SAPIENZA
UNIVERSITÀ DI ROMA

Share-everything Parallel Discrete Event Simulation on Multi-core Machines

Sapienza University of Rome
Ph.D. program in Computer Engineering
XXXI Cycle

Mauro Ianni

Thesis Advisors

Prof. Francesco Quaglia
Dr. Alessandro Pellegrini

Co-Advisor

Dr. Luca Becchetti

Reviewers

Prof. Philip A. Wilsey
Prof. Georgios K. Theodoropoulos

A.Y. 2017/2018

Thesis not yet defended

Share-everything Parallel Discrete Event Simulation on Multi-core Machines
Ph.D. thesis. Sapienza – University of Rome
ISBN: 000000000-0

Website: http://www.diag.uniroma1.it/∼mianni/
Author's email: mianni@diag.uniroma1.it

We apologize
for the inconvenience

Abstract

Discrete Event Simulation is a powerful technique to mimic the evolution of real-world or envisioned phenomena. A traditional way to achieve high performance simulations is the employment of parallelization techniques, enabling the exploitation of multiple computing units (e.g., CPU-cores). In the field of Parallel Discrete Event Simulation (PDES), a classical way to exploit such scaled up computing power is based on partitioning the simulation model into separate objects, each one representing, in some sense, a work unit assigned to some CPU-core.

Motivated by the ever increasing diffusion and power of multi-core shared-memory machines, this thesis devises and develops an alternative paradigm for PDES, based on the idea that all the threads—so all the CPU-cores—running the PDES platform can fully share finer grain work units, namely individual events possibly bound to different simulation objects: this is the *share-everything* PDES paradigm. This new paradigm allows concentrating, at any time, the computing power—the CPU-cores on board of a shared-memory machine—towards the unprocessed events that stand closest to the current commit horizon of the simulation run. This fruitfully biases the delivery of the computing power towards the hot portion of the simulation model execution trajectory, thus favoring balanced advancement and providing optimized cross-object synchronization dynamics.

In our vision and design, sharing is not reflected into unaffordable thread coordination costs, since the core concept we rely on is that of non-blocking thread synchronization. In more detail, our first achievement is the design of a non-blocking conflict-resilient event pool used as the central building block for the share-everything PDES paradigm. Successively, on top of this pool we have built an innovative cross-layer scheduling mechanism to deliver events to be processed to threads in a non-blocking way—also in terms of accesses to the object states. Finally, we present the design of a kind of ultimate share-everything PDES system for multi-core machines that jointly provides:

1. fully non-blocking coordination of the threads when accessing shared data structures and

2. fully non-blocking speculative processing capabilities—according to the Time Warp synchronization protocol—of the events along simulation time.

We feel our achievements can bring a new perspective to the PDES community, and more generally to researchers interested in the design of HPC platforms in the era of multi-core shared memory machines, given that we slide to the opposite side with respect to traditional HPC approaches based on workload partitioning schemes across computing units.

Acknowledgments

If I look back at the last three years, many things have changed: I am the same one, but deeply different because I received something from everyone I met and that I will take with me in the future.

In the first place I would like to thank my Advisor Professor Francesco Quaglia for believing in me from the beginning and for giving me the opportunity to undertake this path. Standing by my side in every "battle", you gave me continuous stimuli and new points of view. This thesis and I owe a lot to your devotion in the research and the infinite patience towards me. You have been the best mentor who a Ph.D. student could ever wish for.

A word of gratitude goes to Professor Bruno Ciciani for having always taken care of us. You have supported our careers, protecting our studies and offering us the opportunity and the tools to transform the passion that we share in our future. Thanks because your lessons have been so valuable in my career as in my life.

I would also like Alessandro who, despite his very busy schedule between work and family, has always made time for me and my work. Probably, if I had not been encouraged by you to start this adventure, I would not be where I am now. Thank you for believing in me and in all of us, because without you we could not have hoped for a "lockless" future.

A special thanks goes to Romolo, my irreplaceable "schoolmate" and faithful co-author. Together, we shared victories and defeats, fervent discussions and long late-night calls. Thank you because you have been a constant and inexhaustible source of ideas and enriching perspectives; this journey would not have been the same without you.

I would like to thank Davide and Simone with whom I embarked on this adventure and who shared with me the trepidation of those who, swinging between fears and expectations, are preparing to start a new thrilling experience.

At the last, but not least, my thanks go to Stefàno, Stefano, Emiliano, Matteo and Pierangelo for being the best part of the several hours spent at DIAG, making it a better place. I am lucky to have met you in these years.

To all of us, because this is not the end but only the beginning.

As a good Italian as I am, I can not conclude without thanking my mother and my whole family. So: thanks to my parents Giorgio and Ida, my brothers Francesca, Andrea, Paolo and Laura, my aunt Anna and Flavia for supporting me with your warmth during this journey, and thanks to my nephews Alessandro, Arianna, Giolele and Celeste for having enriched my life with your loud cheerfulness.

Mauro

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

During the last decades, a direct implication of Moore's law [1] has given rise to a continuous increase of CPUs' performance, thanks to ever increasing clock frequency. This phenomenon allowed to execute programs faster just by (periodically) replacing the underlying hardware with one based on a higher-clocked CPU without the need for software and algorithms' reshuffling. However, since 2003, because of a physical constraint called *power wall* [2, 3], this trend is no longer in place, as shown in Fig 1.1. In particular, while up to that date the increase of frequency was balanced by a decrease in the required voltage, the extremely reduced size of transistors has posed a lower bound on voltage requirements, causing the chip overheating.

