different threads (*e.g.*, for loops). The compiler is therefore directly involved in the process of manipulating the intermediate representations (IR) of code along the path that finally leads to the generation of the executable. This is done by intercepting #pragma clauses of interest that are expanded to calls to library routines which ultimately embed the logic that allows to apply one parallelization strategy rather than another. Very often, this manipulation involves rearranging completely the code structure with the insertion of new functions with mangled names in order to accommodate the code that resided under the scope of specific OpenMP statements (*i.e.*, executable directives). Such a restructuring of the code clearly alters the data environment of variables associated with the execution of these regions. This is somehow related to the OpenMP memory model which is a relaxed-consistency memory model, whose structure provides the existence of per-thread *temporary view* of the memory within which threads are allowed to cache variables that are only later flushed in shared memory at designated synchronization points—these are generally represented by the entry point and the exit point of these regions—that make the same order through which in-memory operations were performed by different threads visible to all of them. By the way, OpenMP specification also provides a directive called *flush*, which mostly acts as a memory barrier, to explicitly enforce consistency between the temporary view and memory whenever the programmer intends to do it for his specific purposes.

The OpenMP programming model has evolved over the years to support fine-grain and irregular parallelism that currently characterize a large part of modern parallel applications, which is achieved by making the concept of *task* central. Since version 3.0 of the specification [3], OpenMP has made task parallelism an accessible parallelization strategy along with already implemented strategies such as data parallelism and other static forms of functional parallelism. The programmer can now use simple #pragma clauses and an API supported by the runtime environment to schedule tasks, to define their dependency constraints and, as in more recent developments of the OpenMP specification (version 4.5), to devise priorities across tasks [62]. In Figure 4.9 we report a graphical, basic representation of the execution model that OpenMP specifies for the running of tasks. Threads generated to carry out the execution of a parallel region perform their operations in the context of an implicit task, that is the execution state of threads when running code belonging to the parallel region for which the schedule is statically known at compilation time. As soon as a thread encounters the construct for spawning a task—it is a call to the library routine devised to perform creation of tasks, which has been placed along the execution trace by the compiler at the point where the programmer had specified the relative directive—a new explicit task is generated for the associated structured block. New tasks can also be spawned by threads that are performing others, previously generated explicit tasks, in which case we are dealing with descendants of the latter—a hierarchical spawning tree determines the kinship degree between all tasks. Moreover, when a thread encounters the construct for spawning a new task, it may immediately execute that task or defer its execution. In the latter case, which is the most likely to occur unless strong *cut-off* policies have been employed by the programmer—these are implemented in the form of conditional or imperative directives, such as if and final clauses, used to enforce the immediate, nested execution of tasks within the context of their parents with the aim of containing an excessive

proliferation of tasks at run-time—the new task is inserted within a pool of tasks where it waits to be chosen for execution by any thread of the team. The latter normally occurs when one thread reaches a *task scheduling point* (TSP), where it is provided that the thread itself performs task scheduling activities for which the implementation may cause it to perform a task-switch, beginning or resuming execution of a different task bound to the current team. In this regard, a thread that encounters a TSP within the structured block (or function) associated to the task may temporarily suspend the execution of the involved region. By default, a task is created as a tied type of task, that is, its suspended region can only be resumed by the thread that started its execution. Tasks of this type are subject to a number of execution constraints, called *task scheduling constraints* (TSC), that were included in the specification since the first release of the OpenMP tasking model and that limit the schedulability of tied tasks along the execution path of threads that do not meet the conditions imposed by these rules. In particular, the scheduling of a new tied task is constrained by the set of task regions that are currently tied to the thread. If this set is empty, any new tied task can be scheduled. Otherwise, a new tied task may be scheduled only if it is a descendant of every task in the set. Conversely, the programmer may specify the untied clause along with the compiler directive devised to create the task so as to enable any thread in the team to begin its execution under any circumstances, or to resume it after a suspension as there are no particular constraints defined for the scheduling of this type of tasks.

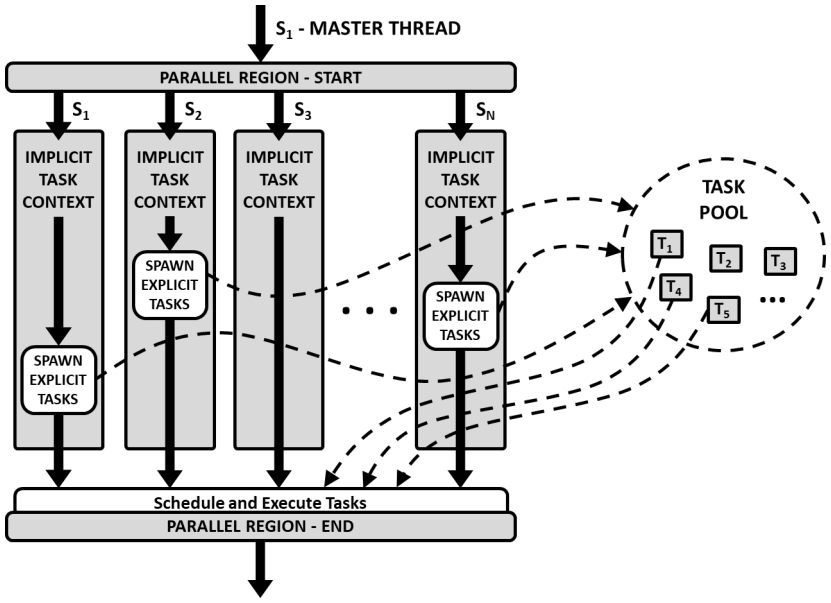

Figure 4.9. OpenMP task execution model.

Overall, the evolving nature of the OpenMP specification poses the need for a continuous evolution of its supporting runtime environments, which should reflect as much as possible the real nature standing behind the specification into the actual dynamics that OpenMP applications experience. As for the latter aspect, one of the main limitations of current implementations of OpenMP runtime environments—like the one offered by the GNU OpenMP (GOMP) project—stands in the impossibility

to asynchronously preempt a running task in order to assign the computing power offered by some CPU-core to another task. In fact, task-switches can only occur if the running task spontaneously yields the CPU-core. This may happen either upon task completion, or when the running task reaches a TSP, where an implicit call to the library routine devised, *e.g.*, to spawn a new task (task directive), to wait for the dependencies to be satisfied (taskwait directive), or to cede control to another task by the will of programmers (taskyield directive), returns control to the runtime environment.

A first drawback from this limitation is that we may observe scenarios where a new task scheduled with a higher priority is delayed by a lower priority one that has already gained control of the CPU-core. This is somehow adverse to reaching the objective that higher priorities should be directly translated by the runtime environment into better timeliness and more prompt CPU-dispatching of task execution. This problem is exacerbated by the fact that a task in OpenMP simply corresponds to the execution of a function specified by the programmer whose CPU demand can be arbitrary. Also, the function might even interact with blocking services of the underlying OS, so that its turnaround time, or the time to reach the TSP, can be further stretched. All these aspects can delay excessively the activation of some higher priority task that is already standing.

Another drawback that we observe with current implementations of OpenMP runtime environments is that joining a task with one it depends on (*e.g.*, via the taskwait directive), makes an heavy usage of blocking synchronization services like OS futexes. In particular, common OpenMP runtime environments, such as GOMP, lead a thread S that is currently running a task T_i to block when T_i needs to join the execution of a child task T_j that has already been CPU-dispatched along another concurrent thread S'. In other words, the take-off of task T_i with no actual block of thread S can take place only under the scenario where task T_j is not currently taken in charge by, or has been bound to, another thread. Limited to this scenario, thread S can immediately take in charge task T_j by temporarily suspending task T_i, hence without incurring in the blocking phase. In any other scenario, the above described problem is somehow critical in terms of the capability of the runtime environment to exploit the underlying hardware resources. In fact, having a thread to block because of a dependency constraint with a task that is the child of the one it has currently taken in charge, and that has been bound to another thread, leads in practice to renouncing to the computing power of a CPU-core. The presence of thread-blocking phases in situations where standing tasks could be ideally picked and processed according to TSCs makes the OpenMP runtime environment unable to guarantee the so called *work-conservativeness* property [30, 76]—an algorithm for scheduling on multiprocessors is defined to be work-conserving if it never leaves any thread idle while there remain active tasks awaiting execution. We want to make notice that, although it is known that TSCs and the semantics of tied tasks would have prevented in any case the implementation of work-conserving schedulers—such an issue is of high interest for the research community, since it leads OpenMP applications to be difficult to be formally analysed in terms of their capabilities to match deadlines in presence of tied tasks [84, 76, 83]—the same problem would have occurred even if there were pending untied tasks in the pool, whose adoption should instead allow to always make the schedulers work-conserving. This is further exacerbated by the fact

that, any previously suspended, not yet completed task cannot be resumed until all tasks that were subsequently started to run along the execution path of the same thread have finished their work, regardless of whether they are tied or untied.

All these issues are due to the fact that common OpenMP runtime environments have made the OS-thread central in the implementation of the execution model of tasks, that is, the execution context of all the tasks that subsequentially run and pause along the execution path of a single thread are in effect part of the context of that thread. This means therefore that the OS-thread centric nature of the technology employed to support the execution of OpenMP tasks makes these runtime environments unsuitable for dealing with more articulated forms of scheduling. This type of problem has been widely addressed in the recent years, and several lightweight thread (LWT) approaches have been presented in the literature as a valid alternative to the OS-threads to support the execution of tasks in many runtime environments, each one providing a specific high-level programming model (*e.g.*, OpenMP). This is the case of Converse Threads [43], MassiveThreads [60] and Argobots [75]. All these solutions are based on user level thread (ULT) technology, which helps these LWT libraries in offering lighter mechanisms to tackle massive concurrency, thus avoiding to pay the overhead costs that would have been introduced with the use of conventional OS threading mechanisms (*e.g.*, Posix Threads). By the way, ULTs are migratable, yieldable, and suspendable units of work, each one provided with its own stack, whose management and dynamic scheduling takes place without the participation of the OS. Additionally, Argobots also integrates deferred-work concepts, such as Tasklets, which have been historically used in OS kernel technology to finalize some previous request for task execution via an explicit synchronous call to a tasklet-processing routine. Similarly, the Converse Threads solution integrates the notion of Messages, as a form of deferred-work to be processed after that a poll operation performed by a thread identifies the presence of an incoming message.

Nevertheless, even though the above described solutions have been designed to provide more flexible parallelization and dynamic scheduling of tasks—along with many other state-of-the-art proposals that we have not presented for relevance reasons—they all share a common limitation, that is, CPU-dispatching of the execution of tasks can only occur as a result of a cooperative form of scheduling as any execution flow variation can only take place at specific execution points via explicit synchronous invocations of an API that switches control between ULTs, each one encapsulating a task to be executed. This means therefore that employing such LWT libraries to support high-level programming models, again including OpenMP, is definitely not a suitable approach to efficiently handle the execution of tasks with assigned priority values, which immediately brings us back to the first drawback we discussed earlier. Conversely, it would be desirable to have a solution based on both asynchronous and synchronous switches of the execution flow of tasks used in combination, which would make the runtime environment capable to promptly dispatch on CPU the execution of higher priority tasks as soon as the latter are spawned within the system.

As we already mentioned in the previous chapters, asynchronous switch of the execution flow is a classical target for OS technology, for what it concerns both time-sharing and processing of signals/events. However, the granularity according

to which a conventional OS induces control flow variations is unsuited for making a thread extremely reactive to the need for changing its execution flow. Just to make an example, the implementation of an asynchronous switch bringing a thread to run a higher priority task via Posix signals would make the activation of this task delayed up to the end of the current tick period assigned to the thread by the OS. In fact, the delivery of signals, with the associated control flow variation, mostly occurs when return to user mode, that is, after receiving a timer interrupt or returning from a system call. We want to recall that classical Linux configurations are based on a tick ranging from 1 to 4 milliseconds depending on the parallelism degree of the hardware, which would cause a delay in shifting one thread to execute a higher priority OpenMP task that is clearly inadequate, especially when the task is fine-grain and its waiting time for taking control under these settings can definitely overstep its running time.

This problem is directly tackled by the solution we had appositely devised to cope with the aforementioned drawbacks, which relies on the ULMT technology in order to accomplish the commissioning of the micro-threading model as a support for the preemptive execution of OpenMP tasks. This technology, along with the support given by the IBS hardware facilities, allows asynchronous execution switches along a thread with very fine granularity (*i.e.*, of the order of tens of microseconds) and minimal run-time overhead, without requiring the intervention by the programmer. In this regard, we have designed and implemented extensions to the GOMP runtime environment for Linux and `x86_64` processors which address all the above raised problems. Our proposal is indeed based on the integration of functionalities offered by the micro-threading library that are modularly combined with those already offered by GOMP. Also, all features characteristic of the ULMT technology can be enabled or disabled depending on the will of the programmer at the time at which an already compiled OpenMP program is launched, by specifying the value of an OpenMP environment variable that has been appositely devised to control such behaviour by initializing an internal control variable (ICV) named `ulmt_var`—ICVs are defined by the OpenMP specification as a set of variables that control the behaviour of an OpenMP program, whose name and their number are implementation-specific with no particular constraints except for few predefined ones. In the end, our software modules allow:

- prompt switch to any higher priority OpenMP task that is scheduled while a thread is processing a lower priority one, thus providing better execution timeliness of the former;
- the avoiding of thread blocking phases that arose, in the native version of GOMP runtime, upon reaching task execution barriers imposed by dependencies defined on other tasks, which had been associated in the meantime with different threads.

The above two objectives are met in our design while still guaranteeing all the properties that are demanded from OpenMP runtime environments, such as the avoidance of moving `tied` tasks across threads and the avoidance of TSCs violation, which ultimately make a program to be *conforming* with the specifications—a program is said to be conforming whenever it follows all rules and restrictions of the

OpenMP specification. Our idea for reaching these two core objectives, namely fine grain control of task priorities and better satisfaction of work-conservativeness, is based on exploiting the notion of Task-Context (TC), which is essentially a CPU context that at any time instant can be either running on some CPU-core along the execution path of a thread, or it can be saved into a gomp_task_state data structure. TCs might appear to have resemblances with contexts dealt with by those LWT libraries implementing ULTs. However, we cannot base our design on reusing ULT implementations because, as hinted before, they cannot support asynchronous passage of control between, for instance, a context TC_i and another context TC_j. In fact, ULTs allow switching between contexts, hence between different execution flows along a thread, only under the assumption that all the switches take place via synchronous calls to a switch-supporting API (*e.g.*, setjmp/longjmp). Differently, a TC is in effect a micro-thread context wrapped by the gomp_task_state data structure, which will be assigned to a task via a pointer included in the gomp_task data structure upon its activation—it is the data structure that a thread instantiates once it has invoked the GOMP_task function provided by the GOMP library for spawning tasks, and which is dedicated to keeping all the information about the involved task (*e.g.*, task type, task priority, function, arguments, parent task, child tasks, dependencies, etc.) that is needed to manage it according to the OpenMP tasking model—and which will be finally released upon its completion.

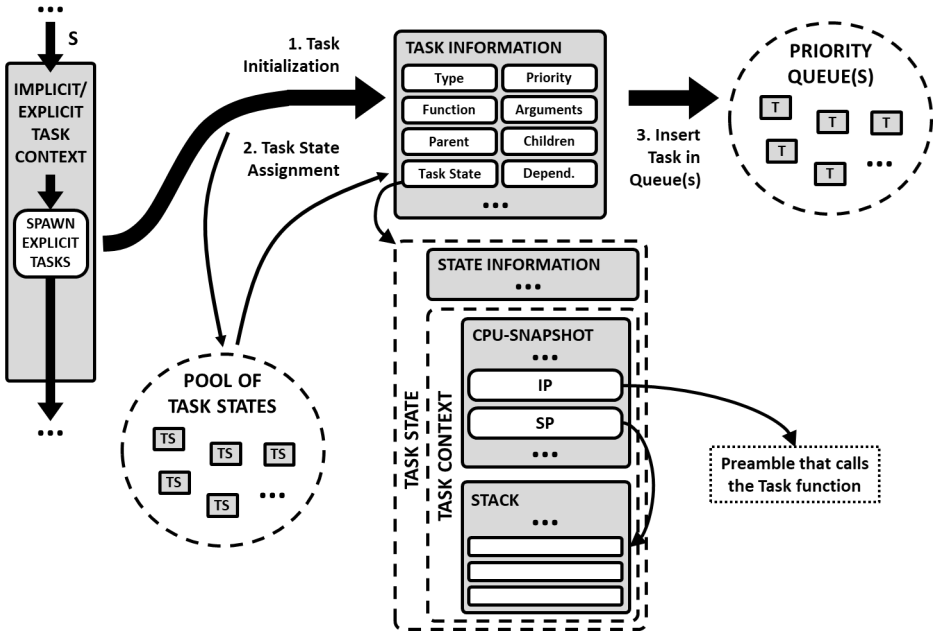

Figure 4.10. Task initialization in the ULMT-based version of GOMP.

In Figure 4.10 we show a graphical representation of the initialization of OpenMP preemptive tasks. Here, a dynamic pool of gomp_task_state data structures is queried in order to retrieve a micro-thread context to associate with the task being created. Such a pool is automatically generated by the runtime while it is making the setup of a parallel region (*i.e.*, gomp_team_start function), by invoking the

gomp_state_pool_init function just prior to instantiate the team of threads. This function is part of a group of functionalities that we have intentionally designed to allocate and operate memory management activities related to the dynamic pool of gomp_task_state data structures. More in detail, it is made up of as many private local pools as there are threads in the team plus a shared global one, in a way that resembles the implementation design of the Hoard memory allocator [4], but with the difference that our solution is easier in that it has only to deal with a single type of fixed-size data structure. Nonetheless, it shares with Hoard the same logic adopted to rebalance free memory across all local pools as soon as one of the latter is discovered to hold an amount of memory that exceeds a dynamically computed threshold, by making this solution scalable and suitable for managing a set of micro-thread contexts—which possibly grows in size—in OpenMP applications where the thread that spawns a task is unlikely to be the one that completes it. We want to make notice that, in our ULMT-based solution of the runtime, the allocation of memory needed to accommodate the gomp_task data structure takes place in the same way as it is performed in the native version of GOMP, that is by calling the malloc function upon every new task activation, so as not to give rise to favourable performance imbalances due to customized optimizations in the code.

Once the team has been created, and before threads reach the barrier preceding the work-sharing region characterizing the execution of each implicit task, we check for the value of an internal control variable (ICV) named ibs_var to allow threads to register themselves for exploiting the IBS hardware facility. Registering will allow them to receive periodical interrupts that will be handled by the Linux kernel module supporting task preemption. When one of these interrupts is received, control is bounced to an early-logic represented by the CFV_TRAMPOLINE routine that we have presented in Chapter 3. It is part of a set of runtime facilities belonging to the micro-threading library that, along with the support given by the kernel module, completes the control flow variation of interrupted threads. In particular, this routine checks specific variables to verify if the task currently executed by this thread may effectively be preempted in favour of another one, and if so it passes control to the ULMT-level task scheduler. Passing through this function, our OpenMP runtime environment allows the thread running the OpenMP application to execute the (asynchronously triggered) task-to-CPU reassignment algorithm aimed at the effective management of the priorities of standing tasks. A timeline schematizing the evolution of a thread running the OpenMP application is shown in Figure 4.11. Initially the thread might be running within the TC_i context. As soon as one of these interrupts occurs, control is passed to the trampoline routine and the execution state of the currently running task is saved into the gomp_task_state data structure that has been associated with TC_i for its whole lifetime. Then, after the task-to-CPU reassignment algorithm has been performed too, the thread decides to switch to the context TC_j of another task, possibly with a higher priority value, thus temporarily suspending the execution of the task with context TC_i. We want to point out that, the asynchronously interrupted thread does only a non-blocking attempt to acquire the GOMP internal mutex before it is allowed to execute the task-to-CPU reassignment—it is a synchronization object used to ensure mutual exclusion between thread accesses on GOMP's shared data structures—which is actually executed only if the mutex try-lock succeeds. This strategy makes interrupted threads not hang

whenever another thread is already executing GOMP activities related to shared data structures. Hence, it provides minimal intrusiveness in terms of wait-access to the mutex-protected critical section by simultaneously interrupted threads.

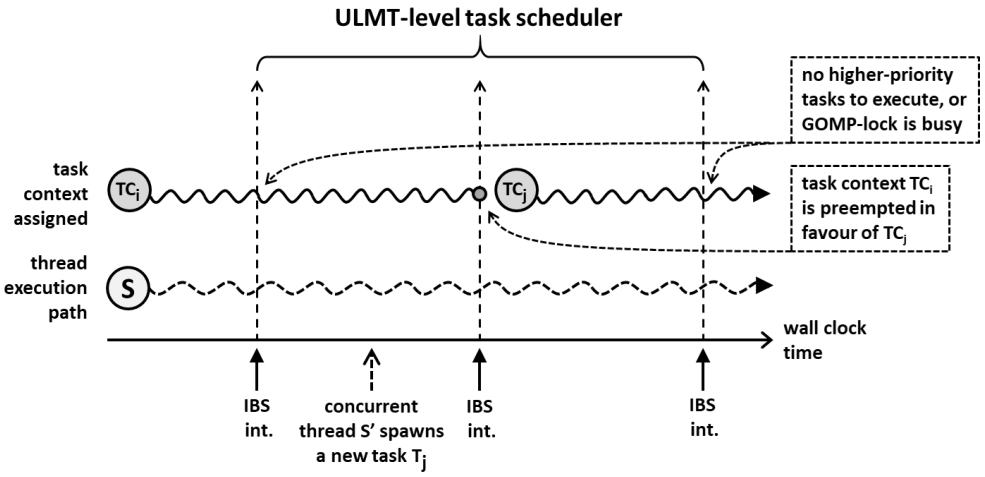

Figure 4.11. Timeline with asynchronous preemption of a task.

Anyhow, the passage of control in our re-engineered environment is not only due to the occurrence of asynchronous events. Rather, the environment also embeds a form of synchronous invocation of the ULMT-level task scheduler. More in detail, while executing a task, a thread may encounter several function calls that match different kind of OpenMP directives—executable and stand-alone directives to be more precise—corresponding to the invocations of the associated library routines. To these function calls also correspond the TSPs at which the runtime environment may or may not renew the scheduling decisions, namely the assignment of tasks to threads. Nevertheless, the execution of some of these functions in GOMP can lead to block the running thread—via calls to system futexes from which it will later wake up upon direct signaling by concurrent threads—because of the need for respecting the dependency constraints that may have been defined among tasks by the programmer whenever they are not found to be directly satisfied upon reaching the TSP. This is the scenario where a task T_i executed along thread S needs to wait for the finalization of a tied task T_j which has already been activated for processing, possibly suspended shortly after, along some other thread S'. In this case, thread S can no longer take care of passing control to T_j at the TSP to resolve the dependency, since this task has been definitively tied to thread S'. We want to recall that, when running in the native GOMP runtime environment, the same scenario would have occurred even if task T_j were of the untied type, since GOMP runtime does not differentiate between tied and untied types when managing tasks. Also, again due to the OS-thread centric technology behind GOMP implementation, the runtime prevents the thread S from starting (or resuming) the execution of another standing task T_k—even when it does not violate TSCs—because the resume of task T_i once its dependencies have been resolved would require the completion of all the other tasks that may have subsequently been taken over by thread S, thus introducing excessive latency for the completion of task T_i with negative effect on its turnaround

time. This is especially bad when, with the aim of performing work-conserving, task T_k is a low priority one. By the way, we are referring to the taskwait directive which finds its implementation in the GOMP_taskwait function and which is designed to prevent a task from continuing to run until all its child tasks have completed, or the gomp_task_maybe_wait_for_dependencies function which prevents the execution of non-deferrable child tasks as long as there are dependencies on their siblings that have not yet finished their work. In our ULMT-based re-engineered environment we avoid this problem since we enable the passage of control to another task via a synchronous invocation to the ULMT-level task scheduler upon reaching a TSP if the associated constraints are not currently satisfied, which is again achieved in a fully transparent manner with respect to the OpenMP applications.

It should also be noted that the above described scenario would have led to experience non-optimal execution dynamics even if a LWT approach, such as the ones we discussed before, were employed to support the execution of OpenMP tasks by mean of ULTs. In fact, such solution cannot rely on any form of asynchronous task-switch along the execution path of threads, hence it would have led to a scenario where the possibly high priority task T_i (tied or untied)—which is paused by thread S that then moves to perform a different task—is left suspended even after all of its dependencies have already been satisfied, which is ultimately adverse to the objective of efficiently handling task priorities. It is therefore clear that, employing the ULMT technology as a support for the execution of OpenMP tasks, via micro-threads, is the only approach that can enable threads to perform work-conserving (where TSCs permit), while still guaranteeing strict time bounds within which the system reacts to the materialization of higher priority tasks to execute, which can suddenly occur due to both the creation of new tasks that are inserted into the system and the resolution of dependencies defined on suspended ones that are now ready to be resumed.

As a final note, GOMP also supports the OpenMP critical directive that maps to the GOMP_critical_start function. This function allows tasks to rely on a global (or named) lock for executing critical sections. In the ULMT-based implementation of GOMP runtime, we have transparently wrapped this function to let it perform a try-lock operation in place of a blocking one, whose failure will lead the wrapper to invoke the ULMT-level task scheduler so as to switch off the CPU the task that needs access to the busy lock. Similarly, we wrapped those functions associated with the set of general-purpose locking routines that rely on the omp_lock_t structure for synchronization purposes. This again enables the execution to slide towards work-conservativeness by enabling a thread, which would otherwise be blocked while running a task that needs to access a busy critical section, to take care of processing other standing tasks.

However, we have not yet presented all the mechanisms at the base of the ULMT-level task scheduler logic. As a first important note, while the native version of GOMP runtime only manages a single queue of standing tasks which are runnable, in the ULMT-based implementation we have introduced multiple queues because of the need for correctly satisfying OpenMP constraints in the more sophisticate task management environment entailing differentiated contexts for the different tasks. As noted, the latter aspect is absent in the original implementation of GOMP since the arrival to a TSP never leads to switch to a different context. In fact, even

under the scenario where the task reaching the TSP has unsatisfied dependency constraints that leads the thread to run another task (*i.e.*, a child task), this happen via a synchronous invocation of the function associated with this task by the thread, which by convention performs the task by relying on a new frame built on top of the same stack. Overall, the list of the task queues exploited in the ULMT-based solution is the following one:

- GRQ (Global-Runnable-Queue) which is the original GOMP global queue. It contains all the tasks that can be run by any thread. When a new tied/untied task is created it is inserted into this queue;
- GBQ (Global-Blocked-Queue) which is a new ULMT global queue that keeps all the currently blocked untied tasks (their contexts have been descheduled by the ULMT-level task scheduler).
- LRQ (Local-Runnable-Queue) which is a new ULMT queue, with an instance per-thread, that keeps all the tied tasks that are currently runnable—possibly after a block phase—and must be finalized by the specific thread given that they have been originally picked from the GRQ by that thread.
- LBQ (Local-Blocked-Queue) which is a new ULMT queue, with an instance per-thread, that keeps all the currently blocked tied tasks, which must be finalized by the specific thread along which they have started their execution.

GRQ and LRQ are implemented as splay trees as it is in the original implementation of GOMP, so as to associate a different node to a sublist of tasks with a given priority. GBQ and LBQ have been implemented as simple doubly linked lists, allowing constant time removal of an element once we have the pointer to the gomp_task structure representing the task to be removed from the queue. GBQ and LBQ keep tasks that are blocked because their dependency constraints are not currently met and those that are blocked since they need to access some busy resource (*e.g.*, a busy lock).

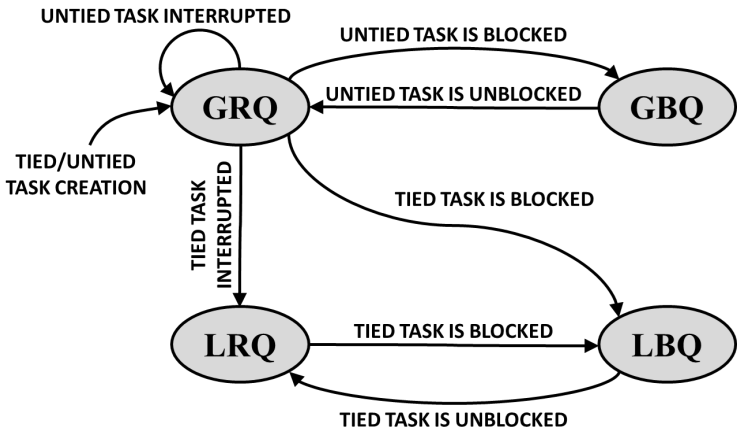

Figure 4.12. Task-state diagram in the ULMT-based version of GOMP.

The moving of tasks among the queues, caused by the IBS interrupt (which is asynchronous with respect to the execution of the task) or by the invocation of a TSP that leads to block the involved task (which is instead synchronous), is depicted in Figure 4.12. A task kept in GRQ can be moved to LRQ if it is of tied type, it has been picked by a thread to execute, and it is then interrupted because of the passage of control to some higher priority task. It will run again only along the execution path of the thread that has initially picked it when it will become the highest priority task observed among GRQ and LRQ (and among those that do not violate the TSCs) by the thread upon running the ULMT-level task scheduler. The transit towards GBQ is only admitted for untied tasks that are CPU-dispatched along a thread and then trap into a TSP with the associated dependency constraints not yet satisfied (or try to access some busy lock). Similarly, tied tasks that are eventually CPU-dispatched either from GRQ or LRQ are then always put into LBQ when reaching a TSP and the associated dependency constraints are not yet satisfied (or an access to a busy lock is attempted). Overall, by the state diagram we have that a tied task is never allowed to bounce back to GRQ or GBQ once picked by a given thread, thus always residing in the per-thread queues along its lifetime. Clearly, when a task is moved back to GRQ from GBQ, or to LRQ from LBQ, it is inserted at the priority level specified upon task creation. However, given that in the original implementation of GOMP such migration of tasks across queues had not been put in place, a decision must be taken on the actual position to select for queuing the task into the corresponding priority level. As for this point, we decided to adopt different queuing policies of unblocked tasks into GRQ and LRQ. For LRQ we adopted a classical tail insertion. On the contrary, for GRQ we decide to take the opposite choice of queuing it at the head. In other words, within the same priority class of GRQ, a task being currently unblocked will be CPU-dispatched before other runnable ones already standing at the same priority level. The motivation for this choice is illustrated with the aid of Figure 4.13. Specifically the GRQ may contain both not yet CPU-dispatched tasks—by the task-state diagram, any task is inserted into GRQ upon its creation—and untied tasks that have already been dispatched at some point in the past and then were either interrupted in favour of some higher priority task, or blocked upon meeting non-satisfied dependency constraints at some TSP (or attempting the access to some busy lock) before it is moved back to GRQ as soon as the constraints became satisfied (or the needed lock has been released). Tasks that were not yet CPU-dispatched can be seen as *cold* ones (highlighted with the blue colour) with the meaning that no resources (memory buffers I/O channels, etc.) were yet committed for their execution. Instead, the tasks that were already CPU-dispatched at some point in time before being suspended due to one of the abovementioned reasons can be seen as *hot* ones (highlighted with the red colour) given the commitment of some resources for their partial execution. The insertion of unblocked tasks at the head of the per-priority level in GRQ when moving them back from GBQ can favour more prompt execution of hot tasks, with the advantage of more timely release of the corresponding committed resources, which in turn can favour aspects such as locality.

An additional optimization we integrated within the ULMT-level task scheduler applies to both GRQ and each individual LRQ associated with the different threads. It is based on the idea of further favouring, within a same priority level, the task

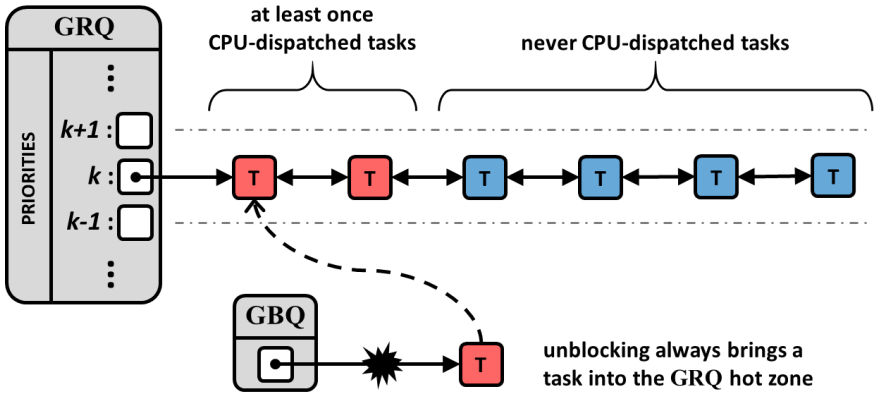

Figure 4.13. Hot and cold task zones into GRQ.

that more recently used the CPU. This can additionally favour aspects such as locality, especially in scenarios with fine-grain activities where the tasks that used a CPU-core after some others were context switched off the CPU may have only partially invalidated cached data that will be accessed by those same tasks when CPU-dispatched again. We note that this optimization can be relevant even in scenarios where a task is untied and is moved across different threads along its lifetime as these threads may run on top of CPU-cores that share lower level caches—the same argument applies to top level caches for, *e.g.*, hyper-threaded CPUs. It is accomplished by keeping for each task T_k a reference (a pointer) to the task T_j that released the CPU in favour of T_k. Overall, each task keeps a reference to the one that was running immediately before its CPU-dispatch operation. When the ULMT-level task scheduler is invoked by some thread, the latter identifies the GRQ or LRQ sublist with highest priority having runnable tasks and, instead of passing control to the task placed at its head, it checks whether the task referred as the last running one (T_j in our example) is on the same priority level and is now runnable. Clearly, if such last running task T_j is currently blocked, we can exploit the reference to a possible last running task T_i seen by T_j, thus favouring T_i if it is runnable and stands at the highest priority level. Such an approach can iterate, but we can anyhow impose a bound on the number of iterations—falling back to CPU-dispatching the task at the head of the non-empty sublist with the highest priority—to make the schedule operation executable in constant time. We also note that all the suspended tasks to which some others refer as their last running ones are tasks that have been CPU-dispatched at least once, and for this reason they are hot tasks. This means that the optimization we have talked about so far also prevents that a cold task kept by the GRQ is CPU-dispatched before some other hot task that possibly exists at the same priority level. This turns out to be very useful in scenarios where a task T_i loses control of the CPU because of an IBS interrupt that leads to pass control to the higher priority task T_j, so that when T_j completes and the ULMT-level task scheduler is invoked again, if T_i stands to the highest non-empty priority level together with other runnable tasks then it is favoured over all the other tasks, thus hopefully finding again cached data to exploit along its execution.

To assess the proposed ULMT-based OpenMP runtime environment we carried out experiments on top of a HP ProLiant multi-core machine equipped with two 2.2GHz AMD Opteron 6174 processors and 32GB of RAM (4 NUMA nodes). Each processor has its own 12 physical cores, for a total of 24 CPU-cores in the system. Also, the operating system is Debian 9 with Linux kernel version 4.9.

In our experiments we assessed the behaviour of the re-engineered runtime comparing performance indexes' values with the corresponding ones achieved by running the native version of GOMP, which we will refer to as Baseline runtime from now on. The comparison is carried out while scaling up the number of threads up to the maximum number of physical CPU-cores in the underlying machine. Also we prevent the runtime system from employing *task throttling* heuristics—these are techniques adopted to serialize the execution of tasks along threads when specific thresholds on the number of standing tasks are exceeded—since they have been proved to be harmful [26] for several application classes. However, we allow the applications to control the overhead for task creation and management by relying on the *cut-off* policies provided by the benchmarks themselves. We want to point out that the proposed ULMT-based runtime perfectly deals with the management of tasks grouped under a single task context—as it occurs with *undeferred* and *included* tasks (*i.e.*, as a result of the activation of the if and final clauses)—which the runtime dispatches along threads by taking care to respect the most stringent constraints among those of all the grouped tasks.

As for the benchmarks, we used various applications from the Barcelona OpenMP Task Suite (BOTS) [16], which is a suite that has been devised to bypass the limitations of previous OpenMP benchmarks in terms of their capability to generate irregular and fine grain workloads with task dependencies. Nevertheless, one limitation of BOTS, which has not yet been resolved even by more recent proposals like KASTORS [85], is the lack of the definition of OpenMP-suited applications specifically devised to stress the runtime system in presence of task priorities, whose cause is related to the recent inclusion of priorities in the OpenMP specification. Given that the effective management of task priorities is the central target for our innovative ULMT-based proposal, we designed and implemented a new open-source benchmark application, named HASHTAG-TEXT, entailing the possibility to configure different task priority levels.

Anyhow, we still used applications from BOTS to assess the overhead produced by our re-engineered GOMP runtime, as well as the benefits that can be obtained thanks to its orientation to work-conservativeness. By the way, given that they do not entail differentiated priority levels, there is no need to exploit IBS interrupts for promptly passing control to higher priority tasks. However, the management of differentiated contexts performed by the ULMT-based runtime is still useful to block/unblock them depending on their dependency constraints. Therefore, the focus of this study is on the assessment of the trade-off between the overhead for separate contexts management and the advantage that comes from a superior ability of the ULMT-based runtime to produce more articulated schedules of tasks along threads. In Table 4.2 we report the set of applications we selected from BOTS together with a few of their most relevant characteristics, for which details we refer the reader to the original specification of the suite of benchmarks. As it is possible to see, with the exception of Alignment and SparseLU in its configuration

Table 4.2. Applications selected from BOTS.

Name	Description	Domain	Nested tasks	Cut-Off	Work-sharing	Taskwait
Alignment	Protein alignment with the Myers and Miller algorithm [59]	Dynamic programming	no	none	for	no
					single	
Floorplan	Computation of the minimum area size including all cells	Optimization	yes	none	single	yes
				depth-based		
SparseLU	LU matrix factorization over sparse matrices	Sparse linear algebra	no	none	for	no
			yes		single	yes
Strassen	Hierarchical decomposition for multiplication of large matrices [22]	Dense linear algebra	yes	none	single	yes
				depth-based		

based on the *for* work-sharing construct, the selected applications have *nested tasks* spawned at run-time which synchronize with their parent tasks by mean of taskwait directives. Hence, they can represent good test cases to determine whether our design can provide advantages through the avoidance of thread blocks upon reaching a taskwait, which is where the orientation to work-conservativeness stands. On the other hand, all the selected benchmarks, except for Floorplan and SparseLU in its configuration based on the *single* work-sharing construct, are characterized by an embarrassing parallelism as no thread ever blocks because of task dependency constraints, and for this reason have no particular scalability issues. Therefore, they are useful especially for the determination of the overhead of our proposal in a scenario where work-conservativeness is not a real concern. Also, they cover a set of important different domains, ranging from dynamic programming to optimization, passing through both sparse and dense linear algebra applications. In Table 4.3 are reported the execution times obtained by the experiments performed with the above described applications—computed as the average over 8 runs—whose values represent the number of clock cycles elapsed between the beginning and the end of each application run. Also, we report the results of experiments carried out with both the Baseline and ULMT-based GOMP runtime, providing relative speed-up values (either positive or negative) of the latter with respect to the former.

As for the Alignment, we observe almost negligible positive speed-up values by the ULMT-based runtime. This because the benchmark does not provide for the spawning of nested tasks throughout its execution, thus it does not need to rely on the taskwait construct to accomplish synchronization between parent and child tasks. In this scenario, managing separate task contexts to prevent threads from blocking due to task dependency constraints does not offer advantages. However, the overhead introduced by the ULMT-based runtime is essentially negligible, also motived by the fact that around the 99% of the execution time is spent running

Table 4.3. Results with the applications from BOTS.

Version		8 Threads		16 Threads		24 Threads	
		Execution-Time (x10^6)	Speed-Up	Execution-Time (x10^6)	Speed-Up	Execution-Time (x10^6)	Speed-Up
Alignment							
for-untied	Baseline	18.936		9.752		6.627	
	ULMT	18.898	+0,20%	9.730	+0,23%	6.598	+0,43%
for-tied	Baseline	18.926		9.576		6.267	
	ULMT	19.030	-0,55%	9.601	-0,25%	6.447	-2,88%
single-untied	Baseline	18.980		9.769		6.669	
	ULMT	19.022	-0,22%	9.746	+0,24%	6.616	+0,80%
single-tied	Baseline	18.899		9.523		6.190	
	ULMT	18.949	-0,27%	9.589	-0,69%	6.337	-2,37%
Floorplan							
untied	Baseline	153.534		239.963		330.681	
	ULMT	245.655	-60,00%	420.417	-75,20%	558.558	-68,91%
untied-if	Baseline	5.956		3.145		2.020	
	ULMT	4.716	+20,81%	2.633	+16,28%	1.912	+5,33%
untied-manual	Baseline	3.359		1.781		1.079	
	ULMT	2.642	+21,34%	1.406	+21,07%	904	+16,28%
tied	Baseline	185.456		386.984		616.214	
	ULMT	288.368	-55,49%	600.359	-55,14%	893.358	-44,98%
SparseLU							
for-untied	Baseline	172.677		90.696		63.972	
	ULMT	170.640	+1,18%	90.135	+0,62%	63.429	+0,85%
for-tied	Baseline	172.752		90.352		63.785	
	ULMT	170.714	+1,18%	89.631	+0,80%	63.177	+0,95%
single-untied	Baseline	167.308		86.180		59.281	
	ULMT	160.393	+4,13%	82.281	+4,52%	57.034	+3,79%
single-tied	Baseline	167.064		86.195		59.216	
	ULMT	160.308	+4,04%	82.210	+4,62%	56.576	+4,46%
Strassen							
untied	Baseline	33.637		19.656		14.046	
	ULMT	38.888	-15,61%	22.908	-16,55%	18.090	-28,79%
untied-if	Baseline	35.585		22.285		17.819	
	ULMT	34.587	+2,80%	21.312	+4,37%	16.602	+6,83%
untied-manual	Baseline	35.661		22.220		18.246	
	ULMT	34.514	+3,22%	21.078	+5,14%	16.752	+8,19%
tied	Baseline	33.246		19.163		14.243	
	ULMT	32.796	+1,35%	18.985	+0,93%	13.638	+4,25%

code not included in the GOMP library, a phenomenon linked to the relatively large granularity of tasks in Alignment, which is about 14 milliseconds on the used computing system. Overall, the relative cost for the creation of task contexts and the management of task-switch operations introduced by the use of ULMT technology is irrelevant with this benchmark execution profile.

As for the Floorplan, the granularity of its tasks is much finer and varies depending on whether a cut-off policy has been implemented or not—here, tasks have a granularity of about 550 nanoseconds when no cut-off is employed, in contrast

to 480 microseconds observed when a manually implemented cut-off strategy was present in the code or when determined if clauses were added to the task directive. This leads to the scenario that is opposite to that of ALIGNMENT, where around 98% of time is spent executing code of the GOMP library, rather than code of the overlying OpenMP application, whenever no cut-off policy was implemented. By the data, we note that tied and untied versions of this benchmark without any cut-off applied produce results, in terms of execution time, which are two order of magnitude worse than those achieved with cut-off policies implemented. Also, the execution times of experiments performed without cut-off do not scale with the number of threads with both the Baseline runtime and the ULMT-based one. Looking at data, the ULMT-based runtime gives rise to large negative speed-up, compared to the Baseline one, only under this pathological setting—this is due to the additional task management operations—which is essentially representative of an application level misconfiguration that gives rise to trashing phenomena while managing tasks. Differently, when running with adequate cut-off strategies applied in the code, hence with the correct settings for this application profile, the ULMT-based runtime leads to obtain positive speed-ups that range from 5.33% to 21.34%. Since FLOORPLAN provides for the intensive spawning of nested tasks throughout its execution, we observed in the experiments carried out with the ULMT-based runtime thousands of task-switches occurring upon the execution of code associated with the taskwait construct, which confirms the benefits achievable when forcing the OpenMP scheduler to behave work-conservatively whenever possible. This result is further relevant when considering the very fine granularity of FLOORPLAN tasks.

As for the SPARSELU, we can group the results in two sets. In the first one, we find results obtained through the execution of those versions that use the for work-sharing construct to immediately distribute all work across threads. These experiments are characterized by an embarrassingly parallel execution scheme where no thread ever interferes with the work of others, nor they need to synchronize because of dependency constraints defined between parent and child tasks since no taskwait directive is provided in the code. Conversely, in the second group we find results obtained through the execution of those versions that rely on the single work-sharing construct to generate all those first tasks that will subsequently spawn the nested ones, which synchronize with their parents by mean of taskwait directives. However, we observed by the experiments that, upon the execution of the taskwait construct, the threads have mostly found either locally spawned tasks available to be taken in charge (*i.e.*, children of the one currently performed) or the dependency constraints already met, which are favourable cases to the Baseline runtime. All these reasons explain the low, although positive, speed-up values achieved by the ULMT-based runtime with these versions. In any case no penalty is noted for the additional work by ULMT-based runtime related to managing separate task contexts.

As for the STRASSEN, we note that the ULMT-based runtime shows positive speed-up, up to slightly more than 8%, in all configurations except the one with untied tasks and no cut-off policies implemented, where it shows an important negative speed-up. The motivations are twofold. First, the absence of any cut-off leads, as hinted, to more pressure on the handling (creation and scheduling) of tasks, which with ULMT technology is more costly because of the separation of task contexts. Second, and more important, the ULMT-based runtime has an extra

overhead paid whenever the pick of a different task to perform requires the suspension of the currently executed untied one, which also needs to be inserted again into the appropriate global queue other than merely saving its context. However, with well configured application settings, and proper usage of cut-off strategies, the overhead caused by the management of task-switches in the ULMT-based runtime pays off because of the achievement of better usage of CPU-cores thanks to its orientation to work-conservativeness.

By all these experiments performed with the applications from BOTS we were able to empirically observe the overhead generated by the use of ULMT technology to support the execution of OpenMP tasks for different application classes, which was found to be negligible in almost all scenarios deemed adverse for the ULMT-based runtime. Also, we have noticed how the performance of applications characterized by very irregular parallelism—those that do not lead to experience an embarrassingly form of parallelism—are more favoured when these applications run in the ULMT-based GOMP runtime environment with respect to the Baseline one, by obtaining also important positive speed-ups on the execution times.

However, we have not yet been able to quantify the benefits that the execution of OpenMP tasks with the support of ULMT technology can provide in terms of efficient management of task priorities, which was the main objective of this work. In this regard, we have designed and implemented the HASHTAG-TEXT benchmark which, as already pointed out, is fully complementary to literature benchmarks since it makes use of OpenMP task priorities. The HASHTAG-TEXT benchmark has been conceived to handle a huge dataset of <hashtag,text> pairs to serve requests coming from users with different priority levels, who wish to retrieve all texts associated with a given hashtag passed in input. This is the classical case of access to posts in scenarios like social networking. The benchmark uses a collection of real *tweets*, taken from an open-dataset containing Twitter messages, to populate the structures allocated at the service startup. In more detail, the service relies on one or more hash-tables to handle partitions to uniformly distributed data extracted from the same dataset, each one having a fixed but configurable number of buckets where colliding <hashtag,text> pairs are hung in a list. Since no data is ever replicated across different hash-table instances, relying on the functional parallelism offered by the OpenMP-tasking model appears to be a good choice for accomplishing the application objectives. In fact, it is possible to search for texts associated with a given hashtag in parallel across the different hash-tables.

To go into more detail about the features of this benchmark, upon arrival of requests at the service, with a given priority ℓ, the service itself takes care of mapping these requests to OpenMP tasks by assigning them to the priority level $(2 \cdot \ell)$. This is done in such a way as to preserve the priority order of requests even among the tasks to which they have been mapped, but leaving room for a priority level between any two subsequent request priorities that will be used to accommodate their child tasks in charge of searching for texts within the different hash-tables. These latter tasks are then assigned to the priority level $(2 \cdot \ell) + 1$. By the way, the tasks to which the requests have been mapped upon arrival at the service do not perform any particular computation but spawning as many research tasks as there are hash-table instances before synchronizing with them by mean of the taskwait directive. For a better understanding, in Figure 4.14 we report the directed acyclic graph of tasks in

the HASHTAG-TEXT benchmark, which has been built according to the sporadic DAG model considering the TSPs as shown in [84]. In this graphic is shown a task τ_i composed of several task region parts, each one corresponding to either a TSP or a task-specific block of operations that never branches in parallel. This is the OpenMP task to which a request has been mapped upon its delivery to the service. As soon as this task gets activated along the execution path of some thread, it spawns a number of child tasks τ_{ij}—this occurs upon the execution of the task parts labelled $p_{i,j}$ which correspond to the reaching of `task` directives—in charge of searching for texts in parallel, which then synchronize with their parent task by mean of the `taskwait` directive corresponding to the task part labelled $p_{i,n+1}$.

In our experiments, the application has been configured to generate requests with three different priority levels, namely 1, 2, and 3, according to a mixture of 60%, 30% and 10% respectively. This mimics a scenario with a majority of *normal* users, a good percentage of *silver* users, and a minority of *gold* ones. Furthermore, since the arrival of requests in the HASHTAG-TEXT benchmark has been designed to follow a Poisson process with configurable average arrival rate λ, we decided to set the corresponding average interarrival time period $1/\lambda$ to four different values in our experiments, namely 0, 10, 50 and 100 microseconds, which give us the possibility to observe the outcomes of the service execution with different workload levels. Finally, we have set to 4 the number of hash-tables keeping the data partitions, each one provided with 100 buckets of collision lists. We have chosen these values since we observed that they provide good load balancing even with the lowest workload level, while they do not impair parallelism with higher arrival rates. Despite the HASHTAG-TEXT benchmark is designed to work correctly with both `tied` and `untied` tasks, we decreed the use of `untied` ones applies better to the actual semantic of the application. Additionally, they stress more the ULMT-based runtime because of the expensive management required to handle this kind of tasks—this is due to the need for concurrent accesses by threads to the global task queue. As for the periodic IBS interrupt, we have selected three different periods, which are 100, 50 and 25 microseconds.

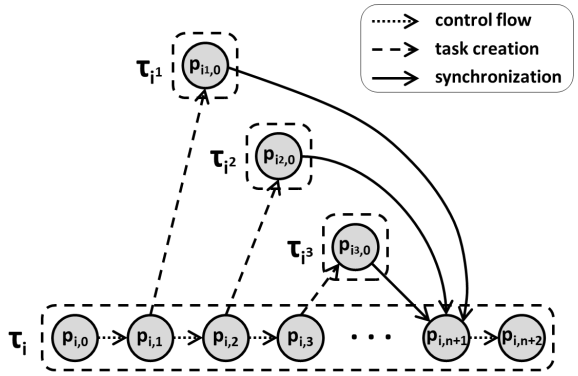

Figure 4.14. DAG of a task in the HASHTAG-TEXT benchmark.

In Figure 4.15, we show the results obtained by the experiments performed without having set any interarrival time between requests, meaning that we have a continuous injection of tasks within the system. This is the most intensive workload

scenario, in which the continuous creation of tasks also generates the heaviest contention for accessing data structures where these tasks are placed before being executed, possibly after being suspended. The bar charts represent the speed-up (slow-down) values of the ULMT-based runtime with respect to the Baseline one for both the application execution time and the response time to requests for the three different priority levels, which were calculated as the ratio between the time difference and the time returned by the experiment performed with the Baseline runtime. By the results, the ULMT-based runtime outperforms the Baseline at all thread counts, with speed-up of the execution time that ranges from 1.5% to about 6.5%. Noteworthy, the application benefits from the support given by the ULMT technology for the execution of OpenMP tasks, together with the mechanisms provided by the re-engineered runtime, thanks to its support for work-conservativeness, while it gets no benefits from the exploitation of IBS interrupts and the associated capability for asynchronous task-switching. This is somehow expected because of the corner case related to the absence of interarrival time between requests, which leads the threads in both the Baseline and ULMT-based runtime to mostly process the highest priority tasks along time, since the corresponding queues are unlikely to empty, thus annihilating the capability of the ULMT-based runtime to more promptly switch to the execution of standing higher priority tasks. Anyhow, this is an anomalous workload scenario that we have considered just for the purpose of completeness of the analysis.