However, industries, academic institutions, as well as final users are still demanding for improved computing capacity in the machines. This has brought hardware vendors to exploit the ever increasing density of transistors to increase the number of CPU-cores, which translates into hardware explicit parallelism in the machine chip-set. A machine with multiple CPU-cores can perform multiple software tasks in parallel, thus hopefully providing better capability to execute software applications. Clearly, the benefits by this transition are not only experienced on the side of daily programs usage. Rather, large amounts of CPU-cores on board of a same machine has led off-the-shelf multi-core machines to become a good (or even excellent) opportunity for hosting HPC applications, like simulations. This happened also because of the ever increasing size of physical memory (RAM) packed in a same node. An aspect, the latter, that plays a central role in simulation applications since large/huge models, or models with very deep levels of details, not only require competing CPU speed, but also big in-memory (not disk/swap-based) processing capabilities to achieve reasonable performance.

On the downside, several literature methods and platforms for High Performance Simulation (HPS) have been conceived for scenarios where large computing power to be dedicated to a specific simulation application was essentially based on clustering machines via networking and message passing facilities. Consequently, several optimizations to make simulation platforms efficient were based on the assumption of distribution of the workload among loosely coupled computing nodes, as opposed to a single highly parallel multi-core machine. The multi-core era imposes, therefore, to rethink how simulation systems need to be conceived so as to let them become

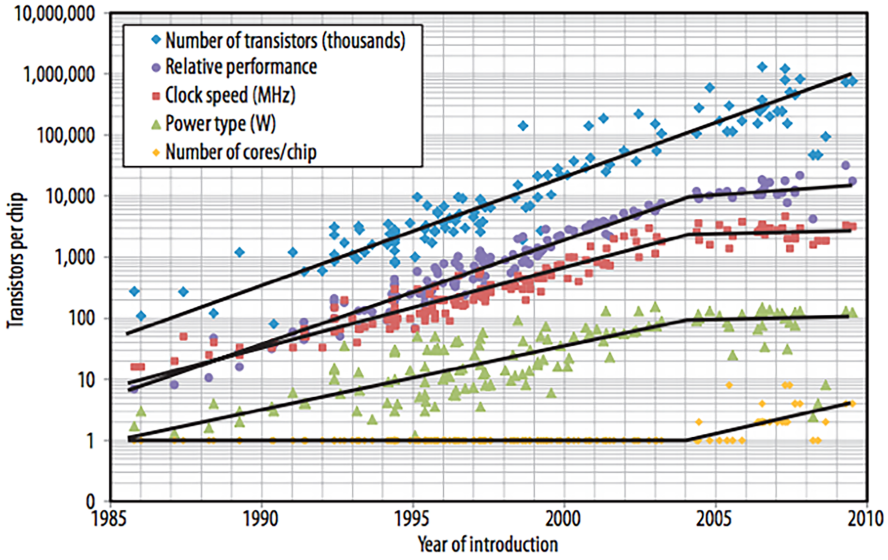

Figure 1.1. Trend of CPUs over the years

multi-core specific systems, rather than simply re-adapting their design originally tailored to a different kind of hardware platforms.

This thesis is devoted to study how to rethink simulation systems, particularly Discrete Event Simulation (DES) ones, to fully exploit the opportunities provided by multi-core machines. At the core of our proposals there is a very simple concept, namely that threads running a DES application on a multi-core machine can share the accesses to any portion of data representing the state of the simulation system—including application-level data representing the current state of the simulation model being executed. Hence the term "share-everything" simulation.

On the other hand, sharing data across concurrent threads is not a commonly (or historically) accepted way of devising parallel scientific applications, which have been typically implemented according to opposed paradigms, mostly relying on data partitioning. As we will show, such paradigms, when applied to DES, can be significantly overstepped by the share-everything DES paradigm, especially when facing hard to-cope-with workloads—such as those exhibiting non-regular runtime profiles—or workloads with very fine grain tasks.

The central problem to cope with when actuating the share-everything DES paradigm is how to make the overall simulation system truly scalable. In fact, data sharing requires coordination mechanisms across threads in order to maintain data consistency in face of concurrent updates. Our approach to scalable share-everything DES on multi-core machines will tackle the problem of guaranteeing scalability by rethinking the algorithms that the simulation engine will run in order to sustain the simulation workload. The objective is not only to outperform simulation systems based on data partitioning—or adaptations of them to better match the features of multi-core hardware—but also to enable DES to be run on future

generation systems, possibly equipped with ever largely increased CPU-core counts, while still providing improvements in performance.

Overall, when we think of a DES application run in parallel on a multi-core machine according to the share-everything approach, we need to jointly consider two potential impairments to scalability (and performance):

A) Threads need to coordinate with each other while concurrently accessing shared data structures in order to keep them consistent; this aspect relates to synchronizing threads along wall-clock time.

B) Threads need to coordinate with each other because of (potential or actual) causality constraints among the simulation model updates they are performing; this aspect relates to synchronizing threads along the simulation (virtual) time axis.

Therefore, an ideal share-everything DES platform should guarantee scalability along the aforementioned two dimensions (wall-clock-time and virtual-time coordination) in a combined manner. On the other hand, the core design principles for share-everything DES systems are to deliver the hardware computing power of a multi-core machine to the actual simulation application in a manner that is intrinsically more effective with respect to what could be achieved by simply re-organizing classical DES systems born to be run on clusters of loosely coupled processors, and not natively thought as of multi-core oriented ones.

This thesis is a journey along a series of new algorithmic approaches and implementations that cover all the above mentioned aspects, starting from the introduction of the baseline logic to be employed for driving the activities of threads within a share-everything DES system, and then going towards ever more complex solutions pointing to the ambitious target of extreme scalability (in conjunction with actual performance).