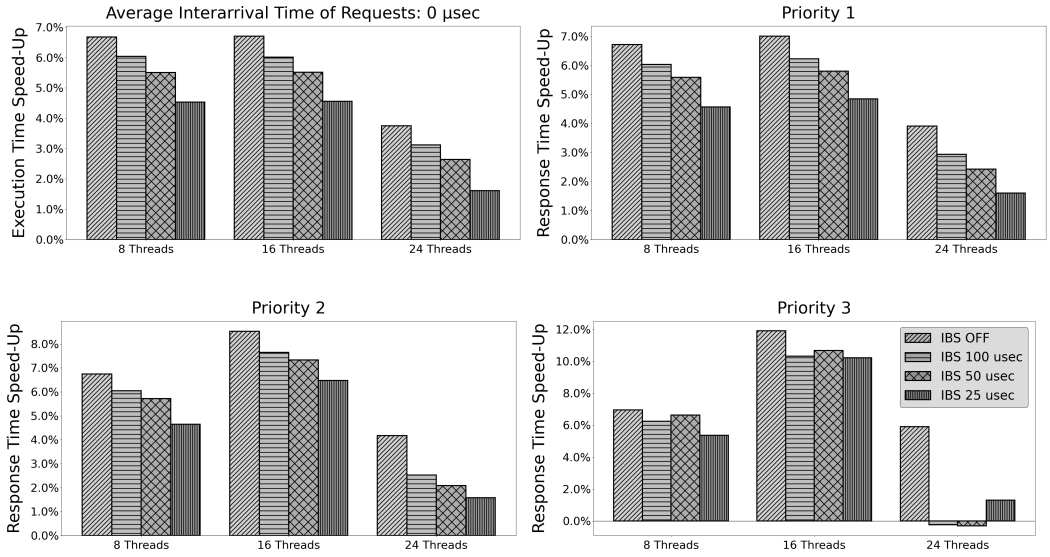

Figure 4.15. Speed-up values of execution and response times when the average interarrival time of requests is set to 0 μsec.

As soon as a reasonable interarrival time of requests is set—representative of workload scenarios that are more likely to characterize the operation of this kind of services when they are deployed on similarly sized computing systems—the benefits provided by the asynchronous task-switching capabilities of the ULMT-based runtime become more evident. In Figure 4.16 we report data for the scenario with interarrival

time of requests set to 10 microseconds. With this setting, the speed-up values of the application execution times provided by the ULMT-based runtime range from 1.5% to 6%, a little less than that achieved with the previous setting, but still higher than that currently obtained with the Baseline runtime, again thanks to the orientation of the ULMT-based one to work-conservativeness. However, the speed-up values achieved on the response time to requests at the higher priority level are considerably increased in all configurations, especially under those settings with a larger number of threads and the highest IBS interrupt rate, in which case the difference with the results obtained without the aid of the IBS hardware support is particularly evident. This clearly confirms the fact that the micro-threading model applied as the execution model for OpenMP tasks is the only one that can make the OpenMP runtime able to promptly react to the sudden materialization of high priority tasks thanks to the possibility that is inherent in this model of asynchronously switching tasks along threads. Globally, the achieved speed-up values of the response times range from 3% to 10%, from 5% to 27% and from 1% to 39% respectively for the requests with priority 1, 2 and 3. This is also an indication of the lightweight nature of our solution to support the asynchronous task switching capability at very fine granularity.

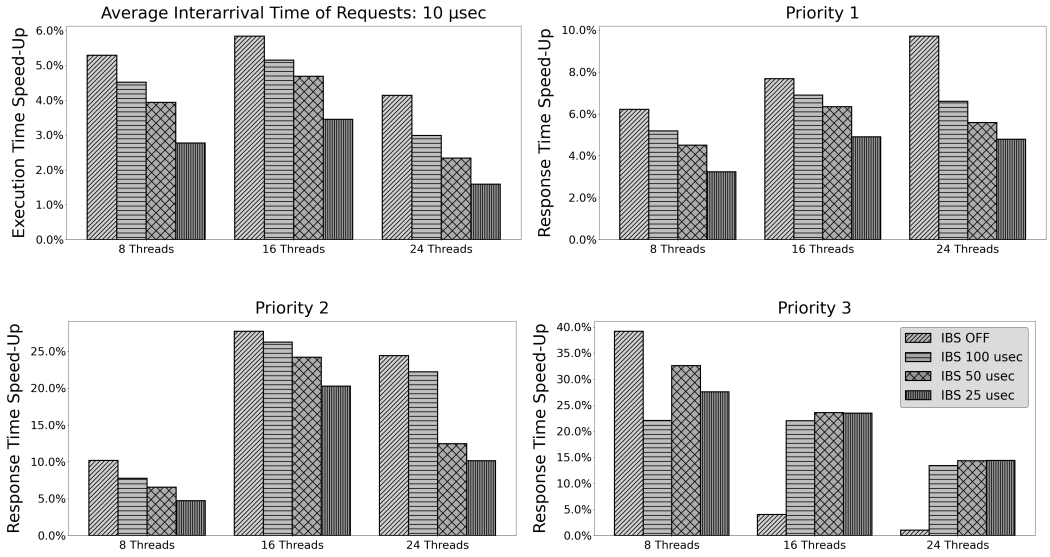

Figure 4.16. Speed-up values of execution and response times when the average interarrival time of requests is set to 10 μsec.

In Figure 4.17, are shown the results obtained by the experiments performed with interarrival time of requests set to 50 microseconds. We can observe speed-up values of the execution time achieved with the ULMT-base runtime that still range from 1.5% to 6%, and speed-up values of the response times that better spread across the highest priority requests by achieving values that range from 4% to 27%, from 8% to 50% and from 1% to 29% respectively for the requests with priority 1, 2 and 3.

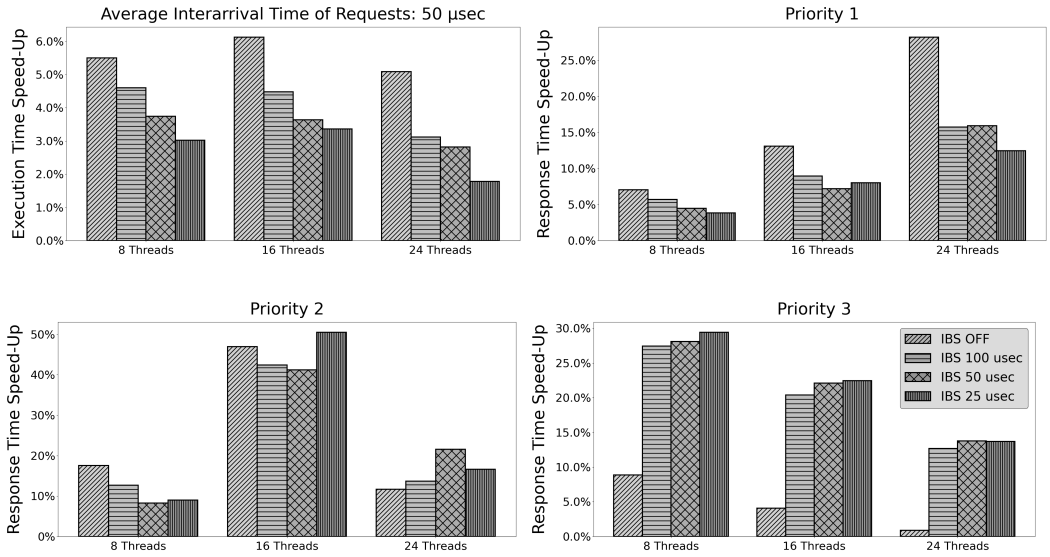

Figure 4.17. Speed-up values of execution and response times when the average interarrival time of requests is set to 50 μsec.

The results reported in Figure 4.18, related to the experiments with interarrival time of requests set to 100 microseconds, highlight even more the benefits deriving from the asynchronous task-switching capabilities of which the ULMT-based runtime is provided. In fact, the execution time speed-up still assumes positive values that range from 1% to 6%, with the only exception of the configuration with 24 threads and no IBS interrupt to support asynchronous task-switches. The motivation is related to the very lightweight workload characterizing this setting, which leads the Baseline runtime to perform like the ULMT-based one when the asynchronous task-switching capabilities are tuned off. In fact, the large number of available threads along with the infrequent arrival of requests ensure that none of the former will have to wait too long before any dependency constraints are resolved by some co-worker when it is running with the Baseline runtime, thus not diverging too much from work-conservativeness. In other words, the condition whereby a thread running in the Baseline runtime is left blocked due to unmet dependency constraints while there is work waiting to be executed does not last long enough to degrade the execution time of the application when it runs with the Baseline runtime. Nevertheless, it still introduces, albeit small, a delay in unblocking the threads, which is a latency that does not affect threads running in the ULMT-based runtime which can always rely in this way on an extra non-blocked thread, even if for a short period of time, to promptly start the execution of tasks at all the priority levels. Clearly at the expense of the execution time because of the overhead costs for managing separate contexts and task-switch operations.

Since the introduction of task priorities in the OpenMP specification is recent, none of the literature OpenMP benchmarks has revealed effective for testing our innovative runtime solution. To bypass this problem, we have presented a fully new benchmark, named HASHTAG-TEXT, that we used to carry out the experiments we described so far, whose results have demonstrated the effectiveness of our proposal.

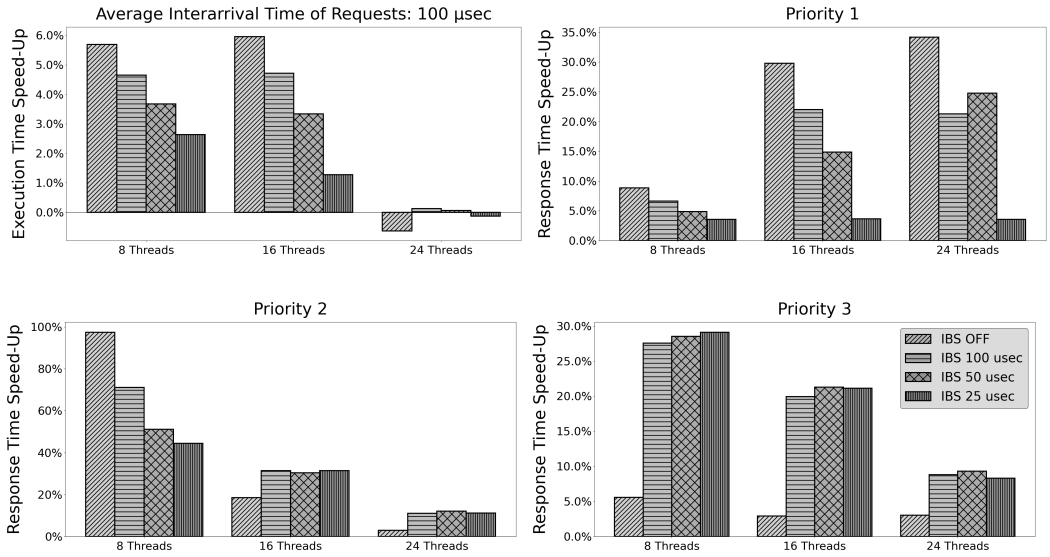

Figure 4.18. Speed-up values of execution and response times when the average interarrival time of requests is set to 100 μsec.

This is a reshuffle of the GNU OpenMP runtime environment, which we have extended with ULMT technology to support the execution of tasks according to the micro-threading model, but still respecting the rules and constraints dictated by the specification for the OpenMP tasking model. Overall, we have introduced two core innovative functionalities: the support for fine-grain asynchronous reassignment of CPU-cores to higher priority tasks, and the support for avoiding the blocking of threads due to dependency constraints whose resolution was charged to different threads (an aspect linked to the so-called work-conservativeness property of OpenMP runtime systems).

4.2 Effective Management of Task Consistency

Speculation is a well-known optimization technique used to improve performance of executions on both single-core and multi-core processor architectures. It is based on the idea of performing work ahead of schedule, without having the a-priori knowledge that the resulting effects will be valid or will have to be discarded due to data dependency and consistency reasons. In Subsection 4.1.1 we largely discussed about transactional memory (TM), presenting it as a paradigm for the management of shared data accesses on multi-core (multi-processor) machines which exploits speculation in combination with concurrency control schemes to make the most of parallelism under both the high- and low-concurrency execution scenarios. In fact, threads are allowed to speculatively perform operations on accessed data that otherwise would have been performed exclusively within a critical section. By the way, conventional synchronization techniques have shown to be unsuitable on modern multiprocessors since they limit parallelism. Rather, the key to highly concurrent programming is to construct classes of implementations that are non-blocking, such

as TM, whereby threads speculatively make tentative changes to shared data that eventually commit if no conflicts are detected. As hinted, the TM paradigm has been implemented at both the hardware and software levels.

As for the hardware implementations, they rely on micro-architectural supports which provide concurrency control capabilities to ensure that correct results for concurrent operations are generated. Such supports allow the notification of conflicts very quickly since they are generally implemented as extensions of cache-coherency protocols. It is indeed clear that any protocol capable of detecting accessibility conflicts can also be extended to promptly detect any kind of conflict between concurrent transactions that would have compromised the serializability of the transaction history. On the other hand, these concurrency control mechanisms also introduce the risk of false conflicts due to the use of cache line granularity. In addition, hardware implementations impose stringent constraint on the size of the write-set that any transaction may use throughout its execution—in any case at most the L1 cache size. Also, they enforce to repeatedly abort transactions that frequently undergo to mode switch, which can always occur due to the arrival of hardware interrupts or upon direct invocation of OS services by applications.

All the aforementioned limitations have then been overcome by the software implementations of the TM paradigm (STM), which in turn have extended their portability to most of computer systems as they do not depend on any particular hardware technology. As the downside, these implementations usually come with a performance penalty when compared to hardware counterparts. This is clearly related to the less performing concurrency control protocols implemented at the software level which cannot rely on architectural supports to timely detected conflicts. In fact, software implementations provide for the detection of conflicts by mean of specific procedures invoked in-line (synchronous) with the transaction operations, whenever it is required by the STM layer. The latter are commonly referred to as validation mechanisms, and are in charge of verifying whether the whole set of data accessed up to this moment is still consistent. Under certain circumstances, consistency validation is an operation that requires to compare the state of each data read with its current state, which is a task that has a cost that grows linearly with the size of a given transaction read-set (it is $O(n)$ in time).

We already mentioned how the TM paradigm was inspired by the concurrency control schemes originally employed by the database management systems (DBMS) to ensure serializability of transaction schedules. Recall, a transaction is defined as a finite sequence of machine instructions whose commit satisfies the properties of serializability and atomicity. In this regard, several STM implementations rely on *optimistic* concurrency control schemes to ensure the before stated properties by allowing transactions to access shared data resources without acquiring locks and without performing particular controls, and by deferring all work of consistency validation to commit time where each transaction can verify that no other transaction has modified the data it has read. If so, the transaction can commit its operations, otherwise it aborts and retries from the beginning. Optimistic concurrency control schemes come from the assumption that multiple transactions can frequently complete without interfering with each other, where conflicts are rare due to low data contention and transactions can complete without the expense of managing locks and/or employing more sophisticated concurrency control protocols. However, if

contention for data becomes frequent, the cost of repeated transaction executions hurts performance significantly as the residual execution of an already doomed to abort transaction can last an arbitrary amount of time. Consequently, the latter scenario leads to a waste of computing resources and energy consumption.

On the other hand, ensuring the correct behaviour of certain STM applications also requires that all running transactions meet specific correctness criteria, such as *linearizability* and *opacity*, at any time of their execution. Linearizability is a guarantee about single operations on single objects, and it is a condition that, to cite [35], provides the illusion that each operation applied by concurrent threads takes effect instantaneously at some point between its invocation and its response. Also, it is a necessary condition, but not sufficient, to ensure atomicity of committed transactions as there must be at least one common time in which all operations can take effect. Opacity [33] is instead a correctness criterion that, at first glance, can be seen as an extension of the serializability property with the additional requirement that every transaction, including non-committed ones, always accesses a consistent state. From another point of view, it is a safety condition that prevents STM applications from running in a incorrect manner whenever transactions could have performed operations on a state that is no longer consistent. All these requirements have made therefore the task of designing concurrency control scheme to support this transaction execution semantics a hard task, given that relying on straightforward incremental validation is not an efficient solution—even if there exist STM implementations that follow this approach such as the ASTM system [55]. It is indeed known that the latter approach would require to perform consistency validation of a given transaction read-set upon each transactional access on shared data, which is an operation that introduces an overhead cost (it is $O(n^2)$ in time) that nullifies any benefit that can be obtained from the execution of transactions.

To reduce these costs some literature works have proposed TM solutions that provide for the execution of transaction in *isolation*, such as Snapshot Isolation [71] tailored for STM, in which transactions are initially provided with a consistent snapshot of all data they will access during execution, then all writes will be atomically performed at a later time than that at which the snapshot is taken. Here, the implemented concurrency control protocol is a lightweight multi-version isolation algorithm that does not use locks and provides for performing a validation check only at commit time. However, despite this algorithm avoids common isolation anomalies like dirty reads, dirty writes and lost updates, it cannot guarantee the serializability property, thus providing weaker execution semantics for transactions that must be taken into account by application developers who wish to rely on this solution. It is in any case an execution semantics that allows to achieve good performance results, along with reduced validation costs, when read-only transactions predominate the workload. Anyhow, any transaction already doomed to abort due to not yet detected conflicts will continue running up to the commit phase where it is effectively aborted, thus leading to a waste of time and resources.

Other works have instead proposed STM solutions based on smarter and more efficient techniques to support stronger execution semantics for transactions, those that meet all the aforementioned properties and correctness criteria. These solutions rely therefore on, let's say *pessimistic*, concurrency control mechanisms that perform a validation check when conflicts that would have potentially compromised the read-

set consistency are likely to have occurred, thus avoiding transactions to continue running as soon as the read-set is no longer consistent. In addition, these solutions differ from each other in that they can rely on visible (as it is for SXM system [32]) or invisible (as it is for TinySTM system [21]) read designs. While the former simplify conflict detection by pessimistically ensuring a consistent view of shared data to applications, the latter are significantly more efficient but require additional effort to preserve consistency, hopefully having to pay an average cost that does not increase quadratically with the number of data read in a transaction. Anyhow, even with such STM solutions aimed at constantly preserving the read-set consistency of live transactions, there is no guarantee about the fact that the threads that are carrying on the execution of transactions perform operations that will surely commit. These threads may indeed perform an arbitrary amount of non-transactional operations between two consecutive transactional accesses to shared data while sudden conflicts may arise at any time during this period, thus making the residual non-transactional operations a waste of computing resources—as a consequence of the fact that detection of conflicts in STMs can take place only upon direct interaction with the transactional layer. Also, at least for those solutions relying on the invisible read design, it is not even possible to guarantee that the data read can linearize for sure at commit time along with any updates, thus making impossible to atomically commit the transaction. This is due to sudden *write-after-read* conflicts that can always arise after having made invisible reads on shared data, which are not promptly detected by the less pessimistic concurrency control mechanisms and which will not be detected up to the next validation check on the read-set—it is indeed true that this kind of conflicts does not corrupt the consistency of data read over which threads are performing operations, even if their occurrence will lead updating transactions to abort due to the impossibility of atomically committing the operations performed.

It is therefore clear that, regardless of the transaction execution semantics and the concurrency control protocols employed accordingly, the problem of promptly detecting conflicts that may have compromised the consistency of data read in STM applications—together with other not detected conflicts that prevent transactions from successfully committing their operations—cannot be addressed only through the design and implementation of ever more efficient conflict detection strategies because, whatever the logic, their execution can only take place when the application interacts with the underlying transactional layer. This problem is of particular interest since from the ability of the STM runtime to promptly rollback transactions doomed to abort derives a saving in the use of hardware resources and in the energy consumption. This becomes more evident when we consider high-concurrency scenarios with long running transactions that perform a huge amount of non-transactional operations between two consecutive transactional accesses or that interact frequently with I/O devices. In these scenarios, nothing prevents the threads from keeping busy the underlying resources while they are performing this kind of operations in the context of transactional blocks, not even when the outcome of such operations is destined to be discarded. STM runtime systems need instead to be supported with mechanisms that make it possible to promptly redirect the execution trajectory of threads to perform validation checks as soon as potential conflicts may have occurred. These asynchronous, lightweight checks must occur more frequently than with calls to functions of the STM framework which clearly depend on the application logic

implemented by the programmer—can be sparse and/or not be uniformly distributed. In any case, such controls must take place regardless of what threads are actually performing and what the control flow graph of the STM application provides.

4.2.1 Prompt Transaction Revalidation in Software Transactional Memory

In this work we precisely addressed the performance and energy efficiency issues that may arise when no shared data accesses occur for a while along the execution path of a thread that is running a transaction. As hinted, the STM layer may not regain control for a considerable amount of time, thus not allowing to early detect if such transaction is no longer able to commit the operations performed due to conflicts that may have occurred in the meantime. This inability to commit will only be discovered later by the STM runtime through a subsequent validity check that will eventually reveal the transaction as doomed to abort. The resulting late aborts will not favour therefore the reduction of wasted computation, penalizing in this way the performance of the execution as a whole. Differently, we want to provide the STM layer with lightweight supports aimed at performing transparent and periodic validity checks of running transactions whenever potential conflicts may have occurred in the meantime. These checks must take place asynchronously with the other operations performed by threads in the context of transactional blocks, always allowing the transactions to promptly resume their execution as soon as they are verified to be still valid. On the contrary, their execution is early aborted as for the effect of having nested lightweight checks along the execution path of the involved threads that have led the transactions to reveal themselves as non-valid, thus savings time and energy that would have been wasted otherwise.

To the best of our knowledge, none of the past literature works provides a solution aimed at early detecting conflicts caused by concurrent threads that may have invalidated the execution of an in-memory transaction, that is, prior to the time when such detections were expected to take place synchronously with other transactional operations as provided by the concurrency control scheme involved. One major trend in TM systems has always been that of reducing as much as possible the incidence of transaction aborts as it is a performance indicator related to the amount of unfruitful work performed. It is indeed known that TM systems outperform lock-based synchronization strategies when the executing workloads contain sufficient inherent parallelism. In this regard, there are some works in the literature that have proposed *transaction scheduling* policies [87] to control whether some standing transaction can be admitted to the processing stage, or needs to be delayed for a while, because of a high likelihood of conflicts with already running transactions. Other works have proposed instead *thread scheduling* policies [14, 13] as an alternative approach to the reduction of the incident of aborts, by determining at run-time the well-suited thread level parallelism for TM applications. Overall, both approaches provide solutions that either try to sequentialize conflicting transactions on the same thread or control the concurrency degree of TM applications by changing the number of threads/transactions that are allowed to run in parallel. Then, there are works [73, 15, 10] in which analytical runtime *decision models* have been proposed to determine suited levels of parallelism for running applications, as a way to avoid

trashing due to excessive transaction aborts. Anyhow, none of them is an attempt to reduce the processing time of transactions destined to abort.

Once again, we find the micro-threading model to be the ideal solution to cope with the above discussed problem as in the first instance it allows to overcome the limitation represented by the fact that threads carry on the execution of transactions as non-interruptible tasks. So, analogously to the work presented in Subsection 4.1.1, we decided to exploit the ULMT technology to accomplish the commissioning of the execution model for transactions based on micro-threads. This makes it possible to temporarily pause the execution of in-memory transactions, so as to asynchronously pass control to a function designed to verify whether or not the suspended transactions have any chance to successfully commit their operations. In Figure 4.19 are shown the two possible evolutions of the proposed procedure that are either to resume the interrupted execution or to early abort the transaction. A thread that undergoes control flow variation due to the arrival of an IBS interrupt is shifted to execute the VALIDITY_CHECK function which verifies the validity of the currently performed transaction—this involves a check on both the read-set consistency and the possibility for a transaction to atomically commit its tentative changes on shared data. It is indeed common that the VALIDITY_CHECK function coincides with the validation mechanism already in use within the STM system to carry out validation and consistency check according to the concurrency control scheme involved. If no conflicts that would prevent the transaction to commit are detected, then *a*) the interrupted execution is immediately resumed, otherwise *b*) the transaction is aborted and the execution rollbacked to start a retry.

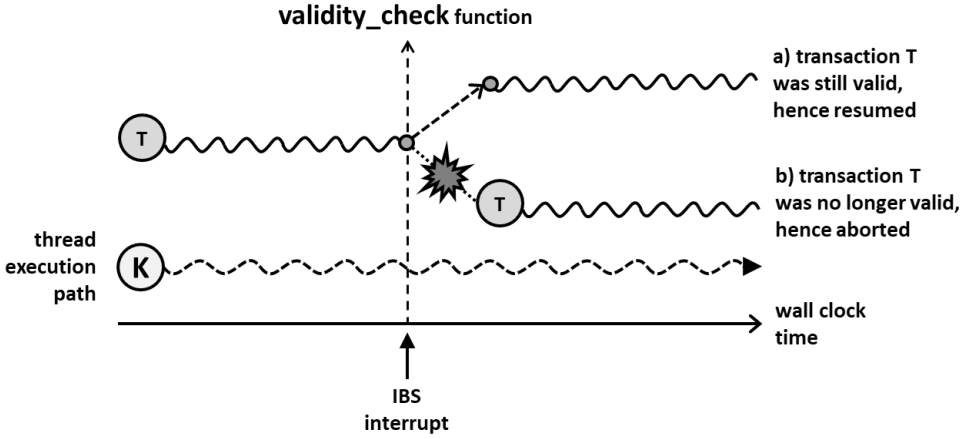

Figure 4.19. Early check of the transaction validity.

Although this solution is in principle integrable with any STM framework and suitable to support whichever concurrency control protocol aimed to meet the requirements for accomplishing different transaction execution semantics, we chose the TinySTM package [21] as the STM runtime system to use for assessing the effectiveness of our solution. TinySTM is a word-based STM implementation that uses a single-version variant of the Lazy Snapshot Algorithm (LSA) [20, 70] and a time-based design to benefit from the performance advantage of invisible reads without incurring the quadratic overhead of incremental validation. It is word-based

in that the STM design can detect conflicts between concurrent transactions on the granularity of memory regions, *e.g.*, 8 bytes. It is time-based in that it relies on a shared atomic counter—namely the *global version clock* (gvc)—to keep track of the logical time advancement that occurs with the committing of transactions. In this regard, when a transaction attempts to commit the speculatively performed changes on shared data, the gvc is atomically incremented by the transaction and its new value is reflected as the new timestamp of updated data. Also, TinySTM is said to use a single-version variant of the LSA due to the fact that the employed concurrency control protocol relies on an implementation of this algorithm that does not retain the value of old versions of data being accessed. In particular, LSA exploits the concept of logical time to efficiently construct *snapshots* of versioned data, accessed by a given transaction, that remain consistent during its whole execution. Every transaction maintains therefore a snapshot that corresponds to a range of valid linearization points, which would allow the transaction to commit only if it is non-empty at completion time. It is initially set to a range between the current value of gvc and, let's say, the infinite value, whose bounds are then adjusted to match a new range resulting from its intersection with the validity range of the most recent version of each new data read—since TinySTM uses a single-version variant of LSA, the validity range of a data being read always starts with its modification timestamp and terminates with the current value of gvc. This allows LSA to keep the read operations of a transaction invisible to other transactions, and to verify consistency of the accessed data by maintaining a validity interval for snapshots on the basis of data modification timestamps obtained from the gvc. It is clear that, when a transaction reads the latest version of a data, the upper bound of its validity range is capped by the current value of gvc. However, it is possible that the data read has a validity range that starts from a timestamp which is greater than the upper bound of the snapshot owned by the transaction, so that the transaction itself must attempt to perform an *extension* of the same snapshot. The latter essentially requires to check if all previously read data are still valid in a snapshot that includes the timestamp of the one currently read, thus decreeing that the extended snapshot is representative of a consistent state of data read. Otherwise, the transaction cannot read a valid version of data while maintaining a non-empty snapshot, and for this reason it is forced to abort. It must be noted that any updating transaction can commit with a certain timestamp only if such timestamp falls within the bounds of its snapshot since all read and write operations it performed during the execution must linearize at a common time as if they were atomically performed. For serializability reasons, the latter must coincide with the value of the shared counter represented by the gvc, which has been atomically incremented by the involved thread just prior to perform the ultimate validation of data read by the transaction being committed.

From this brief overview of the LSA specifications it is possible to understand how TinySTM is a highly efficient STM system already in its base implementation, and for this reason a strong opponent against which to assess our solution. Nevertheless, the abovementioned write-after-read conflicts are a kind of conflict that remains hidden from the concurrency control mechanism employed by TinySTM up to the next performed validation, which does not necessarily coincide with the next transactional access to shared data by the STM application, as shown in the example reported in Figure 4.20, thus increasing even more the time spent in executing transactions

doomed to abort. As for the latter aspect, an undefined number of transactional accesses can be reflected in reading data whose most recent version has a timestamp that immediately falls within the bounds of the snapshot, thus not requiring to perform a validation of the extended snapshot which would have instead revealed this kind of conflicts.

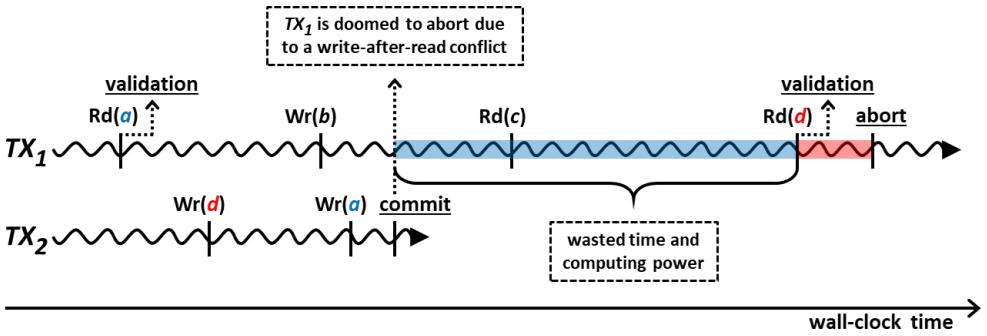

Figure 4.20. Wasting of time and computing power due to an unrevealed write-after-read conflict.

Our prompt revalidation approach exactly solves this problem, which is relevant when also considering that the action performed by the application code within the transactional block are essentially arbitrary and only related to the way the application logic is implemented, which may ultimately involve a huge amount of non-transactional operations over which the STM runtime has no control. It must also be noted that our prompt revalidation architecture operates synergistically with the aforementioned snapshot extension mechanism since, as a side effect, our solution can also anticipate a portion of the extensions that would be performed by TinySTM spontaneously, thus making a higher fraction of future reads fit into the current snapshot. As such, the before stated VALIDITY_CHECK function coincides with the stm_extend API offered by TinySTM. The core parts of this architecture are still the IBS interrupt handler together with the CFV_TRAMPOLINE function. As already discussed in Chapter 3, the latter function is expected to execute upon return from an IBS interrupt in such a way to complete the control flow variation (CFV) procedure initiated by the IBS interrupt handler. It is in charge of restoring the original stack of the micro-thread assigned to the interrupted transaction. Also, it makes some checks about the preemptibility of the involved micro-thread just prior to save its context. Anyhow, for the purposes of this work and efficiency reasons we have revised the original version of the CFV_TRAMPOLINE function in such a way to include additional checks that will be immediately evaluated upon return from an IBS interrupt. The revised version of this function is reported in Algorithm 5 under the name CFV_TRAMPOLINE_REVAL. These checks are intended to avoid the passage of control to the stm_extend function, which would carry on a revalidation of the whole read-set otherwise, whenever it is known that no conflict can have occurred or it is unlikely to have occurred for the involved transaction. On the contrary, a revalidation of all data read by the current transaction is advised and pursued by invoking the stm_extend function which, depending on the outcome of the snapshot extension procedure, determines whether the previously saved execution context

must be restored (line 23 in Algorithm 5) to continue the execution of the transaction or the latter must be aborted (line 25 in Algorithm 5) as its snapshot of data read can no longer be extended to commit time. In the second case, the transaction execution state is destined to be discarded together with any data read and written during its current execution instance, which is an operation accomplished by invoking the stm_rollback API offered by TinySTM.

Algorithm 5 CFV trampoline for transaction revalidation.

```
 1: procedure CFV_TRAMPOLINE_REVAL:
 2:     SWITCH_STACK()
 3:     tx ← GET_CURRENT_TRANSACTION()
 4:     if tx == NULL then                          ▷ not running a transaction
 5:         RESET_RECENTLY_VALIDATED()
 6:         return
 7:     ub ← GET_UPPER_BOUND(tx.snapshot)
 8:     gvc ← GET_GLOBAL_VIRTUAL_CLOCK()
 9:     if ub == gvc then                           ▷ no commit occurred
10:         RESET_RECENTLY_VALIDATED()
11:         return
12:     rv ← GET_RECENTLY_VALIDATED()
13:     if rv then                                  ▷ recently validated
14:         RESET_RECENTLY_VALIDATED()
15:         return
16:     pc ← READ_PREEMPTION_COUNTER()
17:     if pc > 0 then                              ▷ non-preemptible code region
18:         SET_STANDING_INTERRUPT()
19:         return
20:     MtSnap ← GET_CPU_SNAPSHOT()
21:     if CONTEXT_SAVE(MtSnap) == 0 then
22:         if stm_extend(tx) == 1 then             ▷ snapshot extension
23:             CONTEXT_RESTORE(MtSnap)
24:         else
25:             stm_rollback(tx)
26:     return
```

The decision on whether to invoke the stm_extend function in order to perform an asynchronous and transparent transaction revalidation is therefore directly actuated into the trampoline function in a lightweight manner. Specifically, the CFV_TRAMPOLINE_REVAL function avoids bringing control to the STM layer if:

a) the thread running the STM application is not currently executing a transaction (verified at line 4);

b) the thread is executing a transaction but the transaction is surely valid (verified at line 9);

c) the thread is executing a transaction but the transaction has been already (very) recently validated (verified at line 13);

d) the thread is executing a transaction but its execution flow cannot be momentarily interrupted (verified at line 17).

As for points *a*)-*c*), the CFV_TRAMPOLINE_REVAL function makes some checks on the value held by three main variables, that are: a per-thread pointer to the

structure that holds the metadata of the currently executed transaction, and which is called transaction_ptr; the upper_bound variable instantiated along with the snapshot owned by the interrupted transaction, which is a value less than or equal to the global virtual clock (gvc); and a per-transaction variable named recently_validated indicating whether or not the currently performed transaction has been (very) recently validated. It is clear that, if transaction_ptr is equal to `NULL` then the problem does not arise as this pointer is always updated upon entering and exiting a transactional block. Also, if the upper_bound holds the same value of the gvc no concurrent commits can have occurred since the transaction's snapshot was last extended, meaning that the interrupted transaction is certainly valid because of the absence of any update on shared data. The recently_validated variable is instead a sticky flag used to check whether the validation task has been already performed between the arrival of two subsequent IBS interrupts, that is, because of a snapshot extension attempted by the STM runtime. This flag is set to *true* whenever a synchronous execution of the stm_extend function takes place. Then, the CFV_TRAMPOLINE_REVAL function tries to reset this flag to $false$ regardless of any possible filtering condition (lines 5, 10 and 14 in Algorithm 5), as a way to capture the notion of *time proximity* with respect to the last occurred validation. However, finding this flag set to *true* does not imply that concurrent updates have not certainly occurred in the meantime. It is only the attempt to avoid performing the transaction read-set validation when potential conflicts are unlikely to have occurred. We would like to make notice that, if the recently_validated variable is found set to *true* too many times, then it is possible that the time period between IBS interrupts is too large. On the contrary, if for a high number of times the upper_bound variable is is found to hold a value that equals the value of the gvc, then the time period between IBS interrupts may be too small. Both cases can be addressed by fine-tuning the time interval between subsequent IBS interrupts.

As for the point *d*), we note that interrupting the execution of some external library function to carry out the revalidation task would require that function to be re-entrant along the same thread. The same reasoning applies to functions within the STM layer, which are expected to perform entirely their work. Both cases are solved by mean of per-thread atomic counters, namely the preemption_counters offered by the micro-threading library, that increment by one each time the aforementioned code regions are accessed along the execution flow of threads, and decrement when leaving these regions. As already discussed in Chapter 3, the logic related to the updating of these counters is completely transparent to the STM applications and to the STM runtime as it is realized through wrappers surrounding calls to these functions that are injected into the code by the compile/link infrastructure. However, this approach would lead to simply forgoing the exploitation of IBS interrupts delivered during the execution of non-preemptible regions, which in turn would prevent transactions from early aborting their execution as soon as they become invalid. Therefore, similarly to what we did in the work presented in Subsection 4.1.1, we use a second variable named standing_interrupt to indicate that an IBS interrupt has arrived while the thread was running in non-preemptible mode (line 18 in Algorithm 5). Once the execution comes out from one of these regions, the code placed at the tail of the surrounding wrapper first checks if preemption_counter is equal to zero and then verifies if standing_interrupt has been set to *true*, in which

case a deferred execution of the revalidation task takes place. The execution of this task is in any case dependent on the outcome of checks referred to in points *a*)-*c*) which are performed this time synchronously by the wrapper.

Overall, the resulting preemptive STM architecture will be such as to allow the thread that are carrying out the transactions to be able to periodically pause their executions so as to perform fast checks aimed at determining whether potential conflicts may have been occurred, and in case to carry on a prompt revalidation of the whole read-set. In Figure 4.21 is shown an execution example with two threads that are charged of the execution of two concurrent transactions, whereby the transaction TX_2 generates a conflict on the data a with the transaction TX_1 upon committing its work. It is therefore clear that the updating transaction TX_1 is no longer able to commit its work as there is no possibility of linearizing both read and written data at a future commit time, thus it would be desirable to interrupt as soon as possible this execution in order not to waste computing power for a long time. The latter is accomplished with the aid of ULMT technology and the support given by the IBS hardware facility, which together allow to perform asynchronous (with respect to the operations performed by the transaction) fast checks at regular intervals (according to the filtering rules in *a*)-*d*)), which eventually lead to revalidate the transactional read-set in order to detect conflicts that condemn the transaction to be aborted.

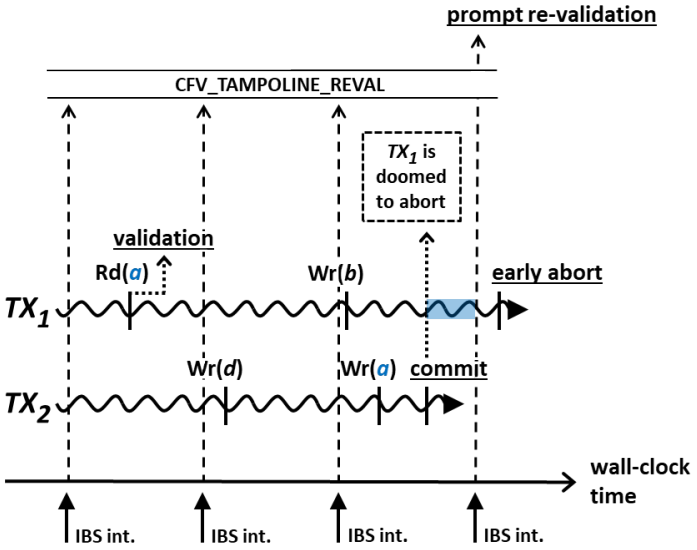

Figure 4.21. Prompt revalidation resulting in early abort.

To assess the effectiveness of our architecture we ran the experiments on a 64-bit NUMA HP ProLiant Server equipped with 64GB of RAM and four 2.4GHz AMD Opteron 6128 processors, each one having 8 cores, for a total of 32 CPU-cores. Also, the operating system is OpenSuse 13.2 with the version 3.16.7 of Linux kernel. To study the effects of transaction revalidations occurring at different frequencies along the application lifetime we have used three different time intervals to characterize the arrival rate of IBS interrupts, namely 200, 100 and 50 microseconds.

As a benchmark application, we have used a port of the TPC-C benchmark [78] to the STM environment. TPC-C is representative of OLTP workloads and includes

5 different transaction profiles that simulate a wholesale company supplying items from a set of ware-houses to customers within sales districts. In our experiments we instantiated one district, and generated a workload made up by requests spanning 4 different transaction profiles specified by the benchmark, excluding the delivery profile since it is conceived to be executed in deferred mode as per TPC-C specification. In table 4.4 is reported the list of transaction profiles, and their characteristics, that we have employed in our experiments. We note that transactions belonging to different profiles exhibit very different CPU demands and different data access pattern, thus enabling a study of our proposal with an articulated workload. In our porting to the target STM environment, CPU demands range from tens to several hundred of microseconds. In addition, transactions from different profiles contribute with a different percentage of the final workload. This diversity in the granularity of different transaction profiles further allows to assess our prompt revalidation architecture against a non-favourable workload, in that it also includes transactions which are unlikely to be hit by IBS interrupts due to their very fine-grain nature, *i.e.*, payment and order status. Hence, in this scenario, the capability of our architecture in terms of possibility to hit running transactions via IBS interrupts is limited to a fraction of the overall workload.

Table 4.4. Transaction profiles and associated workload characteristics.

ID	Transaction Profile	CPU Demand	% Mix
1	new order	$\approx 350\ \mu sec$	49
2	payment	$< 10\ \mu sec$	43
3	order status	$\approx 10\ \mu sec$	4
4	stock level	$\approx 650\ \mu sec$	4

We run our experiments with continuous injection of transactional requests, using either 8, 16 or 24 threads for processing the requests, and 6 threads for managing the socket pool from which the work is retrieved—transactional requests are issued by a workload generator that runs on another machine connected via a switched 100Mb Ethernet. We decide to vary the number of threads used to process transactions in order to assess our proposal with different levels of actual transaction concurrency. In any case, at each thread count we always run with the highest concurrency level since we configured TinySTM to rely on the commit-time-locking (CTL) scheme for the tentative write operations—rather than the encounter-time-locking (ETL) one. Anyhow, the described configurations have led to use at most 94% of the CPU computational power, thus avoiding hardware resources saturation which would have affected the reliability of the experimental analysis. Also, each experiment with 8 threads entails 1 million committed transactions, while all the experiments with 16 and 24 threads entail 2 and 3 million committed transactions respectively. As a final aspect, we do not need to manage any pool of contexts this time as the number of simultaneously active transactions is always less than or equal to the number of threads employed in the experiments. This means that a single micro-thread context is sufficient to handle consecutive transaction executions along the same thread, each one in a fresh incarnation of its content.

In Figure 4.22 we report the transaction throughput that we observed in the different configurations—labelled with a common prefix PTR—plus a Baseline experiment where our prompt revalidation architecture is disabled. Each histogram refers to an average over 10 runs of the same configuration, for which we avoided to report their variances since the results for different runs where within the 2% of each other. We have also included the throughput observed when running with the setting of TinySTM based on ULMT technology but no control flow variation is ever actuated upon the arrival of IBS interrupts—labelled with a common prefix OVH-IBS—as a way to assess the overhead introduced by the IBS interrupt handling along with the management of micro-thread contexts. By the result, we see that the prompt revalidation architecture allows improving the system throughput, compared to the Baseline, by up to 17% and up to roughly 8000 additional transactions per second in absolute terms. The maximum gain is noted for the 24 threads case, meaning that our prompt revalidation mechanism leads to better exploit the increased parallelism in the execution of transactions. As for the OVH-IBS setting, it can be seen that it shows no more than 9% worse performance with respect to the Baseline. More important, such performance loss tends to slightly scale down at larger thread counts. Overall, the data suggest that our mechanism provides better benefits in the relevant scenario where there is a high degree of actual transaction parallelism, which gives rise to many conflicts that cannot be detected in time by the underlying STM runtime in case of the Baseline setting of TinySTM. Moreover, the overhead results show that we are able to consistently defeat the actual overhead introduced by the IBS interrupts and their management logic, which does not undermine the benefits of our prompt revalidation mechanism.

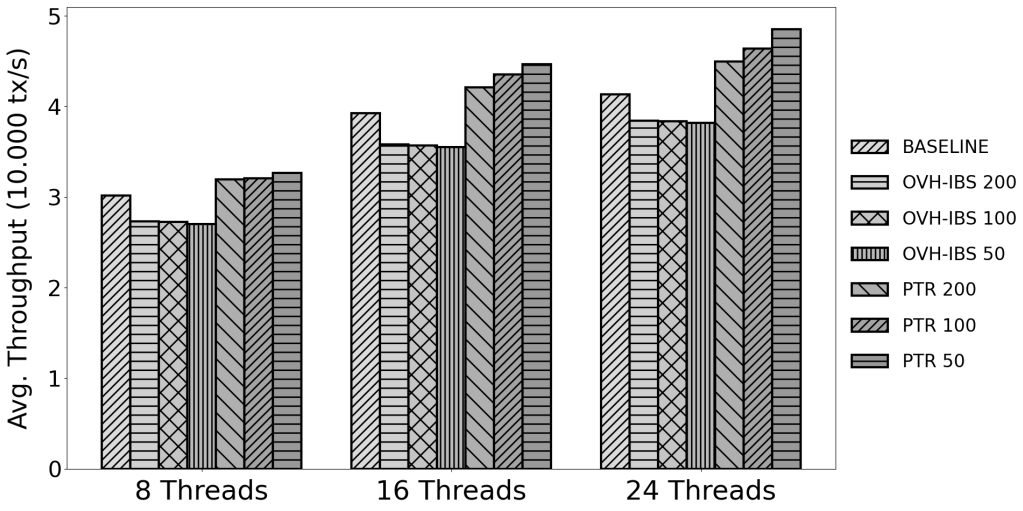

Figure 4.22. Throughput of committed transactions.

A second batch of experiment results are reported in Figures 4.23, 4.24 and 4.25 in order to show the performance of our system in terms of, respectively, increase in the number of successful validations with snapshot extension per commit, variation in the number of aborts, and improved turnaround time per profile. The number of successful validations with snapshot extension per commit is definitely increased

compared to the Baseline, with values that grow steadily while moving from PTR 200 to PTR 50. This illustrates that our architecture is much more capable to check the validity of ongoing transactional work and re-evaluate running transactions. As for the aborts, we can see that our prompt revalidation architecture gives rise to no more than 130% of the aborts experienced by the runs made under the Baseline setting for the two transactional profiles that constitute the major portion of the overall workload, *i.e.*, new order and payment. However, a higher number of aborts does not necessarily imply a longer turnaround time for completing a transaction. In fact, we can see that the average turnaround time per profile—the latency from the start of a transaction processing to its commit, including the time of intermediate aborted runs—is reduced by up to 25% for the most relevant profile, namely the new order transaction, which is long running and has a very relevant weight in the workload mix. On the contrary, a significant worsening of the turnaround time is noted only for the stock level transaction, which has however a marginal weight in the workload mix and does not affect the system throughput.

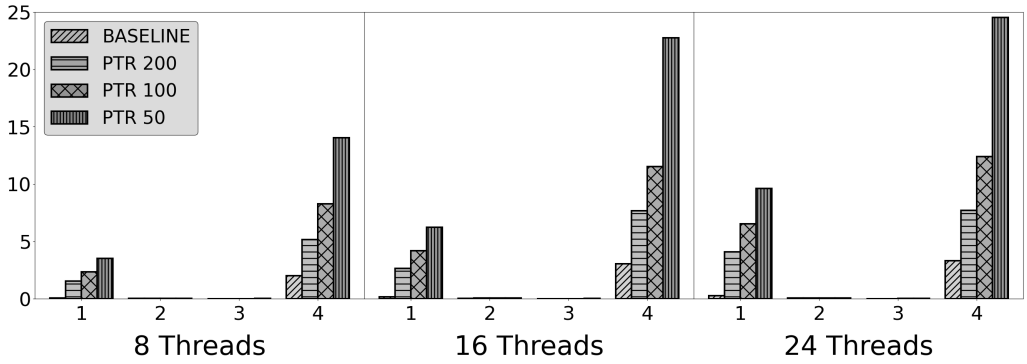

Figure 4.23. Number of successful validations with snapshot extension per commit (y-axis) per transaction profile (x-axis).

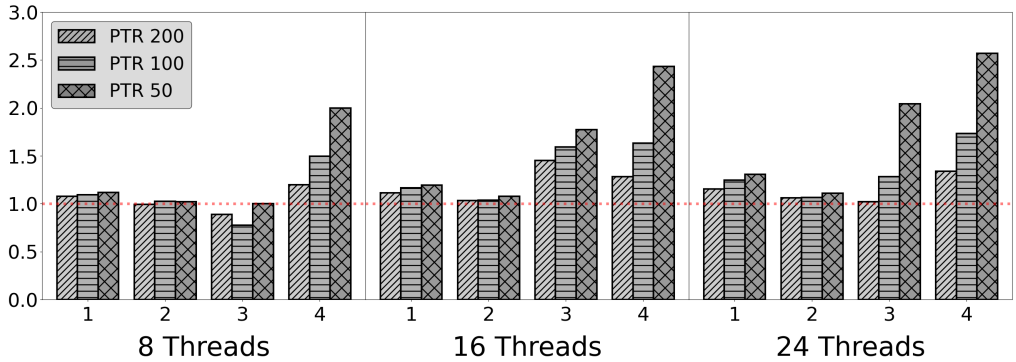

Figure 4.24. Total number of aborts (y-axis) per transaction profile (x-axis) relative to Baseline.

Overall, the experimental data confirm the effectiveness of our architecture based on ULMT technology and the support given by the IBS hardware facilities in order to periodically re-evaluate the transaction validity and its accessed data consistency.

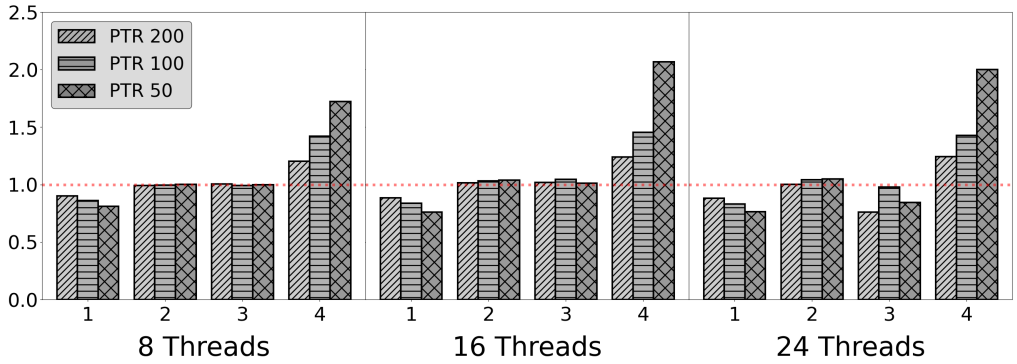

Figure 4.25. Average turnaround time (y-axis) per transaction profile (x-axis) relative to Baseline.

This is accomplished transparently with respect to STM applications, through prompt revalidation of running transactions that occurs asynchronously with other operations performed in the context of transactional blocks, in a lightweight manner. Such approach is independent from the STM system taken into account, and it is combinable with the implementation of any concurrency control scheme supporting the execution of in-memory transactions. In fact, it synergistically cooperate with the latter in order to early detect conflicts whenever the STM runtime misses this opportunity, thus leading to experience early aborts of no longer valid transaction executions with a reduction of the CPU cycles spent for performing work destined to be discarded.

Chapter 5

Interrupt-Driven Micro-Thread Scheduling

In Chapter 4 we have shown how to use specific performance counter registers built into `x86_64` processor architectures in order to enact time-based control flow variation (CFV) and micro-thread scheduling. Such solution has allowed to extend the scheduling capabilities of ULMT-based systems to points of the execution where the renewal of task-to-thread assignment was not expected to take place, those coinciding with the expiration of the timers whose functionalities are implemented by the registers stated above. As hinted, when following this time-based approach to support preemptive execution of micro-threads, the time interval between two subsequent CFVs for a given thread can be either regular—a fixed value specified by the thread when registering in the kernel module—or irregular—a value that varies throughout the execution to adapt to the time interval that allows to reach the highest performances. In any case, both approaches lead to set such intervals with values resulting from the timing according to which specific events, those causing program state changes, have occurred either in past runs or in the current one, or by simply monitoring performance and taking actions accordingly. It is clear that selecting too large values will lead to little overhead but will not either pay off in benefits due to fewer chances and consequent latencies for reacting to sudden program state changes. On the contrary, too little values will make the system more responsive in terms of timely detecting program state changes, but at the cost of having to pay the overhead introduced by too frequent activations of the CFV procedure. The right value is likely to lie in the middle, which also confirms the fact that in order to be reactive upon the occurrence of program state changes we must accept that several activations of the CFV procedure will not lead to fruitful renewals of the work assigned to threads. This is the curse of the time-based approach to which certain applications are doomed to undergo, those for which there are no viable alternatives for threads to detect program state changes but performing periodic controls. As for the latter, we want to mention those applications for which program state changes may not depend on the operations performed by the threads participating in the execution of the application's tasks, rather they might depend on external activities not provided with the application logic. Furthermore, such changes may also be due to environmental conditions which can only be tested at

run-time.