Our work brings advancements in two distinct areas of research, even though these advancements are presented as a combined result of research efforts in these areas for the achievement of cross-area solutions. The first one is obviously the one of DES systems and applications, in particular Parallel-DES (PDES), for which we will provide an overview of its basic concepts and reference methods in Chapter 2. The other area is the one of non-blocking thread synchronization, to be recalled in Chapter 3.

Our journey along the new ideas, algorithms and implementations will start in Chapter 4 with the introduction of the baseline concepts standing behind the share-everything PDES paradigm, as it is conceived in this thesis. In our view, share-everything PDES systems are oriented to addressing a specific problem affecting traditional PDES systems—or their reshuffling for multi-core machines—namely the inability to "optimally" react (or adapt) their runtime behavior to very irregular workloads, showing sudden skews of events targeting specific portions of the simulation model. They are also tailored to deliver in a better way the computing power to actual simulation tasks in scenarios where fine grain tasks characterize the simulation model execution. The presentation in Chapter 4 stands as a baseline picture, and does not dive into any specific solution enabling/implementing share-everything PDES.

We start presenting such solutions in Chapter 5, where the problem of thread coordination along wall-clock time is tackled via the introduction of a new non-blocking shared event-pool management algorithm based on the innovative concept of *fallback execution path* for event-pool operations. The fallback path, rather than the classical abort/retry path characterizing non-blocking thread synchronization algorithms in the literature, takes place (and is exploited) when conflicting accesses by concurrent threads to the same data portion (the "currently hot" part of the shared event pool) actually occur. This result makes a first realization of share-everything PDES suited for scenarios where the ratio between the execution time spent by threads in simulation-model specific tasks is a reduced percentage of the overall execution time spent by threads within the whole set of PDES-system tasks (application vs runtime environment tasks). A representative scenario is that of very fine grain tasks when processing simulation events through simulation-model specific software. However, by itself, such a result does not make the share-everything PDES system capable of reacting to kind of adverse workloads yet, namely those with clustering of activities (simulation model updates) on portions of the overall model state.

This capability is achieved via an innovative way of scheduling the tasks to be processed by concurrent threads, still via a non-blocking approach, which does not only cope with the current state of the event pool, but rather also with the current locality of operations by threads on the simulation model state. This is an innovative *cross-layer* non-blocking scheduling approach, which we present in Chapter 6.

Up to such result, speculation in the execution of the simulation model plays a role in our solutions, although limited. In Chapter 7, we finally present the design of an algorithm for share-everything PDES on multi-core machines which fully tackles the two coordination problems devised above (wall-clock time vs virtual time), thus leading to the *ultimate share-everything PDES system*. This solution becomes an additional instrument for the exploitation on multi-core machines according to the innovative share-everything approach in scenarios where the previous ones may result less effective, e.g., when dealing with reduced lookahead in the simulation model.

Each solution has anyhow its relevance, especially depending on the actual context where it would be used (lower or larger CPU-core counts, small vs large lookahead, more or less regular workload and so on). This is an additional motivation for our choice of releasing all of them–namely, their implementations—as free software to the community.

1.1 Publications

The innovative research contents presented in this thesis appeared in the following publications:

1. Davide Cingolani, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia. Mixing Hardware and Software Reversibility for Speculative Parallel Discrete Event Simulation. In *Proceedings of the 8th Conference on Reversible Computation*, RC, Springer-Verlag, July 2016.

2. Romolo Marotta, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia. A Conflict-Resilient Lock-Free Calendar Queue for Scalable Share-Everything PDES Platforms In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, ACM, May 2017.

3. Mauro Ianni, Romolo Marotta, Alessandro Pellegrini and Francesco Quaglia. Towards a Fully Non-blocking Share-everything PDES Platform. In *Proceedings of the 21st IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, DS-RT, IEEE Computer Society, October 2017.

4. Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini and Francesco Quaglia. The Ultimate Share-Everything PDES System. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, ACM, May 2018.

5. Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini and Francesco Quaglia. Optimizing Simulation on Shared-Memory Platforms: the Smart Cities Case. In *Proceedings of the 2018 Winter Simulation Conference*, WSC, IEEE Computer Society, December 2018. Invited paper.

All the research effort I spent during my doctorate lead me to also contribute to the technical content of the following publications:

i. Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia. Anonymous Readers Counting: A Wait-free Multi-word Atomic Register Algorithm for Scalable Data Sharing on Multi-core Machines *IEEE Transactions on Parallel and Distributed Systems*, TPDS, vol. 30, no. 2, pp. 286–299, November 2018.

ii. Emanuele Santini, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia. HTM Based Speculative Parallel Discrete Event Simulation of Very Fine Grain Models. In *Proceedings of the 22nd International Conference on High Performance Computing*, HiPC, IEEE Computer Society, December 2015.

iii. Romolo Marotta, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia. A Non-Blocking Priority Queue for the Pending Event Set. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, SIMUTools, ICST, August 2016.

iv. Romolo Marotta, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia. A Lock-Free O(1) Event Pool and its Application to Share-Everything PDES Platforms. In *Proceedings of the 20th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, DS-RT, IEEE Computer Society, September 2016. Winner of the Best Paper Award

v. Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia. A Wait-free Multi-word Atomic (1,N) Register for Large-scale Data Sharing on Multi-core Machines. In *Proceedings of the 2017 IEEE Cluster Conference*, CLUSTER, IEEE Computer Society, September 2017.

vi. Mauro Ianni, Romolo Marotta, Alessandro Pellegrini and Francesco Quaglia. A Non-blocking Global Virtual Time Algorithm with Logarithmic Number of Memory Operations. In *Proceedings of the 21st IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, DS-RT, IEEE Computer Society, October 2017. Candidate for (but not winner of) the Best Paper Award

vii. Romolo Marotta, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia. A Non-blocking Buddy System for Scalable Memory Allocation on Multi-core Machines. In *Proceedings of the 2018 IEEE Cluster Conference*, CLUSTER, IEEE Computer Society, September 2018.

viii. Romolo Marotta, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia. NBBS: A Non-blocking Buddy System for Multi-core Machines. In *Proceedings of the 19th International Symposium in Cluster, Cloud and Grid Computing*, CCGrid, IEEE Computer Society, May 2019.