Nonetheless, in almost all the application contexts, the program state evolves as a result of the effects generated by the execution of the tasks belonging to the application. These evolutions can take effect upon completion of tasks or in intermediate stages of their execution depending on the application logic. However, whatever the timing, at least one thread is always aware of their occurrence, meaning that this thread could potentially be in charge of informing other threads of the occurrence of such changes, all those that need to re-evaluate their current execution in order to prevent the application from running suboptimally. Anyhow, a similar behaviour cannot be achieved through classical signaling mechanisms as their operation is limited by the timing according to which the OS activities take place over time. Conversely, we would like to dispose of a mechanism that allows the aforementioned threads to induce a timely and asynchronous activation of the CFV procedure along the execution path of other threads in a lightweight manner. With a similar solution to support the preemptive execution of tasks in ULMT-based applications it would be possible to set up highly reactive and performing systems. This can be easily deduced from the fact that any activation of the CFV procedure along the execution path of threads would take place only when it actually needs to occur, that is, after a thread has performed operations that have led to a change in the program state, a variation that can have led to non-optimal execution conditions under which it is not desirable to leave the application running, which is why a prompt re-evaluation of the execution of one or more threads is required.

In order to make the proposed approach feasible, we have again moved our attention to specific registers present within the advanced programmable interrupt controller (APIC) with which all modern processor architectures are equipped. More precisely, we refer to the Local-APIC (LAPIC) embedded into each CPU-core that we discussed extensively at the beginning of Chapter 4. In Figure 5.1 we report a high-level representation of the APIC configuration in a SMP system. As already mentioned, the LAPIC controls the delivery of interrupts to the relative CPU-core by mean of a set of registers whose values determine the mode of handling internal and external interrupts. Among them, there is a special register named Interrupt Command Register (ICR) [1, 39] which can be programmed by threads with sufficient privileges in order to trigger software-initiated interrupts to other LAPICs. The latter are even known as Interprocessor Interrupts (IPI) and can be issued by any CPU-core of a SMP system with the aim of forcing the execution of an interrupt handler on one or more remote CPU-cores. By the way, this technology is already used in OS kernels to manage various kind of tasks, such as making the address space of a process consistently accessible by all its threads. This involves sending IPIs to notify other CPU-cores of changes in the mapping or access rules to virtual memory caused by a thread running on another CPU-core, thus leading the CPU-cores hit to flush their TLB for rejuvenating its content on the basis of page-table updates. The OS also uses the IPI technology to enable all (or a subset of) the CPU-cores to be notified that a given function needs to be executed by them. This is the case where threads running on different CPU-cores have to perform a CPU-reschedule because of something happened on another CPU-core, namely the sender of the IPI. However, the IPI technology has never been exploited to provide solutions useful for specific application contexts, nor to coordinate the execution of threads when the

actions taken by one of them require all the others to re-evaluate their activity.

This is therefore the architectural support we were looking for to implement a low-level software infrastructure specifically designed to allow the threads to interrupt the execution of other threads running on distinct CPU-cores with the final goal of inducing asynchronous CFVs along their execution path. Hence, these threads can be subjected to a CFV at any time while performing task-specific operations as a result of receiving IPIs issued by other threads that have decreed that the execution trajectory of the former needs to be re-evaluated. Precisely for this reason, we refer to this approach of supporting the preemptive execution of tasks as the *interrupt-driven* approach. Somehow, this approach can also be seen as a new paradigm for the execution of micro-threads, which enables self-adjusted execution dynamics in ULMT-based applications. More specifically, the applications implemented by following a parallel programming model that relies on task-parallelism as the strategy for decomposing the work, and which rely on the ULMT technology in order to accomplish the commissioning of the micro-threading model for the execution of tasks, can see the threads employed to carry out their tasks to possibly undergo interruptions as a result of program state changes caused by other threads which, aware of the effects that such changes will have on performance, promptly inform the former of the need to re-evaluate their execution in order to prevent the applications themselves from running in conditions that are far from the optimal.

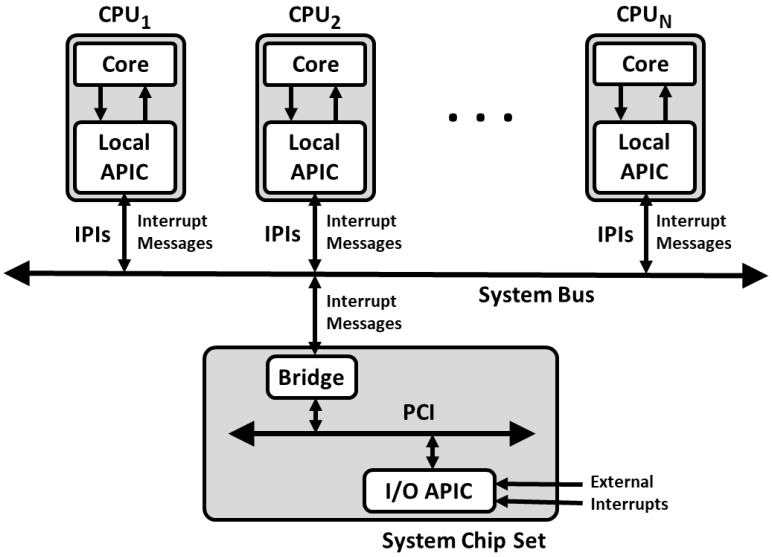

Figure 5.1. APIC configuration in a SMP system.

We already mentioned how ICR is the register to be used to send IPIs toward remote LAPICs. More in detail, it is a 64-bit LAPIC register that is provided for issuing several types of interrupt to whichever CPU-core present in a SMP system, including the one to which the register actually belongs. These comprise system management interrupts (SMI), non-maskable interrupts (NMI), initialization messages (INIT) and start-up inteprocessor interrupts (SIPI) other than normal interrupts (Fixed) issued along with a vector number. Once the ICR register is

programmed by a given CPU-core, the IPI is immediately written on the system bus over which all LAPICs are listening for messages or interrupts destined to them. Therefore, as soon as one of the LAPICs determines that the IPI was in fact destined to it, the proper interrupt delivery protocol is activated by that LAPIC which, depending on the type of the interrupt, also determines the manner through which the interrupt should be delivered to the relative CPU-core. Figure 5.2 shows in detail the ICR register. To send an IPI, software must set up the ICR to indicate the type of IPI message to be sent and the destination processor or processors. As for the former, it can be set by writing into the three bits reserved for the message type (MT filed) to indicate whether the interrupt is of type Fixed (000), Lower Priority (001), SMI (010), NMI (100), INIT (101) or SIPI (110). Differently, the destination is specified in the eight MSb of the ICR register (DES filed) which are expected to keep the physical ID of the target LAPIC whenever the bit reserved for the destination mode (DM field) is zeroed, otherwise it specifies a logical destination which may be one or more LAPICs with a common destination logical ID. Anyway, the destination field is not ignored as long as the two bits reserved for the destination shorthand (DSH field) have been set to zero. On the contrary, another semantics is applied for sending IPIs, which can be: send only to itself (01), send to all including itself (10), or send to all excluding itself (11). As for the other fields belonging to the ICR register, there are the trigger mode bit (TGM filed) used to specify if the interrupt detection is level-sensitive (1) or if it is triggered upon edge-switch (0), the level bit (L field) to indicate the interrupt assertion (1) or de-assertion (0), and the delivery status bit (DS filed) which is read to discern when the IPI has been sent and the LAPIC is waiting for it to be accepted by another LAPIC (1) from when the sender LAPIC is actually idle (0). In addition, there are two more bits reserved for the remote read status (RRS filed) which can be accessed to check whether the current read status of a remote LAPIC is encoded as invalid read (00), delivery pending (01), or delivery done and valid read (10). Last but not least, the eight LSb of the ICR register (VEC field) are used to specify the vector number that is sent along with the IPI when the interrupt is of type Fixed. In this particular case, the interrupt delivery protocol provides for handling the IPI through an interrupt handler which is pointed by an interrupt-gate whose descriptor within the interrupt description table (IDT) is placed at the position specified by the vector number. As hinted in Chapter 4, the IDT is a data structure used to implement the software-side interrupt vector table that every processor consults to determine the correct response to interrupts and exceptions. This makes therefore the ICR register particularly powerful as it can potentially be used to program any kind of interrupt having a corresponding entry in the IDT table with the final goal of executing the associated handler on one or more CPU-cores.

Since our goal is to set up a low-level software infrastructure aimed at allowing the threads to induce the asynchronous activation of the CFV procedure along the execution path of other threads, we enable these threads to program the ICR register so as to send IPIs of type Fixed along with the vector number that matches the offset in the IDT table that corresponds to the interrupt-gate descriptor associated with the handler implemented within the kernel module provided with the ULMT technology. However, despite the ICR register can be read and written through simple data movement instructions, the APIC register space to which it belongs is

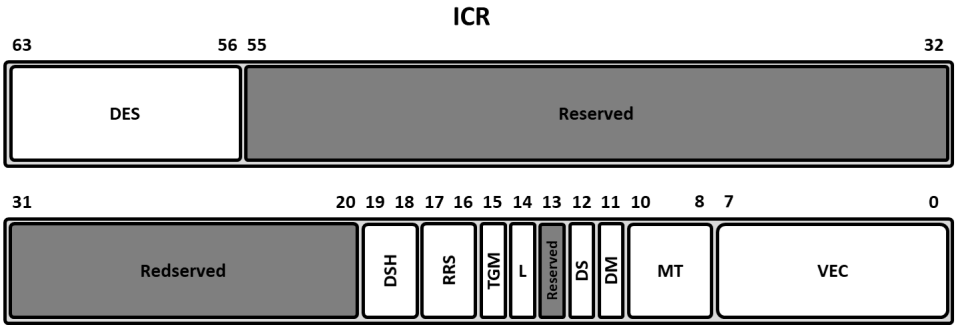

Figure 5.2. Interrupt command register.

a 4KB memory-mapped region that can only be accessed when running with the highest privilege level. This means that a thread can program this register only when it is actually running in kernel mode. Thus, all the logic for reading from and writing to the ICR register must be implemented within a specific function of the kernel module that is possible to call from user-space, possibly passing the identifiers of all the CPU-cores that are intended to be hit. To cope with such a problem, we decided to implement a new system-call directly within the kernel module, which is then hooked to the system-call table when the module is installed into the OS. In Listing 5.1 is reported the system-call that every thread can invoke when running in user mode in order to force the delivery of an interrupt to one or more CPU-cores. The handler that shall be activated will depend on the value of the ipi_vector parameter.

Listing 5.1. Programming of the Interrupt Command Register.

```
static inline long
__do_sys_send_ipi_syscall(unsigned long cpus_bitmap) {
    // bitmap of target CPUs
    struct cpumask cpus_mask;
    // bitwise AND of the input bitmap
    // with the bitmap of active CPUs
    int not_empty = cpumask_and(&cpus_mask,
        to_cpumask(&cpus_bitmap), cpu_online_mask);
    // checks if the result is non-empty
    if (not_empty) {
        // relies on the send_IPI_mask function pointed
        // by the per-cpu apic variable to send IPIs
        apic->send_IPI_mask(&cpus_mask, ipi_vector);
    }
    return 0;
}
```

Once invoked, this system-call immediately checks whether the intersection of the bitmap passed as input with that of the currently active CPUs is not an empty set, and if so it is then possible to send IPIs to all the CPU-cores belonging to the intersection set. It is possible to notice that we relied extensively on services and variables that are commonly used in kernel-space to assist the execution of several activities performed by the kernel itself, whose symbols have been exported by the OS code to make them visible to the kernel modules as well. Among these, there is the cpu_online_mask variable which is a bitmask initialized at boot of

the OS to indicate what CPUs must be considered by the kernel for scheduling threads, and which can be updated during its operation when, for example, the CPU hotplug support provided by the OS detects new physical nodes installed in a NUMA system. We want to point out that having relied on these services has not only led to write elegant functions that match the coding style of kernel sources, but it has also allowed us to rely on already implemented logics to deal with certain problems that can always arise when carrying out these activities—this mostly involves the preemptibility of kernel threads and their sensibility to the arrival of interrupts while they are performing operations related to the management of sending IPIs. Last but not least, it exploits per-architecture specific optimizations, such as those implemented within the send_IPI_mask function, which have been devised to efficiently send multiple IPIs to different CPU-cores at once—this mainly regards groups of CPUs that share a common logical ID.

However, hooking up a new system call by installing a kernel module in newer versions of the Linux kernel is no longer as easy as it was in older ones. In fact, in past versions of the kernel there was an explicit variable for the system-call table that has been later removed for obvious security reasons, so that we had to find new ways to install our system-call. In this regard, the main difficulty was precisely to guess where the table actually resided, which was further exacerbated by the randomization of the address space layout (ASLR) of the kernel in addition to the lack of any symbols that could lead to discovering the address of the system-call table. To overcome this obstacle we have implemented a hacking strategy that relies on the Linux kprobe service to retrieve the address of a system-call that has not been blacklisted by this service. With this value in the hand, and by knowing the address of the well known sys_ni_syscall which recurs several times into the system-call table, it was possible to implement a pattern-matching procedure to intercept the desired address, even because the size and position of the relative entries within the table are always known—according to the standard defined by the Portable Operating System Interface for Unix systems (POSIX). This procedure consists in seeking within the address space of the kernel up to when data read in memory match the compared pattern, always taking care to verify that the virtual pages accessed are actually valid. At this point we can appropriate of an entry previously occupied by the sys_ni_syscall that is now overwritten with the address of our new system-call. It must be noted that this approach is perfectly compatible with the recent Page Table Isolation (PTI) patch [31] designed to counteract security attacks based on hardware-level speculation, like Meltdown [50] and Spectre [45]. In fact, our new system-call is still dispatched by the original dispatcher used for all the other system-calls which already implements all the logic related to this patch at the beginning and end of the assembly stub that is entered upon the execution of the syscall instruction.

Anyhow, we have not yet discussed how IPIs are handled on the receiving side. In this regard, the interrupt delivery protocol provided for handling IPIs of type Fixed does not stray too far from the procedure followed for the management of local interrupts. In fact, unless interfacing with the system bus, the firmware embedded into the LAPIC uses the vector number delivered along with the IPI to displace within the IDT table in the same way as if the interrupt were issued from another source. Hence, the handler that will be activated upon receipt of the IPI will depend

on the vector number that has been specified by the sender that, analogously to what we did for the IBS interrupt handler, has been set to the value corresponding to the offset of the interrupt-gate descriptor within the IDT table that holds the address of the spurious_interrupt handler. Therefore, for the same reasons discussed in Chapter 4 regarding the way by which the IBS interrupt handler is installed into the OS, the installation of the IPI handler also occurs by replacing the call to the smp_spurious_interrupt function with a call to the handler provided by our kernel module. This is accomplished again by performing binary inspection of the spurious_interrupt procedure whose address is always known as it is stored in the aforementioned descriptor within the IDT table. We want to make notice that this procedure is compatible with all the Linux kernel versions and with the PTI patch included into the more recent ones, regardless of whether the logic associated with this patch is made active at the boot of the OS.

5.1 Effective Management of Causality Errors

The term simulation commonly refers to an approximate imitation of the operations describing the evolution of a system over time. It is the process of modelling a real system and conducting experiments for the purpose of either understanding the system behaviour or evaluating various strategies for the operation of a system. Simulation can then be used to show the eventual real effects of alternative conditions and courses of action, especially when the real system of interest cannot be engaged because it may not be accessible, or it may be dangerous or unacceptable to engage, or it may simply not exist. The simulation has historically been used in many fields of application, among which we mention the simulation of technology for performance tuning and optimization, scientific simulation of natural or human systems to gain insight into their functioning, and the simulation of financial markets of particular interest to the economy. Hence, the simulation has had numerous positive implications in many application contexts, in each of which it has been implemented according to different methodologies and by following different approaches. This in turn confirms that the simulation cannot be reduced to a single technique, but differs for different classes of simulation models. There is the class of the continuous simulation which is the representation of the evolution of systems based on continuous time, which usually involves dealing with numerical integration of differential equations. The class of stochastic simulation concerning the simulation of systems where some variables or processes are subject to random variations, thus meaning that replicated runs are likely to produce different results. Then there is the class of discrete-event simulation (DES) which studies systems whose states change their values only at discrete times, as a result of a discrete sequence of events that occurred in time. Between consecutive events no changes is assumed to occur, hence the simulation can directly jump in time from one event to the next. A DES model therefore represents a system as a set of connected entities that execute and communicate through the exchange of events. By the way, the process of modelling a physical system begins by decomposing the system itself into a finite set of components, also known as simulation objects, that logically represent the physical processes of the system being simulated.

However, although DES is in principle an intuitive and easy to understand simulation model, is surprisingly difficult to parallelize in practice. We refer to the parallel discrete-event simulation (PDES) model which provides for the execution of a single DES program on a parallel computer. Similarly to DES, modelling a system in PDES requires to identify the physical processes to simulate, each one represented by a different simulation object. In addition, PDES also provides for mapping each simulation object to a different logical process (LP)—it is effectively a separate execution context for each simulated physical process—which can potentially be charged to a given process or thread to run in parallel with the others. This is why PDES has always been a highly interesting topic, just because it represents a problem domain that often contains substantial amount of parallelism that can be exploited by parallelizing the execution of some events. On the other hand, concurrent executions of events at different points in simulated time introduces interesting synchronization problems that are at the heart of the PDES problem. In fact, a direct port of the DES model to parallel computing quickly runs into difficulty. This can be easily inferred by the fact that a single execution of an event destined to some simulation object directly or indirectly affects the initial execution state of all events that follow this one along the simulation path of the same logical process and that of the others, since its processing can give rise to the scheduling of one or more events in the simulated future in order to model the causality relationships that characterize the evolution of the system being simulated. Given such a data-dependent nature of DES programs, it is crucial to always select the oldest event for the execution since selecting a more recent one would amount to simulating a system in which the future can affect the past. The latter can always arise in PDES whenever the execution of two events with a happens-before relation [46] defined between them does not follow the correct order, which unavoidably leads to generating an error that is commonly referred to as *causality error* [24] or violation.

To cope with the aforementioned problem several paradigms for the execution of PDES programs have been presented in literature, among which the Virtual Time [40] paradigm has established itself as a reference one with its well known implementation called Time Warp mechanism. A virtual time system is defined as a distributed system enabled to perform operations in coordination with a virtual clock that ticks the virtual time. As for the virtual clock, this is not an object instantiated within the system that can be accessed by processes or threads to monitor the advancement of the simulation time as a whole, rather it abstracts completely from real time and its way to flow. Also, it is independent of the virtual clock that can currently be seen by any other logical process. A local view of the virtual clock is indeed used by a logical process to assign a virtual time coordinate to each new scheduled event, namely its timestamp, in order to determine its position along a common virtual time axis along which all the events that contributes to the evolution of the simulated system are placed. It is therefore indicative of the order according to which each event should be performed with respect to any other event that can be reached along any possible path made up of edges representing causality relationships and that connect these events. Any two events connected through a similar path are constrained to be processed in real time by respecting the same order imposed in the virtual time so as not to violate the causality relationship defined between them.

Differently, for all the other couples of events, namely those that are not subject to a happens-before relation, there is no need to respect any execution order in the real time whatever their timestamp. This is precisely where the opportunity to exploit parallelism lies, in the possibility to carry on the execution of unrelated events on parallel hardware. However, the events scheduled for whichever logical process are generated at run-time and marked with timestamps by possibly different logical processes charged of the execution of other events that have smaller timestamps than those of the former, still according to the happens-before relation. This means that causality errors can still occur during the execution of some logical process which has already hazarded the execution of an event that is virtually beyond the current lookahead—it is a simulation parameter that determines the ability to predict at a certain simulated time C all the events that will be generated up to a simulated future time $C + L$ with complete certainty, where L represents the lookahead. In these conditions, the logical process for which some thread is speculatively performing an event that belongs to the simulated future can be subject to the arrival of a another event that is virtually in the past with respect to the one currently executed. The latter are the so-called *straggler* events, whose arrival is the cause of causality errors. Nonetheless, the Time Warp mechanism is an optimistic implementation of the Virtual Time paradigm which has been appositely designed to cope with this kind of errors by undoing the effects of speculatively performed events. It is said to be optimistic because, in contrast to systems that use some kind of block-resume mechanism to keep logical processes synchronized, Time Warp relies on general lookahead-rollback as its fundamental synchronization mechanism. In fact, each logical process is allowed to speculate over the execution of events that belong to the simulated future, regardless of whether causality conflicts may arise. When a causality error is finally detected, the involved logical process is immediately rolled back to a virtual time preceding that of the event that has generated the causality conflict, so as to retry forward execution along a revised path. However, although the Time Warp mechanism provides for the presence of many loosely synchronized local virtual clocks that occasionally jump backward to solve causality errors, from the point of view of the PDES application's semantics there is a single global virtual clock (GVC) that always progresses forward at an unpredictable rate with respect to real time—it is the commit horizon of the simulation run, before which the effects of all processed events have been made permanent—and which determines the speed with which the simulated system evolves.

One can immediately notice how the rate by which the GVC seen by the programmes advances in real time is a first important performance index for the simulation itself. It is indeed clear that different values of the rate for the same PDES program are representative of different throughputs of committed events. Behind the latter can anyway be present a series of processed but not committed events, due to causality errors, whose number is also a performance index for the execution of a PDES program. More precisely, the ratio between the number of committed events and the total number of processed ones is an useful information to understand if and why the observed throughput is lower than it could potentially be. It is indeed known that higher parallelism degrees do not always correspond to higher throughput values, since more and more causality errors are likely to occur when approaching too high values of concurrency [69]. These errors must therefore be

tackled by rolling back one or more logical processes to branches of their simulation path that are still deemed correct. In turn, such rollbacks possibly lead to undo the sending of previously created events that were intended for other logical processes which might need to rollback accordingly. The latter phenomenon is also known as *cascading* rollback and it is particular adverse to performance of discrete-event simulations that run on PDES systems based on the Time Warp protocol.

Several works have been presented in the literature to address the issue of reducing the negative incidence of rollbacks on the performance of speculative PDES systems. Among these there are some proposals [7, 8, 36, 86] aimed at well balancing the workload across concurrent threads with the idea that diminishing the divergence in the advancement of concurrent logical processes in virtual time provides the expectation of a reduction of the frequency of causality errors. Other works have instead presented solutions based on either throttling [82] or bounding speculation via synchronization schemes that are not purely optimistic [12, 61]. While the first type of approaches provides for the introduction of artificial delays in the execution of some logical processes that would otherwise speculate too much in the simulated future, the second type involves the imposition of temporary blocks on the execution of events that are too far ahead in virtual time. Anyhow, both approaches aim to prevent some logical processes from executing events that are too far from the commit horizon, and for this reason prone to causality errors. Another solution which tackles the issue of reducing the impact of causality errors, and the associated waste of resources, has been proposed in [53]. This approach is essentially based on message broadcasts into group of simulation objects as a sort of signaling mechanism to notify that some event, which can have generated a chain of other events destined to the group members, is no longer causally consistent, thus meaning that the chain of events is no longer consistent too. The broadcast will then allow the logical processes that have processed at least one event belonging to the chain to early rollback.

Then, there are other works in which the authors have proposed solutions to reduce the overhead costs that are inevitably introduced by the management of the state recoverability of logical processes that rollback, with the aim of leading the PDES system to obtain higher performance results from the execution of simulation programs. These typically involve either checkpointing strategies [67, 65] or reverse execution [47] in order to restore snapshots of the execution state of simulation objects that are in the virtual time prior to the events that have caused causality errors. As for checkpoint-based recoverability, some solutions provide for logging the whole state of a simulation object at each event execution or after an interval of executed events, while other solutions implement the incremental logging of only the modified state portions so as to reduce the amount of writes to perform in memory. Overall, these are an attempt to find out a combined cost optimization for rollback activities and for all the activities that enable correct rollbacks. As for state recoverability based on reverse execution, it essentially relies on the generation of logs of instructions used to cancel the effects of the operations performed by the events that are doomed to rollback, thus restoring the old values that formed the state being recovered. This approach clearly gives the advantage of recovering exactly the state preceding the virtual time of the event that has caused the causality error, thus not requiring to rerun any events that are still consistent. On the other

hand, it requires to collect the logs of at least all instructions that update memory locations when the events are executed in forward mode. A hybrid solution was presented in [9] to get the best of the two philosophies, according to which the PDES system is allowed to query a cost function at run-time to determine what is the best combination of checkpoint interval and the subset of events for which to collect logs of instructions.

Another way of limiting the overhead caused by rollback is based on adopting smart strategies for selecting what event should be dispatched along a process or thread as soon as the latter completes the processing phase of another event. Lowest-timestamp-first scheduling (LTF) [49] is a classical reference, which is however agnostic of the CPU-demand by the events. Therefore, it can lead to suboptimal choices in scenarios with non-minimal variation of the CPU-time required to process events that have very close timestamps. On the contrary, the proposal in [68] takes into account the event granularity, and does not favour coarser grain events that have timestamps falling in a given (short) virtual time window left delimited by the lowest-timestamp pending event. This method is essentially based on less promptly starting the execution of events that, once CPU-dispatched, might produce a longer CPU-burst of activities that can in any case be invalidated by a causality error.

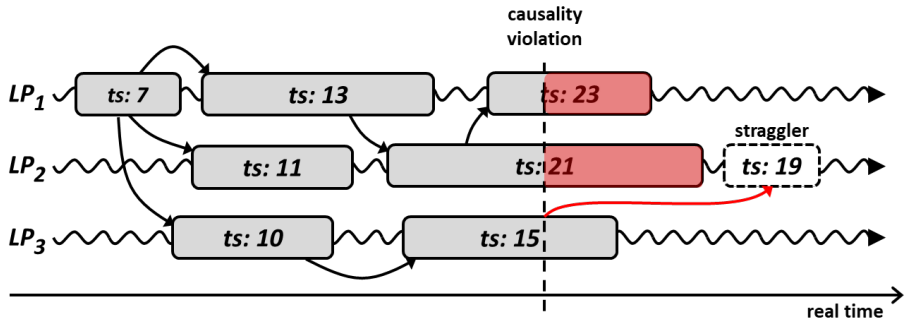

Figure 5.3. Time and resources wasted (red) after a causality violation occurs.