Chapter 2

Parallel Discrete Event Simulation

Simulation is the art of mimicking the evolution of some real world or envisioned phenomenon over time. This takes place by describing the phenomenon via a *model*, which is formed by a set of variables and functions determining transitions in the state variables. The action of carrying out the simulation is also referred to as *model execution*. Nowadays, there are numerous applications where simulation is employed, for both analysis and design purposes: traffic flow, chemical reaction, buildings evacuation, biological system and so on. In particular, simulation is exploited for multiple goals, like events prediction [4], parameters' optimization and what-if analysis, as well as scenarios exploration.

Simulation applications fall in two main categories: *continuous* simulations and *discrete event* simulations. As for the former category, the imitated phenomenon evolves along a continuous time-axis, and the model-update functions rely on the extensive use of mathematical formulas describing the relation between the values of the state variables along time. As an example, if we consider a circuit made up by transistors, resistors and capacitors, the behavior of such components along-time is known to be captured by a set of equations. A continuous simulation uses these equations, in the context of the specific components' environment and connectivity leading to the circuit structure, to produce a continuous graph which accurately reflects the circuit state evolution along time. Unfortunately, mathematical equations used in continuous simulation applications can be computationally intensive for being solved, which makes simulation a hard task to be carried out especially in presence of thousands of interconnected elements. For this reason, continuous simulation may be slow and consequently used only to simulate a relatively small number of components which are described at a low level of abstraction.

The focus of this thesis is on the opposite class of discrete event simulation, whose baseline concepts are provided in the next section.

2.1 Discrete Event Simulation

In *Discrete Event Simulation* (DES), also known as event-driven simulation, the evolution of the imitated phenomenon is driven by a sequence of events—possibly updating the state variables used to model the phenomenon—which occur at discrete-time instants. More in detail, these events are referred to as discrete since they are characterized by an impulsive duration, meaning that the starting time of the event overlaps with its ending time, leading to an instantaneous change of the state of the modeled phenomenon. Overall, in DES the state-updates that are carried out along model execution are dispersed, and no state change ever takes place in between the occurrence times of two subsequent discrete events.

An example of discrete-event simulation can be given by modeling the evolution of a chemical compound. In such scenario the only significant events, able to produce a change in the state of the modeled phenomenon, are the actual chemical reactions occurring at specific points in time. Of course, considering the counterpart real-world phenomenon, many features of the compound could change between those reaction events, but since they are not considered as relevant in the model—in terms of their impact on the final outcome of the simulation—they can be dropped. This is an important aspect, which makes DES a method with intrinsic baseline ability to provide speed while executing the simulation model. Further, the discrete-event-based approach can be often applied to model (or approximate) continuous phenomenon, e.g., in order to speed-up their simulation. In these cases, we talk of discretized models.

Historically, DES has been approached by two different (although related) perspectives: a formal one, oriented to some clear-cut definition (or specification) of DES models [5, 6], and a methodological one, which is instead more suited/oriented for the actual development of simulation systems and applications [7]. The second perspective is the one targeted in this thesis, for which we provide an overview in the next section.

2.1.1 Methodological approach to DES

The software that forms a simulation application can be seen as the composition of two main parts: the first one is the implementation of the *simulation model* describing the phenomenon to be imitated. Here we have the data structures used to represent the model state and the software functions that access these data structures for carrying out the execution of the events. The second one is the *simulation kernel*, namely the part which takes care of driving the model-execution, namely of driving the coherent and correctly sequentialized activation of blocks of code included in the first part. While an encoded simulation model is devised to be run on top of some (generic) simulation kernel, a lot of research effort has been spent to build simulation kernels that can reveal efficient (e.g., fast) in differentiated simulation scenarios (namely, when supporting different simulation model implementations).

Although being oriented to practical aspects and real simulation models/kernels realizations, the methodological approach to DES is still based in some minimal specification of a few main components:

- a set of *simulation states* S representing the static part of the simulation, namely discrete snapshots that can be taken along model-execution, and that are represented by a set of variables;
- a set of *events* E, corresponding to the "causes" of state transitions in the simulation model;
- a *transition function* $\sigma(s, e) : S \times E \to S$ that, at the time when an event $e \in E$ occurs, starting by the current state s, and based on the nature (also referred to as *content*) of the event, determines the new simulation state s'.

Mapping the definition of these baseline components to a software-based realization of the simulation, a DES model implementation can be seen as a set of *event-handler* functions (or, more simply, event handlers) to be used as callback functions that pass control to the code portion implementing the simulation model. The event-handlers capture events in input and, depending on the content of the events, they produce effects in the simulation model state. Overall, the set of event handlers are nothing more than the implementation of the aforementioned transition function $\sigma(s, e)$, that, taken the current state and the received event, produces the new state for the simulation.