However, none of the solutions belonging to the optimization classes discussed above is worried about the waste of resources resulting from the execution of events already doomed to be rolled back due to causality errors. In Figure 5.3 is reported the graphical representation of an event execution scenario which is typical for PDES, whereby some thread speculatively takes in charge the execution of an event intended for the logical process LP_2 with timestamp 21 while some other thread charged of the execution of another event with timestamp 15, which was destined to the logical process LP_3, schedules a new (straggler) event that is in the past in virtual time with respect to the one currently performed for LP_2. The latter is therefore no longer causally consistent and its execution should be immediately abandoned by the involved thread in order to early approach to the unavoidable rollback phase instead of wasting computing resources (depicted in red) for the remaining time required to complete it. The reason why this execution dynamics do not take place is clearly related to the inability of PDES systems to promptly react to the need of rolling back causally inconsistent events. In fact, processes or threads

that are charged of the execution of such events continue running up to their end, where the PDES runtime will be able again to regain control of the execution and will have therefore the possibility to rollback these events. It is clear that nothing can be done to recover the time already spent in performing work that suddenly becomes intended to be discarded. This does not anyway means that we cannot do something to prevent the execution of events that are no longer causally consistent from lasting too long over time. The latter scenario is further exacerbated by the fact that these events can also schedule new events during their residual execution, thus spreading non-consistent work which will clearly require to be undone in the approaching rollback phase—this takes place by sending anti-messages with the aim of annihilating the previously scheduled events, which may or may not be already processed in the meantime. So, to avoid a further waste of resources and time, the PDES system should be assisted with adequate software and hardware supports that would make it possible for threads to early rollback the execution of events that are no longer causally consistent, prior the time at which such an action was expected to take place.

To this end, the ULMT technology would prove to be very useful as it would allow to promptly interrupt the event processing phase that some thread is carrying out on behalf of some logical process as soon as a causality error arises, in order to lead the same thread to early perform the rollback phase. With the aid of the ULMT technology, all the events destined to a logical process can be executed within the micro-thread context that has been reserved for the logical process itself, hence all these events can be subject to eventual asynchronous control flow variations (CFV) while they are being executed. The latter would then lead to perform a routine specifically devised to check whether a causality error is actually occurred, and if so to rollback all the events belonging to the no longer causally consistent branch of the simulation path. Also, it turns out that the interrupt-driven approach we discussed at the beginning of this chapter is the ideal approach to support the execution of logical processes with a preemptive form of scheduling, that is, the approach that would make it possible to promptly interrupt the event execution carried on by some thread exactly when a program state change, which might have affected the execution state of the involved logical process, is occurred. It is clear that, in the context of PDES, this coincides with the send of events that might have led to the materialization of causality errors along the simulation path of the target logical processes. In this scenario, the thread that is scheduling a new event on behalf of some logical process may also send an IPI to the thread that is currently performing an event for the logical process intended to receive the newly created event, if any. In this way, the target thread is induced to temporary pause the event execution so as to check whether the involved simulation object has received another event that is virtually in the past with respect to the former. If so, it will be possible to early rollback by preventing the thread from spending additional time and resources in executing an event that is no longer consistent.

The only work that to our knowledge has proposed a solution which introduces the possibility of preempting the event processing phase is the one in [64]. In this work, a preemtion mechanism has been specifically implemented to enable prompt switch of a simulation object—the one for which a given thread is currently performing one of its events—with another simulation object to which a higher

priority event has been (very) recently delivered, namely the one with the lowest-timestamp among those currently bind to the involved thread, as required by the LTF scheduling strategy. Similarly to what the micro-threading model provides for the execution of tasks, in this work logical processes are provided with execution contexts appositely designed to make them preemptible at any time during the event processing phase. The preemption mechanism is then based on a polling scheme actuated via the exploitation of LAPIC-timer interrupts which allow the involved threads to periodically check whether other simulation objects have received a higher priority event. This is accomplished by partitioning into multiple fine-grain intervals the system tick interval assigned to each thread by the OS, thus enabling the LAPIC-timer to issue interrupts with a higher rate.

However, although this solution has some features in common with the micro-threading model, more precisely with the ULMT technology, the creation and management of separate execution contexts follow the less performing procedure presented in [17]. As hinted, the latter is a portable but extremely heavy solution that relies on specific facilities offered by the POSIX signals specification in order to arm a signal handler that will perform by using an alternate stack. The deriving execution context will then be assigned to a simulation object. This means that the preemptive architecture at the basis of this solution is not suitable for setting up new contexts at run-time—this is a general consideration about the technology implemented for supporting the execution of preemptive tasks, since this feature is not actually required for the purposes of this work because the number of simulation objects is fixed—hence not even suitable for accomplishing the commissioning of a micro-threading-like execution model for activities belonging to those applications that are characterized by very irregular parallelism, and for which the decomposition of work takes place dynamically according to the task-parallelism paradigm followed by many modern parallel programming models. As we have extensively discussed in previous chapters, supporting this type of parallelism in order to experience new and more reactive form of task scheduling is instead the strong point of the more sophisticated ULMT technology. Furthermore, the partitioning of the system tick interval by reprogramming the LAPIC-timer is no longer a viable solution in the more recent versions of the Linux kernel as a common clock event source can be used by the kernel to schedule both high resolution timer (*hrtimers*) and periodic events [29], thus making difficult to bypass the clock event distribution system without affecting the timing according to which certain activities are expected to be carried out by the OS. Differently, we have always relied on hardware that does not interfere with the functioning of the OS or that could be used in sharing with it (*e.g.*, the IPI mechanism). Last but not least, a solution based on periodic arrivals of LAPIC-timer interrupts in order to check the presence of events with a lower timestamp shares the same limitations that affect the time-based approach for supporting the preemptive execution of micro-threads. In this regard, it cannot show an arbitrary fine granularity as the polling overhead would reveal unaffordable, nor it can rely on larger granularities as they would tend to reduce the benefits in performance due to higher reaction latencies. Moreover, just a reduced percentage of polls will actually reveal useful in the early detection of events with lower timestamps.

On the contrary, the proposed interrupt-driven approach, which is based on the exploitation of the IPI mechanism supported by conventional chipsets by all the major

vendors (Intel, AMD and ARM), makes it possible to induce interrupts on remote CPU-cores with almost zero delay and only when it is actually needed to occur, that is, when any event executed for the same logical process targeted by the newly created event is currently dispatched on some CPU-core. It results therefore that our approach is particularly suited for speculative PDES that run on top of multi-core shared-memory machines for which additional optimizations aimed at reducing the overhead introduced by the procedures implemented for sending the IPIs and for handling those delivered to the recipient side are possible, and which we will discuss extensively in the following subsection dedicated to presenting the implementation design of a new preemptive PDES architecture based on ULMT technology, thus exploiting the micro-threading concepts along with the interrupt-driven approach in order to improve performance of PDES.

5.1.1 IPI-based Virtual-Time Coordination in Speculative Parallel Discrete Event Simulation

We have already discussed how the classical parallelization methodologies for DES involve dividing a complex model into multiple simulation objects that interact via cross-exchange of timestamped events. Consequently, the implementation of PDES platforms adhering to these methodologies has been historically based on explicitly assign groups of simulation objects to different threads. This was also dictated by the fact that, in order to scale with performance, PDES ran in parallel on distributed systems where the exchange of events was accomplished through message passing. Thus, it was usual that a certain number of simulation objects were temporarily assigned to processes or threads running on different nodes of a distributed system. Such assignments were then subject to periodic rebinding actuated in order to keep the resource usage well balanced on the medium long term [7, 66] according to the classical approach for which the workload is migrated towards the available computing power.

Anyhow, with the advent of multi-core shared-memory machines new implementation trends for PDES platforms have arisen, which are based on the new concept of migrating the computing power towards the workload. Reinforcing this concept is the idea that all threads running the PDES platform can fully share finer grain work units, namely individual events possibly bound to different simulation objects, thus relieving the PDES engine of the need to deal with load balancing issues. This is the share-everything PDES paradigm [37] which provides for the presence of a unique queue keeping the events destined to whichever simulation object [56], from which any thread picks its next event to process. Such a paradigm has therefore the intrinsic advantage of enabling threads to always select the events that are closest to the commit horizon of the simulation, thus reducing the probability of experiencing rollbacks at run-time. By the way, the Ultimate Share-Everything (USE) [36] simulator is precisely the PDES system that we have chosen to integrate our solution aimed at supporting the preemtive execution of logical processes according to the interrupt-driven approach. As its name suggests, it is a highly optimized PDES engine for multi-core shared-memory machines that adheres to the aforementioned paradigm. USE has been precisely designed to provide non-blocking progress in both virtual and real time which is guaranteed by the exploitation of

fine-grain synchronization techniques that confine critical sections to the execution of individual atomic instructions, thus ensuring scalability while accessing shared data and metadata within the simulation engine. With the aid of such techniques to mediate the non-blocking accesses to the single queue of events, it is guaranteed that the computing power is always assigned to the processing of events with the lowest timestamps, thus resulting in a reduced number of causality violations and consequently an improved overall efficiency. By the way, each simulation object in USE is bound to a given thread only in the short term, which is the time required to complete the execution of a single event, after which the binding can be re-evaluated. Therefore, adopting USE as the base system allows us to evaluate benefits and shortcomings of our approach in a worst case scenario, namely when it is deployed within an engine characterized by a generally low incidence of rollback and, hence, to be rolled back work.

We want to point out that the solution we have designed and integrated within the USE system in order to realize a preemptive PDES architecture relies on the same user- and kernel-level facilities described in Chapter 3 which form the ULMT technology. In addition to these, new software components, such as the ones aimed at exploiting the IPI hardware facility that we have presented at the beginning of this chapter, are used to complete the commissioning of the micro-threading model for the execution of logical processes together with the interrupt-driven approach implemented for obtaining asynchronous interruptions of the processing phase of events at arbitrary points of their execution, and only after the occurrence of noteworthy simulation state changes that affect the involved logical processes. This means therefore that our solution is fully transparent to the application level code, thus allowing the programmers to not care about how to deal with preemptive logical processes, hence with micro-threads, since the management of their execution is fully charged to the ULMT-based implementation of the USE runtime. In Figure 5.4 we provide a representation of how the above stated software components are involved in the process of notifying a target thread of the occurrence of a causality error. Here, the invocation of the `send_IPI` system-call along the execution path of a thread other than T allows to trigger an IPI interrupt to the CPU-core hosting the thread T. The latter then leads to the activation of the interrupt handler in charge of setting up a CFV for the involved thread T, so that the same thread T is now able to perform the early rollback just after having made some preliminary checks.

Given that this solution is particularly suited for speculative PDES that run on top of shared-memory machines—although nothing prevents this solution from being used to optimize intra-node execution dynamics of larger distributed systems composed of multiple multi-core shared-memory machines—additional user-space facilities and metadata shared by all threads have been integrated within the USE runtime with the sole purpose of allowing the implementation of specific optimizations aimed at containing the overall overhead costs for sending and handling IPIs. In particular, a first optimization concerns the capability of threads that are scheduling new events on behalf of some logical processes to determine whether the logical processes targeted by these new events are currently taken in charge by some thread. The latter prevents such a threads from sending IPI towards any CPU-core if none of the events belonging to the target logical processes are currently executed at that time. Differently, a logical process for which at least one of its events is currently

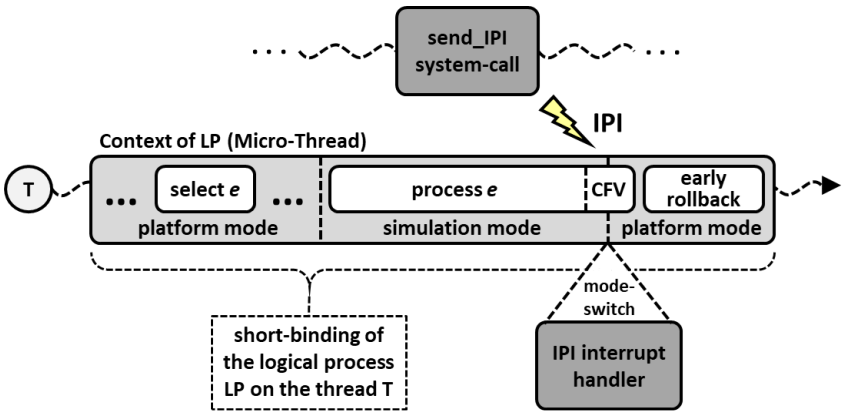

Figure 5.4. ULMT technology to serve early rollbacks.

CPU-dispatched is also candidate for receiving an IPI. In this regard, a second optimization concerns the capability of threads that are scheduling new events on behalf of some logical processes to determine whether the execution of the logical processes targeted by these new events will actually be made no longer causally consistent, that is, these events have smaller timestamp values than those of the events currently performed by different threads for the involved logical processes. This makes it possible to decide if it is actually needed to send an IPI along with a scheduled event, since it would result unfruitful to interrupt the execution of a thread to make it aware of the presence of a new event that belongs to the simulated future. On the contrary, a new event that is in the past in virtual time is also the cause of a causality error and for this reason it is necessary to promptly interrupt the processing phase of the no longer causally consistent event in order not to waste time and computing power. As a final optimization, we avoid the send of multiple unneeded IPIs in scenarios where several threads concurrently schedule new events destined to a common logical process and which are virtually in the past with respect to the one currently performed by some thread for the same logical process. Also, we make the thread hit by this single IPI aware of which event has the smallest timestamp among all the simultaneously scheduled ones, so as to determine what is the most recent state of the simulation object to restore that is still causally consistent. It is therefore clear that with all these optimizations the advantage of our proposal, compared to preempting the execution of a no longer consistent event in the basis of a polling scheme as it would have been if we had relied on the time-based approach or as it is with the solution proposed in [64], are multiple:

- we pay the cost of hardware level cross-CPU-core coordination only if the CPU-core, which is the destination of the IPI, is actually running inconsistent work, that is, when a causality error is surely occurred along the thread running on that CPU-core;
- the interrupt-drive approach based on the IPI hardware facility allows for an almost immediate preemption of the doomed to be rolled back work, since the delivery of the IPI to the CPU-core that is running the causally inconsistent event takes place in very few microseconds;

- we reduce the effect of the event-preemption support on the locality of the access to memory as the activation of the CFV procedure when it is in effect not needed is unfavourable to locality.

The shared metadata structure that has made it possible to implement all the optimizations, hence the functionalities described above, is shown in Figure 5.5. This represents a per-simulation-object *control table* which is accessible by all threads to identify the CPU-core that is currently hosting the thread—let's recall that the micro-threading model provides for pinning threads to CPU-core, hence the binding is well known at run-time—which has taken in charge the logical process of interest, if any. It also keeps track of the event that is currently performed by the thread on behalf of that logical process, as well as its timestamp. In this regard, the control table makes it possible to determine if some other concurrent thread has already detected this event to be no longer causally consistent, and has taken care of triggering the IPI send towards the destination CPU-core. Its fields are as follows:

- the core_id field is the numerical ID—according to the ACPI indexing—of the CPU-core that is currently running an event destined to this simulation object;
- the timestamp field denotes the timestamp of the simulation event that is currently being processed for this simulation object (o0r the one that needs to be processed after this simulation object has finished a rollback phase);
- the event_type field denotes the type of the event being processed;
- the start_time field keeps the wall-clock at which the involved event has been started processing—by reading the timestamp-counter-register (TSC) via the rdtsc machine instruction;
- the already_hit field is used to denote if some thread has already revealed that the current event is no longer causally consistent;

With each simulation object O we associate a pointer $\mathsf{CT}[O]$ into the array CT, which can be either `NULL` or can point to O's control table. When a thread picks the event e to be processed at the simulation object O then it initially fills all the fields of the control table, and right before invoking the event handler associated to e, it updates the pointer $\mathsf{CT}[O]$ to point to the same control table in a fresh incarnation of its content. It is important to point out that the latter operation takes place via an atomic memory update instruction—this is a compare-and-swap (CAS) x86 instruction named cmpxchg—which essentially acts as a memory-fence between all writes within the control table and the effects of e processing. This ensures that the control table has a consistent content visible to all the other concurrent threads as soon as they catch a non-`NULL` value of the $\mathsf{CT}[O]$ pointer. Also, the thread that updates the control table after picking the event e is able to publish towards all the other threads the control table by avoiding lock protected critical sections, hence enabling scalability. Clearly, as soon as the thread finishes working on the event e, it resets the pointer to the value `NULL` still via a CAS instruction, while the buffer hosting the no longer active control table can be collected for later re-usage according to the well-known schemes, such as Epoch-Based Reclamation [23] or

Hazard Pointers [57]. As hinted, the shared-memory nature of our target speculative PDES system enables any other thread that has processed an event which generated some other event e' in the past of the event e'' currently being processed at the simulation object O to know that the causality error has happened. In fact, this can be simply tracked by comparing the timestamp of e' with the value registered in the timestamp field of the control table associated with the simulation object O. In this case, the thread that produced e' attempts to update the already_hit field in the control table via a CAS machine instruction. If it does not fail, it means that this thread is the one that needs to notify the causality violation to the destination CPU-core, hence the pinned thread, via the apposite system-call that we have introduced as part of the ULMT technology for the commissioning of the micro-threading model based on the interrupt-driven approach. Otherwise, a failure in the CAS indicates that some other thread already notified the destination CPU-core of the causal inconsistency via an IPI, and there is no need to resend the IPI in order to preempt the processing of the no longer causally consistent event.

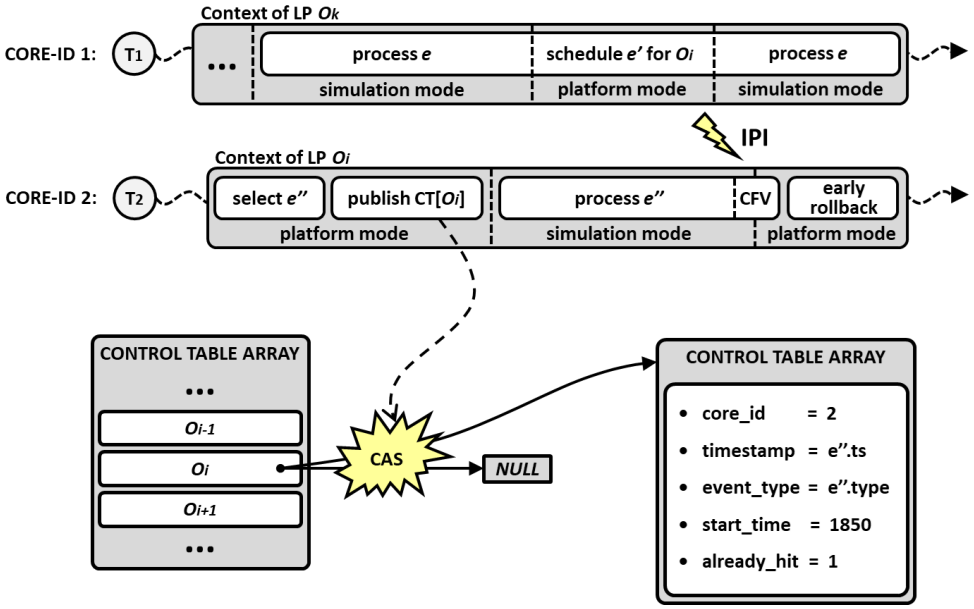

Figure 5.5. Publication of the control table.

As soon as an IPI is delivered to the target CPU-core, hence to the thread that was carrying on the no longer causally consistent execution of some event when the IPI was issued, the activation of the proper interrupt handler leads the interrupted thread to be subject to CFV which in turn leads to the execution of the CFV_TRAMPOLINE_ROLLBACK function by the thread. The pseudocode of the latter is shown in Algorithm 6 which is basically the CFV_TRAMPOLINE function presented in Chapter 3 with the difference that it includes some additional checks which are immediately evaluated upon return from an IPI interrupt, before starting the rollback phase. These checks are intended to avoid the passage of control to the rollback routine, which is a function that never returns the control to the CFV trampoline function, when it is indeed not needed to abandon the current execution flow. This is for example when the IPI arrives late at the destination thread, with

respect to the processing of the event to be hit, so that the thread must first control if an event is currently being processed (line 5 in Algorithm 6) and consequently if the execution of that event has been detected as causally inconsistent by another thread (line 7 in Algorithm 6). If both conditions hold, the logical process has to rollback all speculatively performed events that follow the newly created event in virtual time. Otherwise, the interrupted execution flow is simply resumed. One may also notice that in this implementation of the CFV trampoline function there is no saving of the current execution context with respect to the original version. Rather, a previously saved execution context of the involved logical process, hence of the corresponding micro-thread, is restored so as to perform the rollback phase in a safe manner (line 14 in Algorithm 6). In fact, for each logical process participating to the simulation, a single snapshot of its execution state was taken at the beginning of the simulation, at a point where each one of them can safely carry on the rollback procedure out of any event processing phase, namely when the thread charged of the execution of a given logical process is running in platform mode—this mode of execution characterizes the operations that a thread performs on behalf of a given logical process at the level of the PDES runtime, thus unrelated to functions of the simulation model. Once that micro-thread context is restored, the control flow that characterizes the execution of the CFV__TRAMPOLINE__ROLLBACK function is abandoned and never resumed in the future, hence there is no need to preserve this execution state, even because it is representative of a no longer causally consistent execution branch of the simulation. The restored execution context leads therefore to safely complete the rollback procedure and to terminate the short-binding of the logical process to the involved thread, which can then context-switch in favour of the simulation object for which has been retrieved the event with the lowest timestamp, possibly the same as before.

Algorithm 6 CFV trampoline for early rollback.

```
1:  procedure CFV__TRAMPOLINE__ROLLBACK:
2:      SWITCH__STACK()
3:      lp ← GET__CURRENT__LP()
4:      ct ← GET__CONTROL__TABLE__ARRAY()
5:      if ct[lp.id] == NULL then                          ▷ no event being processed
6:          return
7:      if ct[lp.id].already_hit == 0 then                  ▷ still causally consistent
8:          return
9:      pc ← READ__PREEMPTION__COUNTER()
10:     if pc > 0 then                                  ▷ non-preemptible code region
11:         return
12:     ct[lp.id] ← NULL                    ▷ reset the reference to the control table
13:     MtSnap ← GET__CPU__SNAPSHOT()
14:     CONTEXT__RESTORE(MtSnap)                          ▷ from where rollback early
```

Moreover, after the control table has been published, it is possible that the execution flow will pass through functions, which are called by the event handler, that need to be executed according to an all-or-nothing semantic. A typical example is represented by the memory allocation functions managed by (recoverable) memory allocators, which need to correctly manage metadata in a non-preemptible manner, otherwise the memory allocator would be left in an inconsistent state and therefore

the PDES system would not be able to guarantee rollback-ability of the allocation/deallocation operations. Here, the preemption_counter variables we discussed in Chapter 3, which we regularly use to prevent micro-threads from being preempted while they are running in critical sections or within non-reentrant functions (line 10 in Algorithm 6), play again a crucial role. For all those functions which need to be executed in a non-preemptible manner in order to ensure correctness of the simulation, we offer wrappers such that at the entrance and at the exit of whichever of them the relative preemption_counter variable is respectively incremented and decremented. In this case, when an IPI hits the currently processed event, the routine to which is given control downstream of the CFV is charged of the responsibility of verifying if the event can be immediately squashed. This is done by checking if the preemption_counter variable is currently set to zero as it is normally performed by the trampoline function that is part of the ULMT technology. If the check is negative, it means that we are currently running within one or a chain of calls to non-preemptible functions and we cannot squash the event execution immediately. However, the function-return wrapper is structured in such a way to check, after having decremented the preemption_counter variable, if its value is zero and if the already_hit field in the control table has been set. If both checks are true, it means that the event is no longer causally consistent and the execution of the same activities that would have been performed asynchronously with the CFV procedure now takes place synchronously just upon the return from the currently executed non-preemptible function. A scheme where this deferred squash of a no longer consistent event is adopted is provided in Figure 5.6. If on the one hand such approach is less timely in terms of readiness to react to the arrival of events that have invalidated the causal consistency of the ongoing event processing phase, on the other hand it prevents the delivery of an IPI from being completely lost and therefore it allows to save the whole remaining portion of the doomed to be rolled back work.

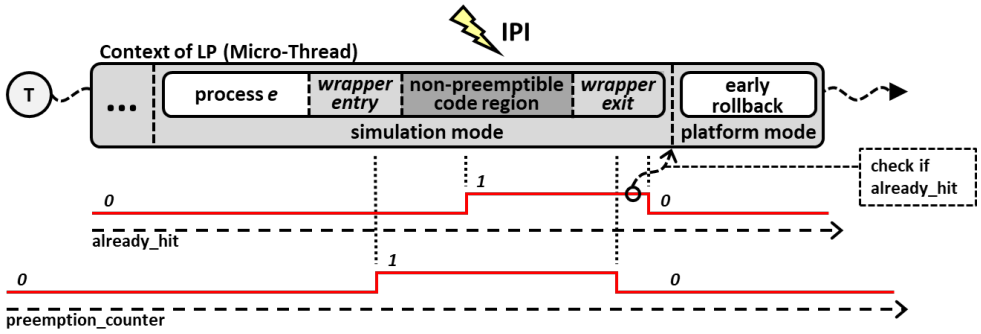

Figure 5.6. Deferred execution of the early rollback phase.

An additional relevant point in our solution has been the introduction of a runtime decision support to determine the actual usefulness of sending the IPI towards a CPU-core that is currently running doomed to be rolled back work. Clearly, such usefulness depends on several factors, among which we can consider the following:

- the expected granularity of the event currently being processed, whose execution was found to be no longer causally consistent;

- the expected residual processing time of the event stated above, calculated from the time when the runtime decision support is queried.

Even though the two metrics above are somehow correlated, they have been exploited in differentiated manners in our solution. In more detail, we keep track of the average CPU-time required to process events of any given type, as defined by the simulation model programmer. This is done at the PDES runtime level by relying on the timestamp counter (TSC) register and the x86 instruction named rdtsc, used to determine both start and end time of the event handler routine so as to take an individual sample of the CPU-time demand by the event of that type. For each event type we have therefore a tuple <type,expected_granularity>, where the expected granularity is computed as the exponential moving average (EMA) over the collected samples (with the parameter α, the weight of old samples, set to 0.2). The usage of EMA allows us to capture scenarios where the activities at the level of the simulation model implementation are non-stationary, meaning that the granularity of the events of a given type can change along the simulation run. Also, the above tuple can be associated with each individual logical process so as to estimate the expected granularity of events of a same type occurring at different simulation objects. This helps in scenarios where the simulation model is not symmetric, in which the execution of events of the same type can lead to observe different values of the CPU-time demanded when these events are processed for different simulation objects. This is why we included the event_type field within the control table data structure. In fact, when some thread determines that an event e being processed by another thread on behalf of a logical process O is no longer consistent, we can exploit the above statistical information to retrieve the expected CPU-demand $CPU(e)$. The latter value is then compared with a threshold value TR_{cpu} as follows

$$CPU(e) \leq TR_{cpu}$$

in which case we can skip sending the IPI towards the target CPU-core as the residual time that would be saved for such a fine-grain event will probably not pay off the cost that would be paid to send it. Clearly, this approach requires the determination of a meaningful value for TR_{cpu}, which in our experiments has been set to 10 times the delay for delivering the IPI to the destination CPU-core after having observed the outcomes of some runs of different applications. We want to point out that the IPI delivery delay (including the cost for calling the send_IPI system-call) is of the order of 1 or 2 microseconds for common chipset (*e.g.*, Intel Xeon and AMD Opteron). Hence, opting for sending the IPI if the expected granularity of event to be preempted is at least of 10 microseconds seems reasonable to possibly save a non-minimal amount of CPU-time that would be wasted otherwise.

On the other hand, while the solution above allows avoiding the send of the IPI when we are confident that no significant revenue will come out, we cannot be sure that for events with CPU-demand larger than TR_{cpu} we can actually take advantage by sending the IPI. In fact, the event to be hit might have already been executed almost completely. Hence, the attempt to squash it via the IPI would mostly result in overhead. To cope with this problem, we exploit the start_time field within the control table, which allows us, in conjunction with the expected event granularity, to compute also the expected residual CPU-time for completing the involved event.

Specifically, the thread that detects the inconsistency of an event e can read the TSC register in order to assess what portion of e has been already processed and to determine the expected residual time $RES(e)$. If the latter value is larger than a second threshold TR_{res}, thus satisfying the inequality below

$$RES(e) \geq TR_{res}$$

then the IPI is sent towards the target CPU-core. As for the threshold TR_{res}, we also suggest the following setup

$$TR_{res} = \beta \times TR_{cpu} \text{ with } \beta \geq 1$$

which constraints the residual time of the event to be hit via the IPI to be at least equal to the minimum granularity of the events that we consider eligible for being interrupted. In our experiments we have set the parameter β to 1 in such a way to let the threshold TR_{ref} to coincide with the value of TR_{cpu} which, as we already mentioned, has been set to ten times the IPI latency observed on the hardware platforms used for the experimental evaluation.

By the way, in order to evaluate the effectiveness and the cost associated with our proposal, we have performed an extensive experimental evaluation that includes both synthetic and real-world simulation models. Also, we tested our preemptive PDES architecture on two hardware platforms, whose details are given in Table 5.1.

Table 5.1. Hardware evaluation platforms.

Platform	AMD	INTEL
Processors	4 × AMD Opteron Processor 6128	2 × Intel Xeon E5-2650v4
Cores (Logical)	32	24 (48)
NUMA Nodes	8	2
RAM	64GB	128GB
Operating System	Debian 5.4.19	Ubuntu 19.04
Linux Kernel	5.4.0	5.0.0
IPI Latency	1 μs	1 μs

As for the benchmark applications, we performed the experiments by relying on two different simulation models: PHOLD [25] and Personal Communication System (PCS). The first one has been exploited to estimate potential overheads of our approach under different configurations in terms of event granularity. Conversely, PCS allowed us to provide an example of the benefits given by reducing the amount of doomed to be rolled back work in a real-world simulation model. In any case, both models have been configured with a number of simulation objects (#SO) equal to 1, 2 and 3 times the number of CPU-cores. The reduced ratio between the number of simulation objects and that of the CPU-cores, as well as the absence of lookahead, make the execution characterized by a high degree of simulation-object execution parallelism, which is considered a challenging scenario for speculative simulation.

As for the PHOLD application, it is a synthetic benchmark for which the execution of each event leads to updating the state of the target logical processes, which keeps track of statistics related to simulation advancement such as the number of processed events and average simulation time advancement observed by simulation objects. It also leads to executing a classical CPU busy-loop for the emulation of a given event granularity. Here, we can distinguish two types of events, that is, regular events, whose events generates new events of any type, and diffusion events, which do not generate new events when being processed. The number of diffusion events generated by the regular ones (denoted as fan-out) is set to 1 in our evaluation. This event pattern leads to scenarios where the average number of events in the event pool is stable, but there are punctual fluctuations. Also, the timestamp increments are drawn from an exponential distribution with a mean value that we have set to one simulation-time unit. Finally, the busy-loop proper of PHOLD event processing has been configured to generate different event granularities for different tests, namely 5, 15, 45, 135 and 405 microseconds on both the hardware platforms, in order to emulate low to high granularity events proper of the large variety of discrete event models. Anyhow, given the very low incidence of rollback we observed with PHOLD under these settings—the percentage of straggler events is below 1% in all configurations—we used this benchmark mainly to estimate potential overheads of our ULMT-based version of the USE runtime compared to the original one.

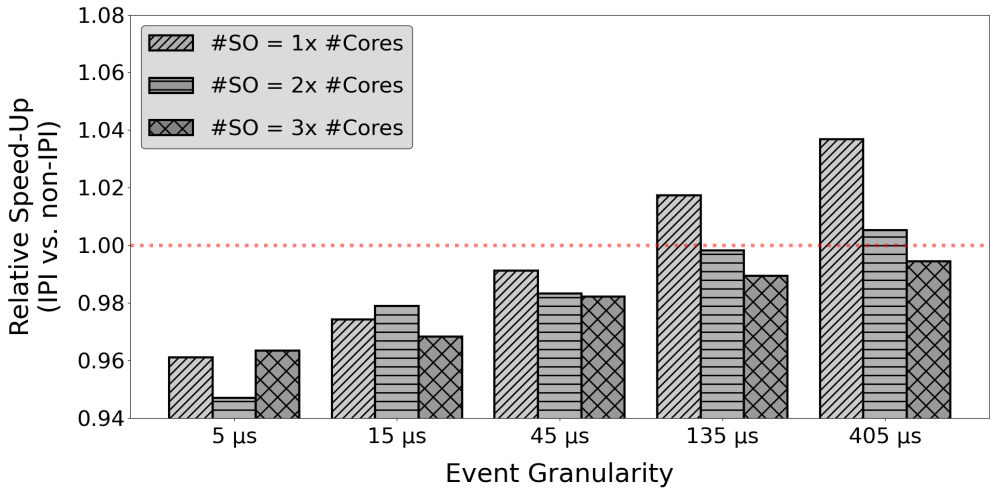

Figure 5.7. Speed-ups obtained with PHOLD on the AMD platform.

The results of this benchmark are presented in Figures 5.7 and 5.8 respectively for the AMD and INTEL platforms, which report the speed-up/slow-down introduced by our solution computed as the ratio between the average throughput values obtained with different SO counts. As for the AMD platform, the plot shows that our support introduces a sensible overhead, about 5%, for very fine-grain events, namely those which take 5 microseconds in the CPU to complete. Such overhead is even smaller, up to 2%, when running on a different and more recent architecture, as for the case of the INTEL platform. Moreover, it must be noted that the runtime decision support, configured to avoid the sending of IPIs towards events with granularity less than 10 microseconds, never enables threads to send IPIs when the average

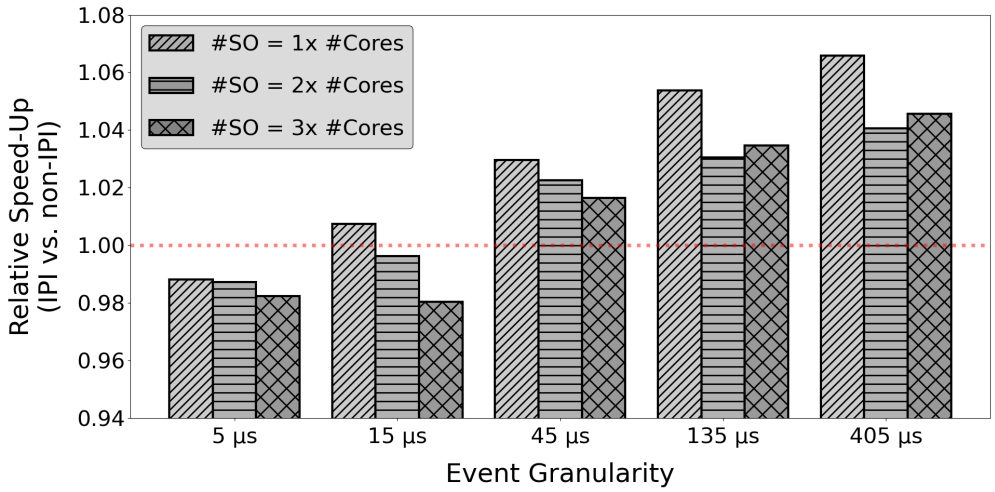

Figure 5.8. Speed-ups obtained with PHOLD on the INTEL platform.

event granularity is around 5 microseconds. Thus, what we are looking at with this particular configuration is pure overhead without any performance reward, since all the effort spent for publishing the control table in a fresh incarnation of its content upon beginning each event execution, as well as querying the control table of a given simulation object when a new event has been scheduled for it, is not repaid in any manner. In any case, this results shows that our approach is non-intrusive at all, while being capable of providing speed-up values up to 4% and 6% respectively for the AMD and INTEL platforms in very adverse scenarios, namely when the number of rollbacks observed even with larger-grain events is still a very small value.

As for the PCS application, it is a benchmark that models a mobile network adhering to GSM technology, where each simulation object models in turn the evolution of an individual hexagonal cell. Each cell can handle a number of N channels, which are modelled via power regulation and interface/fading phenomena, according to the results in [44]. The record associated with channels are then dynamically allocated and released at the beginning and end of calls respectively. Upon call setup, power regulation is performed, which involves scanning the aforementioned list of records to compute the minimum transmission power allowing the current call setup to achieve the threshold-level signal-to-interference (SIR) ratio. Each record is then released when the corresponding call ends or is handed off towards an adjacent cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a model defining meteorological conditions and their variations. For the purpose of our evaluation, the set of parameters that we varied to produce different configurations with which to conduct different experiments comprises:

- τ_A, which is the inter-arrival time of subsequent calls to any target cell;
- τ_D, which expresses the expected call duration;
- τ_H, which expresses the residual residence time of a mobile device into the current cell.

In their turn, these parameters affect the channel utilization factor, expressed as $\rho = \tau_D/(\tau_A \cdot N)$, where the value ρ impacts the granularity of the events, since the more the busy channels the more power-management records are allocated and consequently scanned and updated while processing events. At the same time, higher values of the channel utilization factor lead to higher memory requirements for representing the state of individual simulation objects. Also, CPU and memory demands are bounded depending on the total number N of per-cell managed channels. In fact, when a call-setup operation is requested due to either a call arrival or a hand off arrival, if all the channels are already busy, then the call is dropped, mimicking the real-world scenario where the communication is interrupted whenever the base station has no available resources to support it.

For this model, we have studied a configuration resembling high mobility of the devices involved in communication activities, like during morning hours around a commercial or business area, with many people moving towards their office or work place. Hence, we set the parameters to provide a non-minimal likelihood that an outgoing call is handed off between cells. In particular, τ_D and τ_H have been set to 300 and 120 seconds respectively, while N and τ_A have been set to values that allowed to evaluate scenarios where the channel utilization factor is equal to 0.3, 0.6 and 0.9 of the overall capacity. The latter settings therefore lead to simulate PCS with events having a granularity that varies from 87 to 195 microseconds and from 45 to 100 microseconds in the AMD and INTEL platforms respectively, depending on the channel utilization factor.

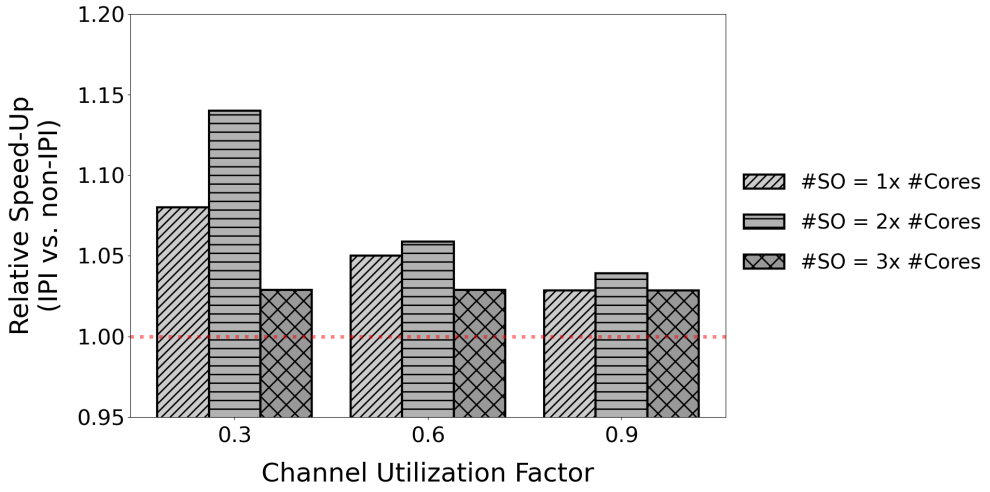

Figure 5.9. Speed-ups obtained with PCS on the AMD platform.

Figures 5.9 and 5.10 show the results achieved while running on top of the two hardware platforms. Coherently with what we observed with the PHOLD model, the benefits provided by the IPI-based mechanism, to accomplish the interrupt-driven preemption of logical processes when they are detect to perform no longer consistent event executions, are more evident when running with lower counts of simulation objects because of the higher degree of actual simulation-object execution parallelism which lead to more pronounced speculation. For instance, the maximum performance is in most of the cases achieved when running with a number of simulation objects

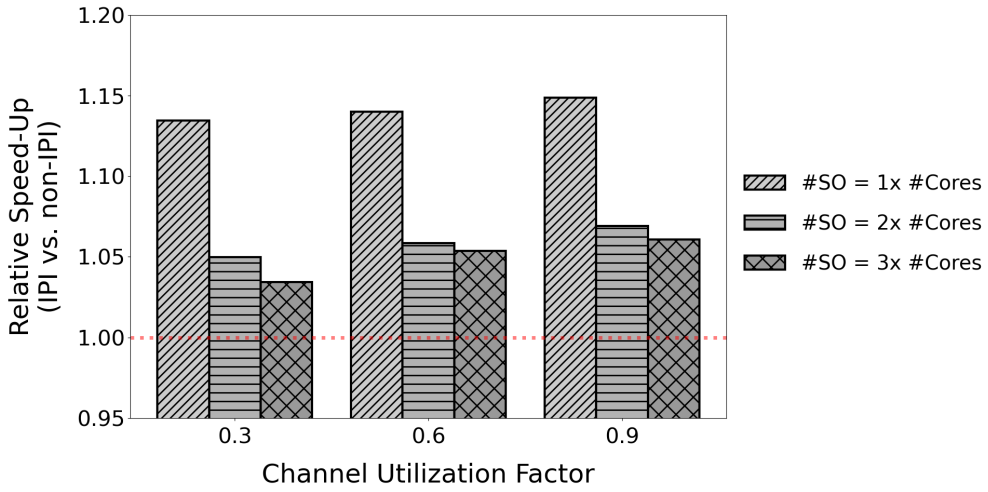

Figure 5.10. Speed-ups obtained with PCS on the INTEL platform.

tied to the number of CPU-cores, giving at least 5% and 13% speed-up in the AMD and INTEL platforms respectively, and providing a maximum speed-up of 15% when running on top of the INTEL platform.

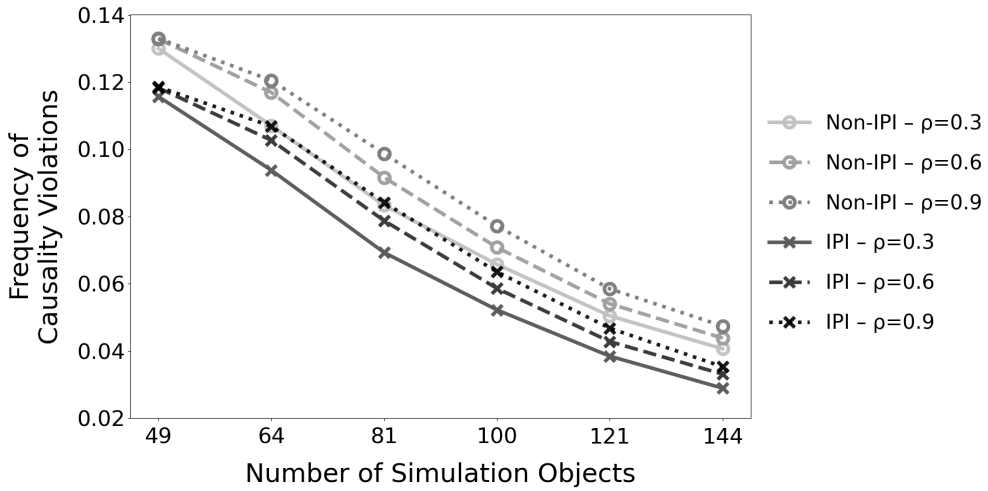

Figure 5.11. Frequency of causality violations observed with PCS on the INTEL platform.

Even though the event granularity is enough large compared to the IPI latency, the benefit introduced by our solution is strongly related with the probability of rollbacks. As for this aspect, we report in Figure 5.11 data related to the frequency of causality errors for the executions carried out on the INTEL platform. The values reported in this graphic were computed as the ratio between the number of rolled back events and the whole number of events processed. By the curves we can see that the frequency of causality violations decreases from about 12/13% to 3/5% when reducing the actual level of execution parallelism, namely when increasing the number of simulation objects, regardless of the channel utilization factor. Hence, even runs

with larger-grain events are characterized by a low rollback rate when the number of simulation objects increases, as a consequence of a decreased possibility of speculating in the simulated future which derives from the fact that a PDS system following the share-everything paradigm, such as USE, is designed precisely to prevent this kind of behaviour, by always selecting the event with the lowest timestamp among all the ones that have already been scheduled for all those simulation objects that have not yet been tied to any thread. Additionally, we can see a slightly reduced frequency of causality errors when running with our ULMT-based preemptive PDES architecture, indicating that the interrupt-driven approach accomplished by sending IPIs for early interrupting the processing phase of events that are no longer causally consistent can also prevent the spreading of inconsistent computation. In fact, early interrupting the no longer causally consistent execution of a given event also avoids injecting in the system additional inconsistent events which would otherwise have been generated by its processing.

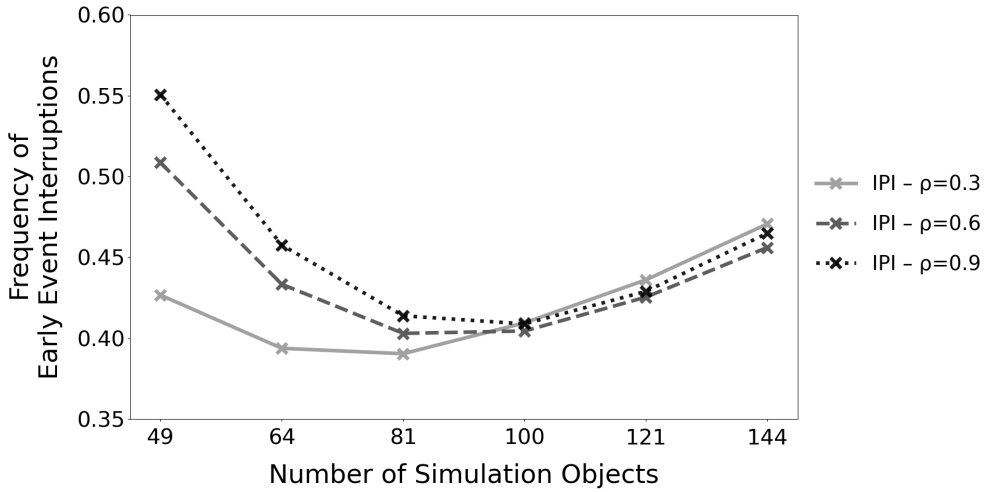

Figure 5.12. Frequency of early event interruptions observed with PCS on the INTEL platform.

Still for runs on the INTEL platform, we show in Figure 5.12 the frequency of early interruptions of causally inconsistent events achieved by sending IPIs, which has been computed as the ratio between the number of early rolled back events and the whole number of rollbacks. By the results, we can see how this frequency ranges from about 40% to a maximum of 55%, depending on the different application settings. This confirms the fact that a good percentage of the causally inconsistent event executions can be actually early rolled back by preventing from a waste of resources and CPU-time. Also, we can notice higher frequency values when running with a number of simulation objects that is three times the number of CPU-cores with respect to when employing its double, which does not clearly means a higher absolute value of the number of early rolled back events as we know there are less chances to experience causality violations, rather it is indicative of the fact that the whole number of rolled back events is a curve that decreases much faster than that of the number of early rollbacks when the amount of simulation objects used increases. The latter phenomenon is explained by the fact that from a reduced actual simulation-

object execution parallelism, hence a reduced incidence of rollbacks, also derives a reduced number of events that need to be silently processed—they are all those events that still lie along a causally consistent branch of the simulation but whose effects have been lost upon restoring an older state of the involved simulation object—after having restored a previously taken snapshot of the simulation-object state. All these silent executions only provide for a reduced set of operations with respect to when the involved events are normally processed. In fact, they do not provide for re-scheduling events that have already been scheduled in past because, as hinted, previous executions of these events are still causally consistent. Nevertheless, such chains of silent executions can anyway be subject to invalidation due to the dispatch of new events by concurrent logical processes. Given that only few operations are actually performed during the silent execution of events, the latter have a granularity that is several times smaller than during normal execution—we also collect statistics about silent executions of events to estimate their granularity—thus meaning that if they are detected to be no longer causally consistent, then the runtime decision support always opts for non-sending the IPI. This condition therefore leads to the scenario where the number of rolled back events during silent executions only falls within the counts of the whole number of rollbacks, but never that of the early ones. Anyhow, the latter has not to be meant as a shortcoming of the runtime decision support, in that it just do what it has been devised to do for having a revenue that pays off the introduced overhead. Rather, it provides a different interpretation of data discussed so far, which underlies the goodness of our solution based on the mechanism for sending IPIs, together with the runtime decision support aimed at reducing the overhead, when the number of early rollbacks is compared with the whole number of rollbacks that occurred only when events were not silently processed in a scenario with an already low incidence of rollbacks.

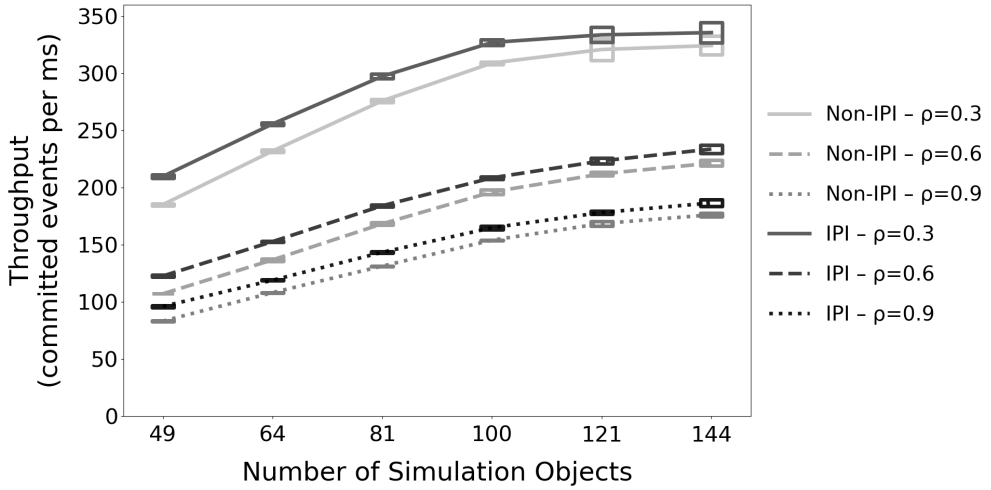

Figure 5.13. Throughput values achieved with PCS on the INTEL platform.

Finally, in Figure 5.13 we offer a different view of the final performance data for the execution of PCS model still on top of the INTEL platform. In particular, we report the throughput curves obtained by the experiments performed with both the

original version of the USE runtime and that of the ULMT-based one that relies on the mechanism for sending IPIs in order to early rollback causally inconsistent event executions, in which are also included the confidence intervals calculated on the basis of the different outcomes achieved over the 10 runs per data-point we carried out. The data show that the advantages by the IPI-based approach are statically consistent across all the simulation object counts.

Just to conclude, inter-processor-interrupts are a fundamental technology in modern software/hardware systems. They allow one CPU-core to notify the others about important tasks to be promptly executed. As hinted, they are already exploited in OS technology to correctly drive the execution of multi-threaded applications, where something occurring on a given CPU-core needs to be reflected on the state of the hardware or shared data structures seen by other threads running on different CPU-cores. However, to the best of our knowledge, IPIs have not yet been exposed for optimizing specific application scenarios, where sudden program state changes require immediate reaction by one or more threads participating to the program execution in order to prevent the application as a whole from performing sub-optimally, thus improving the overall performance. This is the case where such changes are due to operations performed by a thread running on a given CPU-core but whose effects affect the execution state of other threads running on different CPU-cores. Just for this reason the IPI-based approach results very useful as it applies very well for accomplishing the interrupt-driven paradigm we discussed extensively at the beginning of this chapter. Clearly, everything must be assisted by an adequate software technology, such as the ULMT one, which allows to accomplish the commissioning of an execution model for application's tasks that provides for the preemptibility of the tasks themselves, such as the micro-threading model, when certain interruptions occur along the execution path of threads in charge of carrying out them. The work presented in this subsection is precisely an application of this philosophy to the PDES scenario, in which we have proposed a solution where IPIs are used to drive the evolution, in virtual time, of simulation objects running within a speculative PDES environment hosted by a multi-core shared-memory machine.