The event handlers are activated by the underlying implementation of the simulation kernel, which also works as a container of the events to be passed in input to the simulation model. As it will be clear in a while, the set of events contained in the simulation kernel can be dynamic, namely new events can be dynamically produced while model-execution is carried on. However, this must occur following specific rules, based on *causality* concepts, used to characterize the correct evolution of the simulation model execution. These same concepts also impact how (say, in what order) the simulation kernel needs to deliver the events to the event-handler functions invoked as callbacks.

We recall that each discrete event e has a corresponding instant of simulation time T_e for its occurrence, also known as timestamp of the event. When the event e occurs, say it is passed by the simulation kernel in input to some event handler for being processed, the simulation time—the current time as seen by the model implementation—is advanced to the value T_e. In fact, something is occurring in the simulation model exactly at that time. As for this aspect, it is very important to distinguish this "abstract" notion of time from *wall-clock time* (WCT), which represents the passage of time in the real world, exactly as perceived by human beings—and so by a CPU-core via some register regularly incremented at each machine cycle. Although WCT has nothing to do with the simulation time value along model execution, it is a fundamental means for the assessment of the speed according to which we can advance the simulation while we carry out model-execution on a computer system.

At the begin of the simulation, one or more events are injected in the simulation system to allow model-execution to start. Subsequently, new ones can be generated as a result of the execution of others, thus allowing the perpetuation of the simulation. This is the dynamic event generation aspect we were referring to in a previous paragraph. This aspect is also strictly linked to the notion of *causality* among events—hence among transitions in the simulation model state. Formally, if

an event e during its execution generates an event e', it means that the generated event e' *causally depends* on e. This implies that, for the nature of the causal order between events located in time, if an event e with time T_e generates an event e' with time $T_{e'}$, it follows that $T_{e'} \geq T_e$. In other words, e cannot affect the past of the model-execution trajectory; rather only the present and the future. Moreover, in order to guarantee a causally consistent evolution on the model-execution, the simulation kernel has to schedule the processing of events in the order defined by their timestamps.

Anyway, it can still happen that two different (dynamically produced) events, namely e' and e'' will have timestamps such that $T_{e'} = T_{e''}$. In this case we cannot assume these events are ordered by their timestamps, an aspect that opens the problem of selecting what event would need to be executed before, and of how to make this choice deterministic. The latter point is clearly related to the *reproducibility* of the simulation results, namely the capacity to achieve exactly the same outcome in different executions entailing the same initial events as well as the same initial state of the simulation model. In order to guarantee reproducibility many aspects have to be treated (e.g., determinism in the production of random numbers impacting the content of dynamically produced events, as well as the model state), including obviously the definition of an order among contemporary events, e.g., relying on the use of a *tie breaking* function [8]. All of these solutions constitute a classical enrichment of the simulation system.

Another important aspect to consider is that the simulation state can be partitioned in distinct *simulation objects*. The use of simulation objects, although not mandatory, represents a useful extension that can provide the model developer with improved semantic. It is also a means for linking the simulation model implementation with concepts characterizing object-oriented programming (such as modularity and encapsulation). A simulation object is used to represent a portion of the model, which can be, e.g., an *agent* in agent-based simulations, or a *spatial region*, such as a portion of a building, or the cell covered by an antenna. In this way, for the model developer is possible to describe the whole imitated world, concentrating time by time on individual portions. Each portion can have its associated implementation of the event handlers. On the other hand, events will trigger the execution of one or another handler, associated with the different objects, depending on their content.

As noted above, the simulation kernel typically acts as the container of the events that need to be eventually dispatched for processing via the invocation of the event handler. The structure used to store the events to be processed is called *event pool* or *pending event queue*. All the events have to be executed in the order established by their timestamp, so the event pool typically is organized to return, each time a new event-dispatching need to occur, the event with the smallest timestamp. This is also referred to as the not yet processed event with the *highest priority*. For this reason, priority queues are often used at the level of the simulation kernel, exactly ordered according to the events' occurrence time [9]. The event pool can be implemented in different ways, a concept that is also related to studies which have demonstrated that the most efficient solution—e.g., the one with lowest cost/overhead of the operations for managing the insertions and the extractions of the events—can depend on the nature of the simulation model. Some of the most diffused and effective solutions

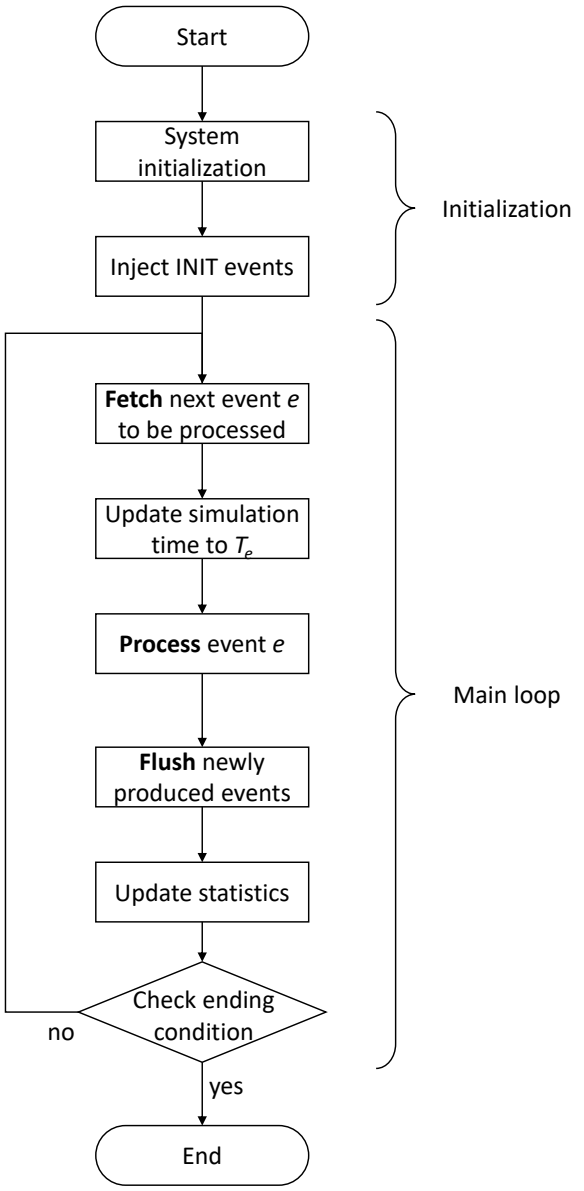

Figure 2.1. Simulation kernel flow chart

are the linked-list, the skip-lists [10], the splay tree [11], the calendar queue [12], and the ladder queue [13].

In some cases, it is important to stop the simulation when a particular condition occurs, with the goal to capture data (e.g., the current state value of the simulation model) at the time of the condition occurrence. For these reasons, the moment (in simulation time) at which the execution of the simulation has to stop needs to be correctly determined in order to gather meaningful information and statistics from the simulation experiment. To this purpose, it is important to define a particular condition, called *ending condition*, whose validity (or non-validity) needs to be as-

sessed, ideally after the execution of each event, in order to decide if the simulation needs to terminate. The ending condition can be defined in different ways, e.g., the occurrence of a specific event, or the reaching of a particular value in the simulation state. Clearly, a simulation time range, whose end point needs to be reached, is still an admissible choice, depending on the specific application.

On the basis of the above outlines, the simulation kernel is the component that orchestrates the advancement of the simulation. The simulation kernel procedure is broadly divided in two phases, a first one used to set-up and initialize the simulation system, and a second one in which the actual simulation is carried out. A typical flow chart of the simulation kernel procedure is shown in Figure 2.1.

During the initialization phase, the simulation kernel drives the sets-up of the whole simulation environment: the (initial) resources required to sustain the execution of the simulation and the basic variables and data structures, used to manage the event pool. Once everything is ready, the simulation initial events (INIT events) are injected in the system. In common implementations this requires interaction with some specific callback implemented at the level of the simulation model component, since general purpose simulation kernels are agnostic of the model being executed on top of them, hence of the events initially firing state transitions. The same callbacks also do part of the job of initializing the state of the simulation model, exactly because the kernel can be agnostic of what shape the state needs to have in a specific application. So, they initially setup/store the simulation state (possibly separated in multiple simulation objects)—this state can anyhow grow and then be shrunk again along model-execution depending on the actual implementation of the event handlers and on the impact of executing them on the layout of the simulation state in memory.

Once the simulation environment is set up and the simulation model is initialized, the real body of the simulation job starts. This is carried out in a simple main-loop cycle, which comes up with the processing of an event at each iteration, until the end of the simulation is reached. In particular, in each iteration the unprocessed event e with the smallest timestamp T_e is fetched from the event pool, the simulation time is updated in order to reflect the advancement in the simulation and, thus, the event e is processed relying on the proper event handler offered by the simulation model implementation. During the execution of this event, new events can be produced and, before moving to the next iteration, these events are flushed to the event pool. At each iteration (or periodically for reducing the associated costs) the ending condition is checked in order to see if the simulation has to be stopped. Moreover, at the end of each iteration, it is possible to perform statistics update, useful to audit the model execution, as well as the runtime behavior of the simulation kernel and of the overall set of the involved software modules.

2.2 Parallelization methods

Implementations of DES simulation systems, based on the concepts we introduced in the previous section, have a core limitation. This limitation is related to the fact that such concepts do not kept into account parallelism. Therefore, any task—either a simulation-kernel task or a simulation-application task—is sequentialized

with respect to the execution of all the others. We may barely refer to this kind of implementations as single process/thread simulation systems.

The historical modification of DES, aimed at including design concepts where parallelism is included, is known as Parallel Discrete Event Simulation (PDES) [14, 15]. PDES is a methodology that allows to execute a DES model on top of a computing system entailing multiple computing units, and having the capability of doing work related to different tasks in parallel.

The feature at the core of PDES is the aforementioned notion of simulation objects, which was not strictly required by classical DES (although it may help by the side of simplifying the job of implementing the simulation model), but rather becomes mandatory in its parallel incarnation. In PDES the simulation object is named *Logical Process* (LP) and represents the container of a disjoint portion of the overall state of the simulation model.

More in detail, in PDES the simulation state is partitioned in N different LPs, uniquely identified and labeled as $LP_0, LP_1, ..., LP_{N-1}$, with the state of each one, composed by private variables (or private memory areas), completely disjoint by the other ones. Thus, the simulation state can be usually represented according to the following formula that relates it to the states of the individual LPs:

$$S = \bigcup_{i=0}^{N-1} S_{LP_i} \wedge S_{LP_i} \cap S_{LP_j} = \emptyset, \forall i \neq j \tag{2.1}$$

where S_{LP_i} is the state portion associated with LP_i.

The LPs are somehow autonomous entities, each one implemented with its own event handlers for managing the occurrence of events involving the portion of the model they represent. The relations between the simulation kernel and the simulation model are quite similar to the ones discussed for the case of DES: the kernel is still in charge of keeping the events—thus acting as event container—to be processed and passing them to some event handler which implements the simulation application logic. However, the notion of task sequentialization typical of DES is overstepped by the fact that the PDES methodology enables event handlers of the different LPs to be triggered for execution in parallel, clearly, if we have enough hardware resources to do it; for example, we can deploy the different LPs—namely their software implementation—on top of multiple CPUs (or CPU-cores).