Chapter 6

Conclusions

We have presented in this thesis a new execution model for tasks that extends the interactions of application-specific activities with the runtime layer hosting the application at arbitrary points of their execution. More in detail, it allows to application's tasks, which were not expected to interact for a long time with software devoted to re-evaluating their execution state, to reassess the execution trajectory of threads in charge of carrying out such tasks so as to eventually renew the work assignment, as well as the execution validity of each single tasks currently being processed. As discussed, the latter would have not been possible unless specifically provided in the control-flow-graph (CFG) of the application, hence included into the code by the programmer in a non-transparent manner, nor there are time guarantees on how long it will take before such spontaneous interactions are likely to occur as the execution flow can always (un)conditionally branches out along several paths due to, *e.g.*, unknown number of iterations in a given loop, invocation of library functions and interaction with the Operating System through system-calls and interrupts. To cope with such problem, we have proposed two different approaches in order to promptly react to sudden program state changes with the aim of preventing the application from performing sub-optimally, thus providing better runtime dynamics with respect to those experienced when no timely intervention is ever actuated. Both approaches are based on asynchronous (and synchronous deferred) interruptions of task executions carried on by threads. A first one is based on a polling scheme actuated via the exploitation of specific performance counter registers proper of the x86 architectures, which aims to make threads able to check noteworthy changes in the program state autonomously at predefined (possibly adaptively varied) time intervals, thus not requiring the cooperation of other threads participating to the overall execution of the application. A second one is instead cooperative, in the sense that the only program state changes of interest for the application's tasks are due to operations performed by other tasks currently being processed along the execution path of threads, which are aware of their occurrence and can then attempt to notify them to those concerned. Both approaches have been applied to improve reactivity of threads in several application contexts such as Transactional Memory, OpenMP and Parallel Discrete Event Simulations in order to promptly respond to the occurrence of priority inversions, and consistency and causality violations. The results obtained confirm the capability of our proposal to provide better run-time

dynamics than when relying on classical execution models that do not provide preemptibility of application's tasks, thus leading to experience better performance results. By the way, the benefits provided by the second approach are multiple and include a reduced overhead due to a small number of control-flow-variations (CFV) occurring along the execution path of the involved threads, only when they are actually needed. Also, thanks to the inter-processor interrupt (IPI) hardware facility exploited to pursue this kind of approach on x86 architecture machines, their occurrence is characterized by very low latencies with respect to when program state changes of interest materialize, thus improving the reactivity of threads in realigning the application's behaviour to better execution dynamics and performance, as we have shown for the case of speculative processing-based applications.

As a final note, just thanks to the very low overhead introduced and the low latency experienced when relying on our mechanisms, we believe that the solutions proposed in this thesis also represent a valid architectural evolution to be considered for future designs of the software environment hosting parallel applications in order to support preemptive execution of application's tasks according to the Micro-Threading model. This mostly regards services offered by the Operating System like system-calls, special devices and interrupt handlers appositely implemented at this lower level to correctly drive the involved hardware and with the aim of providing the applications with the capability to perform task-to-CPU assignment in a manner that is alternative to the thread-to-CPU assignment currently established by the Operating System. On the other hand, user-space system libraries are required to provide software facilities in order to allow the applications to access such services, even better if provided within specific runtime libraries implementing parallel programming models (or environments) appositely extended to accomplish the commissioning of the Micro-Threading model in a completely transparent manner, thus relieving the programmers from the need of worrying about lower level threading concepts and to only focus on the specification of the programming model employed.

Bibliography

[1] AMD64 Technology. *AMD64 Architecture Programmer's Manual Volume 2: System Programming, Publication No. 24593, Revision 3.35* (2020). Available from: https://www.amd.com/system/files/TechDocs/24593.pdf.

[2] Andersson, B., Abdelzaher, T. F., and Jonsson, J. Global priority-driven aperiodic scheduling on multiprocessors. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, p. 8. IEEE Computer Society (2003). Available from: https://doi.org/10.1109/IPDPS.2003.1213082, doi:10.1109/IPDPS.2003.1213082.

[3] Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Su, E., Unnikrishnan, P., and Zhang, G. A proposal for task parallelism in openmp. In *A Practical Programming Model for the Multi-Core Era, 3rd International Workshop on OpenMP, IWOMP 2007, Beijing, China, June 3-7, 2007, Proceedings* (edited by B. M. Chapman, W. Zheng, G. R. Gao, M. Sato, E. Ayguadé, and D. Wang), vol. 4935 of *Lecture Notes in Computer Science*, pp. 1–12. Springer (2007). Available from: https://doi.org/10.1007/978-3-540-69303-1_1, doi:10.1007/978-3-540-69303-1_1.

[4] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000* (edited by L. Rudolph and A. Gupta), pp. 117–128. ACM Press (2000). Available from: https://doi.org/10.1145/356989.357000, doi:10.1145/356989.357000.

[5] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Santa Barbara, California, USA, July 19-21, 1995* (edited by J. Ferrante, D. A. Padua, and R. L. Wexelblat), pp. 207–216. ACM (1995). Available from: https://doi.org/10.1145/209936.209958, doi:10.1145/209936.209958.

[6] Bovet, D. P. and Cesati, M. *Understanding the Linux Kernel - from I/O ports to process management: covers version 2.6 (3. ed.)*. O'Reilly (2005).

ISBN 978-0-596-00565-8. Available from: http://www.oreilly.de/catalog/understandlk/index.html.

[7] Carothers, C. D. and Fujimoto, R. Efficient execution of time warp programs on heterogeneous, NOW platforms. *IEEE Trans. Parallel Distributed Syst.*, **11** (2000), 299. Available from: https://doi.org/10.1109/71.841745, doi:10.1109/71.841745.

[8] Choe, M. and Tropper, C. On learning algorithms and balancing loads in time warp. In *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation, PADS '99, Atlanta, GA, USA, May 1-4, 1999* (edited by R. M. Fujimoto and S. J. Turner), pp. 101–108. IEEE Computer Society (1999). Available from: https://doi.org/10.1109/PADS.1999.766166, doi:10.1109/PADS.1999.766166.

[9] Cingolani, D., Pellegrini, A., and Quaglia, F. Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. *ACM Trans. Model. Comput. Simul.*, **27** (2017), 11:1. Available from: https://doi.org/10.1145/3077583, doi:10.1145/3077583.

[10] 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, Chicago, IL, USA, May 23-27, 2016*, pp. 373–382. IEEE Computer Society (2016). Available from: https://doi.org/10.1109/IPDPS.2016.104, doi:10.1109/IPDPS.2016.104.

[11] Dice, D., Shalev, O., and Shavit, N. Transactional locking II. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings* (edited by S. Dolev), vol. 4167 of *Lecture Notes in Computer Science*, pp. 194–208. Springer (2006). Available from: https://doi.org/10.1007/11864219_14, doi:10.1007/11864219_14.

[12] Dickens, P. M., Nicol, D. M., Jr., P. F. R., and Duva, J. M. Analysis of bounded time warp and comparison with YAWNS. *ACM Trans. Model. Comput. Simul.*, **6** (1996), 297. Available from: https://doi.org/10.1145/240896.240913, doi:10.1145/240896.240913.

[13] 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. Available from: https://doi.org/10.1007/s00607-013-0376-3, doi:10.1007/s00607-013-0376-3.

[14] Dolev, S., Hendler, D., and Suissa, A. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008* (edited by R. A. Bazzi and B. Patt-Shamir), pp. 125–134. ACM (2008). Available from: https://doi.org/10.1145/1400751.1400769, doi:10.1145/1400751.1400769.

[15] Dragojevic, A. and Guerraoui, R. Predicting the scalability of an stm: A pragmatic approach. In *5th ACM SIGPLAN Workshop on Transactional Computing*, POST_TALK (2010).

[16] Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguadé, E. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009*, pp. 124–131. IEEE Computer Society (2009). Available from: `https://doi.org/10.1109/ICPP.2009.64`, `doi:10.1109/ICPP.2009.64`.

[17] Engelschall, R. S. Portable multithreading-the signal stack trick for user-space thread creation. In *Proceedings of the General Track: 2000 USENIX Annual Technical Conference, June 18-23, 2000, San Diego, CA, USA*, pp. 239–250. USENIX (2000). Available from: `http://www.usenix.org/publications/library/proceedings/usenix2000/general/engelschall.html`.

[18] Ennals, R. Software transactional memory should not be obstruction free. In *In Intel Research Cambridge Tech Report* (2006).

[19] Faltelli, M., Belocchi, G., Quaglia, F., Pontarelli, S., and Bianchi, G. Metronome: adaptive and precise intermittent packet retrieval in DPDK. In *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020* (edited by D. Han and A. Feldmann), pp. 406–420. ACM (2020). Available from: `https://doi.org/10.1145/3386367.3432730`, `doi:10.1145/3386367.3432730`.

[20] Felber, P., Fetzer, C., Marlier, P., and Riegel, T. Time-based software transactional memory. *IEEE Trans. Parallel Distributed Syst.*, **21** (2010), 1793. Available from: `https://doi.org/10.1109/TPDS.2010.49`, `doi:10.1109/TPDS.2010.49`.

[21] 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 2008, Salt Lake City, UT, USA, February 20-23, 2008* (edited by S. Chatterjee and M. L. Scott), pp. 237–246. ACM (2008). Available from: `https://doi.org/10.1145/1345206.1345241`, `doi:10.1145/1345206.1345241`.

[22] Fischer, P. C. and Probert, R. L. Efficient procedures for using matrix algorithms. In *Automata, Languages and Programming, 2nd Colloquium, University of Saarbrücken, Germany, July 29 - August 2, 1974, Proceedings* (edited by J. Loeckx), vol. 14 of *Lecture Notes in Computer Science*, pp. 413–427. Springer (1974). Available from: `https://doi.org/10.1007/3-540-06841-4_78`, `doi:10.1007/3-540-06841-4\_78`.

[23] Fraser, K. *Practical lock-freedom*. Ph.D. thesis, University of Cambridge, UK (2004). Available from: `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193`.

[24] Fujimoto, R. Parallel discrete event simulation. *Commun. ACM*, **33** (1990), 30. Available from: https://doi.org/10.1145/84537.84545, doi:10.1145/84537.84545.

[25] Fujimoto, R. M. Performance of time warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulations, 1990*, vol. 22, pp. 23–28 (1990).

[26] Gautier, T., Pérez, C., and Richard, J. On the impact of openmp task granularity. In *Evolving OpenMP for Evolving Architectures - 14th International Workshop on OpenMP, IWOMP 2018, Barcelona, Spain, September 26-28, 2018, Proceedings* (edited by B. R. de Supinski, P. Valero-Lara, X. Martorell, S. M. Bellido, and J. Labarta), vol. 11128 of *Lecture Notes in Computer Science*, pp. 205–221. Springer (2018). Available from: https://doi.org/10.1007/978-3-319-98521-3_14, doi:10.1007/978-3-319-98521-3_14.

[27] GCC Team. An OpenMP implementation for GCC - GNU project - free software foundation (FSF) (2020). Available from: https://gcc.gnu.org/projects/gomp/.

[28] Gennaro, I. D., Pellegrini, A., and Quaglia, F. Os-based NUMA optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses. In *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*, pp. 291–300. IEEE Computer Society (2016). Available from: https://doi.org/10.1109/CCGrid.2016.91, doi:10.1109/CCGrid.2016.91.

[29] Gleixner, T. and Niehaus, D. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the Linux symposium*, vol. 1, pp. 333–346. Citeseer (2006).

[30] Goossens, J., Funk, S., and Baruah, S. K. Priority-driven scheduling of periodic task systems on multiprocessors. *Real Time Syst.*, **25** (2003), 187. Available from: https://doi.org/10.1023/A:1025120124771, doi:10.1023/A:1025120124771.

[31] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is dead: Long live KASLR. In *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings* (edited by E. Bodden, M. Payer, and E. Athanasopoulos), vol. 10379 of *Lecture Notes in Computer Science*, pp. 161–176. Springer (2017). Available from: https://doi.org/10.1007/978-3-319-62105-0_11, doi:10.1007/978-3-319-62105-0_11.

[32] Guerraoui, R., Herlihy, M., and Pochon, B. Polymorphic contention management. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings* (edited by P. Fraigniaud), vol. 3724 of *Lecture Notes in Computer Science*, pp. 303–323.

Springer (2005). Available from: https://doi.org/10.1007/11561927_23, doi:10.1007/11561927_23.

[33] Guerraoui, R. and Kapalka, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008* (edited by S. Chatterjee and M. L. Scott), pp. 175–184. ACM (2008). Available from: https://doi.org/10.1145/1345206.1345233, doi:10.1145/1345206.1345233.

[34] Herlihy, M. and Moss, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993* (edited by A. J. Smith), pp. 289–300. ACM (1993). Available from: https://doi.org/10.1145/165123.165164, doi:10.1145/165123.165164.

[35] Herlihy, M. and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, **12** (1990), 463. Available from: https://doi.org/10.1145/78969.78972, doi:10.1145/78969.78972.

[36] Ianni, M., Marotta, R., Cingolani, D., Pellegrini, A., and Quaglia, F. The ultimate share-everything PDES system. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, Rome, Italy, May 23-25, 2018* (edited by F. Quaglia, A. Pellegrini, and G. K. Theodoropoulos), pp. 73–84. ACM (2018). Available from: https://doi.org/10.1145/3200921.3200931, doi:10.1145/3200921.3200931.

[37] Ianni, M., Marotta, R., Pellegrini, A., and Quaglia, F. Towards a fully non-blocking share-everything PDES platform. In *21st IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2017, Rome, Italy, October 18-20, 2017* (edited by A. D'Ambrogio, R. E. D. Grande, A. Garro, and A. Tundis), pp. 25–32. IEEE Computer Society (2017). Available from: https://doi.org/10.1109/DISTRA.2017.8167663, doi:10.1109/DISTRA.2017.8167663.

[38] Intel Corporation. CilkPlus (2009). Available from: https://www.cilkplus.org/.

[39] Intel®. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B, Part 2, Order No. 253669-072US* (2020). Available from: https://software.intel.com/content/dam/develop/public/us/en/documents/253669-sdm-vol-3b.pdf.

[40] Jefferson, D. R. Virtual time. *ACM Trans. Program. Lang. Syst.*, **7** (1985), 404. Available from: https://doi.org/10.1145/3916.3988, doi:10.1145/3916.3988.

[41] Jonathan Corbet. Deadline scheduling for linux (2009). Available from: https://lwn.net/Articles/356576/.

[42] Jr., P. D. B., Carothers, C. D., Jefferson, D. R., and LaPre, J. M. Warp speed: executing time warp on 1, 966, 080 cores. In *SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS '13, Montreal, QC, Canada, May 19-22, 2013* (edited by M. L. Loper and G. A. Wainer), pp. 327–336. ACM (2013). Available from: https://doi.org/10.1145/2486092.2486134, doi:10.1145/2486092.2486134.

[43] Kalé, L. V., Bhandarkar, M. A., Jagathesan, N., Krishnan, S., and Yelon, J. Converse: An interoperable framework for parallel programming. In *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium, April 15-19, 1996, Honolulu, Hawaii, USA*, pp. 212–217. IEEE Computer Society (1996). Available from: https://doi.org/10.1109/IPPS.1996.508060, doi:10.1109/IPPS.1996.508060.

[44] Kandukuri, S. and Boyd, S. P. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Trans. Wirel. Commun.*, **1** (2002), 46. Available from: https://doi.org/10.1109/7693.975444, doi:10.1109/7693.975444.

[45] Kocher, P., et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pp. 1–19. IEEE (2019). Available from: https://doi.org/10.1109/SP.2019.00002, doi:10.1109/SP.2019.00002.

[46] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, **21** (1978), 558. Available from: https://doi.org/10.1145/359545.359563, doi:10.1145/359545.359563.

[47] LaPre, J. M., Gonsiorowski, E., and Carothers, C. D. LORAIN: a step closer to the PDES 'holy grail'. In *SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS '14, Denver, CO, USA, May 18-21, 2014* (edited by J. A. H. Jr., G. F. Riley, and R. M. Fujimoto), pp. 3–14. ACM (2014). Available from: https://doi.org/10.1145/2601381.2601397, doi:10.1145/2601381.2601397.

[48] Leiserson, C. E. The cilk++ concurrency platform. In *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*, pp. 522–527. ACM (2009). Available from: https://doi.org/10.1145/1629911.1630048, doi:10.1145/1629911.1630048.

[49] Lin, Y.-B. and Lazowska, E. D. Processor Scheduling for Time Warp Parallel Simulation. In *Advances in Parallel and Distributed Simulation*, pp. 11–14. IEEE Computer Society (1991).

[50] Lipp, M., et al. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018* (edited by W. Enck and A. P. Felt), pp. 973–990. USENIX Association (2018). Available from: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[51] Liu, C. L. and Layland, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, **20** (1973), 46. Available from: http://doi.acm.org/10.1145/321738.321743, doi:10.1145/321738.321743.

[52] Love, R., Are, S. H. W., Linus, A. C., and Begin, B. W. *Linux kernel development second edition.* Novell Press (2005).

[53] Madisetti, V. K., Walrand, J. C., and Messerschmitt, D. G. Wolf: a rollback algorithm for optimistic distributed simulation systems. In *Proceedings of the 20th conference on Winter simulation, WSC 1988, San Diego, California, USA, December 12-14, 1988* (edited by M. A. Abrams, P. L. Haigh, and J. C. Comfort), pp. 296–305. ACM (1988). Available from: https://doi.org/10.1145/318123.318205, doi:10.1145/318123.318205.

[54] Maldonado, W., Marlier, P., Felber, P., Lawall, J., Muller, G., and Rivière, E. Supporting time-based qos requirements in software transactional memory. *ACM Trans. Parallel Comput.*, **2** (2015), 10:1. Available from: https://doi.org/10.1145/2779621, doi:10.1145/2779621.

[55] Marathe, V. J., III, W. N. S., and Scott, M. L. Adaptive software transactional memory. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings* (edited by P. Fraigniaud), vol. 3724 of *Lecture Notes in Computer Science*, pp. 354–368. Springer (2005). Available from: https://doi.org/10.1007/11561927_26, doi:10.1007/11561927_26.

[56] Marotta, R., Ianni, M., Pellegrini, A., and Quaglia, F. A lock-free O(1) event pool and its application to share-everything PDES platforms. In *20th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2016, London, United Kingdom, September 21-23, 2016*, pp. 53–60. IEEE Computer Society (2016). Available from: https://doi.org/10.1109/DS-RT.2016.33, doi:10.1109/DS-RT.2016.33.

[57] Michael, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distributed Syst.*, **15** (2004), 491. Available from: https://doi.org/10.1109/TPDS.2004.8, doi:10.1109/TPDS.2004.8.

[58] Microsoft. SetPriorityClass function. Available from: https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setpriorityclass.

[59] Myers, G., Selznick, S., Zhang, Z., and Miller, W. Progressive multiple alignment with constraints. *J. Comput. Biol.*, **3** (1996), 563. Available from: https://doi.org/10.1089/cmb.1996.3.563, doi:10.1089/cmb.1996.3.563.

[60] Nakashima, J. and Taura, K. Massivethreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond - Papers dedicated to Akinori Yonezawa on the Occasion of His 65th Birthday* (edited by G. A. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and

K. Taura), vol. 8665 of *Lecture Notes in Computer Science*, pp. 222–238. Springer (2014). Available from: https://doi.org/10.1007/978-3-662-44471-9_10, doi:10.1007/978-3-662-44471-9_10.

[61] Nicol, D. M. and Liu, J. Composite synchronization in parallel discrete-event simulation. *IEEE Trans. Parallel Distributed Syst.*, **13** (2002), 433. Available from: https://doi.org/10.1109/TPDS.2002.1003854, doi:10.1109/TPDS.2002.1003854.

[62] OpenMP Architecture Review Board. OpenMP application program interface version 4.5 (2015). Available from: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[63] OpenMP Architecture Review Board, Tech. Rep. The OpenMP API specification for parallel programming (1997). Available from: https://www.openmp.org/.

[64] Pellegrini, A. and Quaglia, F. A fine-grain time-sharing time warp system. *ACM Trans. Model. Comput. Simul.*, **27** (2017), 10:1. Available from: https://doi.org/10.1145/3013528, doi:10.1145/3013528.

[65] Pellegrini, A., Vitali, R., and Quaglia, F. Autonomic state management for optimistic simulation platforms. *IEEE Trans. Parallel Distributed Syst.*, **26** (2015), 1560. Available from: https://doi.org/10.1109/TPDS.2014.2323967, doi:10.1109/TPDS.2014.2323967.

[66] Peluso, S., Didona, D., and Quaglia, F. Supports for transparent object-migration in PDES systems. *J. Simulation*, **6** (2012), 279. Available from: https://doi.org/10.1057/jos.2012.13, doi:10.1057/jos.2012.13.

[67] Preiss, B. R., Loucks, W. M., and MacIntyre, I. D. Effects of the checkpoint interval on time and space in time warp. *ACM Trans. Model. Comput. Simul.*, **4** (1994), 223. Available from: https://doi.org/10.1145/189443.189444, doi:10.1145/189443.189444.

[68] Quaglia, F. and Cortellessa, V. On the processor scheduling problem in time warp synchronization. *ACM Trans. Model. Comput. Simul.*, **12** (2002), 143. Available from: https://doi.org/10.1145/643114.643115, doi:10.1145/643114.643115.

[69] Quaglia, F., Cortellessa, V., and Ciciani, B. Trade-off between sequential and time warp-based parallel simulation. *IEEE Trans. Parallel Distributed Syst.*, **10** (1999), 781. Available from: https://doi.org/10.1109/71.790597, doi:10.1109/71.790597.

[70] Riegel, T., Felber, P., and Fetzer, C. A lazy snapshot algorithm with eager validation. In *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings* (edited by S. Dolev), vol. 4167 of *Lecture Notes in Computer Science*, pp. 284–298. Springer (2006). Available from: https://doi.org/10.1007/11864219_20, doi:10.1007/11864219_20.

[71] Riegel, T., Fetzer, C., and Felber, P. Snapshot isolation for software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, pp. 1–10. Association for Computing Machinery (ACM) (2006).

[72] Rivas, M. A. and Harbour, M. G. Evaluation of new POSIX real-time operating systems services for small embedded platforms. In *15th Euromicro Conference on Real-Time Systems (ECRTS 2003), 2-4 July 2003, Porto, Portugal, Proceedings*, pp. 161–168. IEEE Computer Society (2003). Available from: https://doi.org/10.1109/EMRTS.2003.1212739, doi:10.1109/EMRTS.2003.1212739.

[73] 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, Chicago, IL, USA, May 26-29, 2014*, pp. 81–91. IEEE Computer Society (2014). Available from: https://doi.org/10.1109/CCGrid.2014.118, doi:10.1109/CCGrid.2014.118.

[74] Rünger, G. and Rauber, T. *Parallel Programming - for Multicore and Cluster Systems; 2nd Edition*. Springer (2013). ISBN 978-3-642-37800-3. Available from: https://doi.org/10.1007/978-3-642-37801-0, doi:10.1007/978-3-642-37801-0.

[75] Seo, S., et al. Argobots: A lightweight low-level threading and tasking framework. *IEEE Trans. Parallel Distributed Syst.*, **29** (2018), 512. Available from: https://doi.org/10.1109/TPDS.2017.2766062, doi:10.1109/TPDS.2017.2766062.

[76] Serrano, M. A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., and Quiñones, E. Timing characterization of openmp4 tasking model. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2015, Amsterdam, The Netherlands, October 4-9, 2015* (edited by R. Iyer and S. Garg), pp. 157–166. IEEE (2015). Available from: https://doi.org/10.1109/CASES.2015.7324556, doi:10.1109/CASES.2015.7324556.

[77] Sha, L., Rajkumar, R., and Lehoczky, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, **39** (1990), 1175. Available from: https://doi.org/10.1109/12.57058, doi:10.1109/12.57058.

[78] Shanley, K. TPC releases new benchmark: TPC-C. *SIGMETRICS Perform. Evaluation Rev.*, **20** (1992), 8. Available from: https://doi.org/10.1145/141858.141861, doi:10.1145/141858.141861.

[79] Shavit, N. and Touitou, D. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995* (edited by J. H. Anderson), pp.

204–213. ACM (1995). Available from: https://doi.org/10.1145/224964.224987, doi:10.1145/224964.224987.

[80] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts, 10th Edition*. Wiley (2018). ISBN 978-1-118-06333-0. Available from: http://os-book.com/OS10/index.html.

[81] Srinivasan, A. and Baruah, S. K. Deadline-based scheduling of periodic task systems on multiprocessors. *Inf. Process. Lett.*, **84** (2002), 93. Available from: https://doi.org/10.1016/S0020-0190(02)00231-4, doi:10.1016/S0020-0190(02)00231-4.

[82] Srinivasan, S. and Jr., P. F. R. Elastic time. *ACM Trans. Model. Comput. Simul.*, **8** (1998), 103. Available from: https://doi.org/10.1145/280265.280267, doi:10.1145/280265.280267.

[83] Sun, J., Guan, N., Wang, Y., He, Q., and Yi, W. Real-time scheduling and analysis of openmp task systems with tied tasks. In *2017 IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pp. 92–103. IEEE Computer Society (2017). Available from: https://doi.org/10.1109/RTSS.2017.00016, doi:10.1109/RTSS.2017.00016.

[84] Vargas, R., Quiñones, E., and Marongiu, A. Openmp and timing predictability: a possible union? In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015* (edited by W. Nebel and D. Atienza), pp. 617–620. ACM (2015). Available from: http://dl.acm.org/citation.cfm?id=2755893.

[85] Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., and Gautier, T. Evaluation of openmp dependent tasks with the KASTORS benchmark suite. In *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings* (edited by L. DeRose, B. R. de Supinski, S. L. Olivier, B. M. Chapman, and M. S. Müller), vol. 8766 of *Lecture Notes in Computer Science*, pp. 16–29. Springer (2014). Available from: https://doi.org/10.1007/978-3-319-11454-5_2, doi:10.1007/978-3-319-11454-5_2.

[86] Vitali, R., Pellegrini, A., and Quaglia, F. Load sharing for optimistic parallel simulations on multi core machines. *SIGMETRICS Perform. Evaluation Rev.*, **40** (2012), 2. Available from: https://doi.org/10.1145/2425248.2425250, doi:10.1145/2425248.2425250.

[87] Yoo, R. M. and Lee, H. S. Adaptive transaction scheduling for transactional memory systems. In *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008* (edited by F. M. auf der Heide and N. Shavit), pp. 169–178. ACM (2008). Available from: https://doi.org/10.1145/1378533.1378564, doi:10.1145/1378533.1378564.