However, with the aforementioned partitioning, the PDES methodology also requires mechanisms to support the fact that some event affecting a portion of the overall model state can then have effects on another portion. As an example, if we are simulating an evacuation plan of a given area, and the area has been partitioned into regions in the simulation model in order to assign each region (partition) to a different LP, we need mechanisms to let the PDES application manage the scenario where an entity that leaves a region may enter another region. This situation involves evolving the states of two different adjacent regions in a coordinated manner along model-execution.

To cope with this issue, PDES introduces a notion of message-exchange across LPs. In particular, each time an LP experiences a state change—the exit of the entity in our example—which has effect on the state of another LP, the former needs to dynamically create an event—coding the entrance of the entity into the destina-

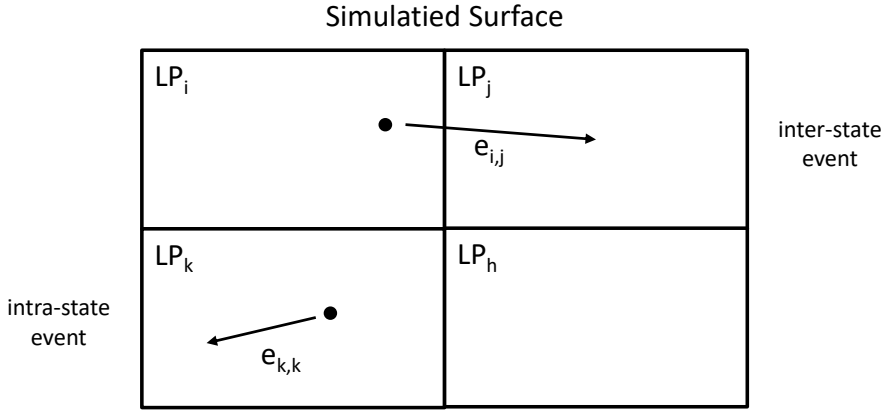

Figure 2.2. Message exchanged across LPs

tion region in our example—and send it via a message towards the *destination* LP. More technically, still referring to the previous example, if some entity is leaving a region represented by LP_i to enter into another one represented by LP_j at time t, when processing the exit event of the entity, LP_i needs to create an event $e_{i,j}$ that is sent to the LPj, which is marked with timestamp t and contains information about the change of location of the entity, as shown in Figure 2.2. On the other hand, it is still possible that an individual LP dynamically produces new events destined to itself, as shown by the event $e_{k,k}$ in Figure 2.2, which is used, e.g., to update the simulated position of some entity within a same region represented by LP_k.

As the reader may have noted, with such new paradigm, LPs can process events in parallel, as if they were independent simulators of portions of the imitated phenomenon. On the other hand, they are coupled by message-exchanges, leading to cross-LP event scheduling—like for the case of the event $e_{i,j}$ in Figure 2.2. In such a scenario we are faced with the big problem of determining what should be the actual dependency graph—the partial order—across executed tasks (events) in order to maintain a consistent model-execution trajectory. Essentially, the partial order among tasks' execution that would need to be respected should provide the illusion that the events produce state transitions in the simulation model—with each transition involving the state of some LP—as if they were executed according to the order of their timestamps.

Clearly, enforcing such a *global ordering* at runtime would vanify any attempt to exploit increased CPU(-core)s count for running the simulation experiment, since we would fall in the scenario, quite similar to the one of DES, where only one task at a time can be active. This would no longer be a parallelized execution.

Fortunately, the PDES methodology provides us with a sufficient condition telling that a model-execution where events are completely sequentialized based on their timestamps is equivalent, in terms of state trajectories of the LPs forming the model—hence of the final global value of the simulation model state—to one where the processing order of the events based on timestamp is ensured *locally* to any LP. This leads to the scenario where we there is not a single value of the current simulation time, but rather to one where each LP is associated with its own view of simulation time advancement, named *Local Virtual Time* (LVT). However, the

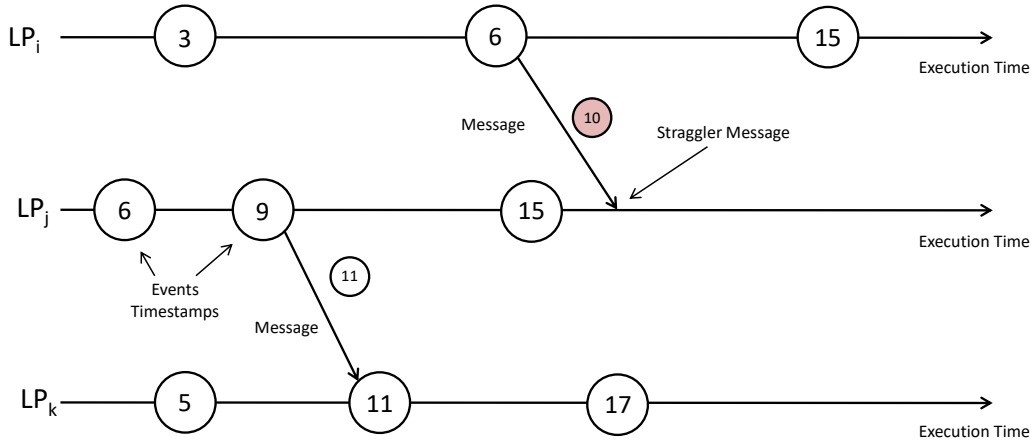

Figure 2.3. Example of Causality Violation

advancement of the LVT needs to comply with the local timestamp ordering property when applied to the whole set of events destined to the LP. This set includes the events dynamically produced by other LPs, like the event $e_{i,j}$ in Figure 2.2.

To better clarify this aspect, let us consider the example timeline in Figure 2.3. Here we have three LPs, for which the processing of events is triggered in parallel along model-execution. As we discussed the correlation among the LPs are actuated via message-exchange and the messages carry events for the destination LP. In the example timeline we have two events produced by some LP and destined to another one. In particular, LP_j produces, while processing its event with timestamp 9, a new event destined to LP_k. If this event is included in the event-processing sequence by LP_k right after the event with timestamp 5 is processed, but before processing the one with timestamp 17, then for LP_k the local timestamp ordering rule is respected. On the other hand, the example shows how, for LP_j, we have a violation, since the event with timestamp 10, produced by LP_i for LP_j when LP_i processes its event with timestamp 6, is not included in the processing sequence of LP_j in the right timestamp order. In particular, when processing such newly produced event by LP_i, LP_j has already processed an event with timestamp 15. This leads LVT_j to be moved ahead of the timestamp of the event incoming from LP_i, which violated the local timestamp ordering rule. This anomaly is also known to as *causality violation*, or causality error. A PDES system needs therefore to cope with causality violations, particularly by avoiding that these violations can really be materialized along model-execution. Otherwise we might experience an incorrect model-execution trajectory. This is the well-known *virtual-time synchronization* problem, which affects how the LPs need to coordinate their actions along simulation time in order to exactly respect the local timestamp ordering rule. Note that here we are not talking of processes/threads synchronization, since virtual-time synchronization only deals with the issue of coordinating the execution of tasks (events) occurring at the different LPs, independently of what process/thread will be in charge of running the event handlers that lead to task execution. This is true under the baseline assumption, characterizing the classical PDES methodology, that each individual LP is still

a sequential entity, so the tasks involving it are executed along a sequence, and with no overlap with each other.

In order to face this problem, with the goal to ensure the correctness of the simulation, different approaches are available in the literature [16, 14]: *conservative*, *optimistic* or *adaptive*. The conservative approach [17, 15, 18] guarantees the correctness of the simulation by ensuring that an event is executed only when it is certain that no other event, with smaller timestamp, will income destined to the LP. The optimistic approach [19] follows the opposite scheme where all the events can be processed at some LP at any time, independently of events that will be dynamically produced by other LPs and will income. On the other hand, in the scenario of an incoming *straggler* event—one with timestamp in the past of the LVT of the destination LP, see Figure 2.3—causal consistency is recovered via proper actions based on *computation undo* schemes. Overall, the conservative approach *prevents* inconsistencies while the optimistic approach *recovers* from inconsistencies if they actually occur. Finally, the adaptive approach [20, 21] combines the conservative approach and the optimistic approach according to different schemes.

2.2.1 Conservative virtual-time synchronization

As hinted, the conservative approach is based on preventing the occurrence of causality violations. This requires algorithms and mechanisms for determining when an event destined to a given LP can be safely processed since we are sure that no other event will ever income with a smaller timestamp. In other words, we need to assess when some event currently stored at the level of the simulation kernel will not be eventually "hit" by some straggler event once processed.

Solutions to this problem have been originally proposed in [17, 15]. In these proposals, it is in charge to the simulation model developer to notify explicitly which LPs are directly interconnected. At the level of the simulation kernel, a channel is defined for each (directional) pair of communicating LPs (namely, for each communication link specified by the model developer). Each channel is a queue marked with the smallest timestamp of the events currently traveling along the channel (those to be processed), or of the last one already extracted and executed, if the channel is currently empty.

With this support, the simulation can be carried out by:

- selecting the event with the minimum timestamp from the incoming channels and triggering the execution of this event at the destination LP; or
- temporarily blocking the LP if the queue marked with the minimum timestamp is empty—in this case, selecting an event from a non-empty queue could produce local timestamp ordering inversions (causality violations), if an event with smaller timestamp is received later on the currently empty channel.

If, by one side, this approach is enough to ensure that nothing wrong happens, on the other side it does not guarantee the advancement of the simulation. More in detail, it can lead to deadlocks caused by the absence of new messages on an empty channel. Let us assume a simulation model that adheres to the LPs' interconnection scheme in Figure 2.4. There is a communication channel between LP_1 and LP_3,

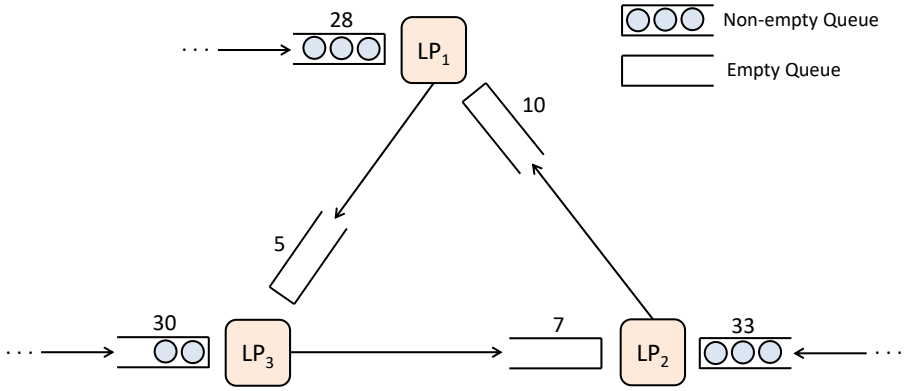

Figure 2.4. Deadlock in Conservative Synchronization

there is one between LP_3 and LP_2 and there is one between LP_2 and LP_1. Further, each one of the three LPs has another communication channel with other LPs. All the queues associated with the three channels connecting these LPs are empty, then there is a deadlock situation. This happens because, even though there are additional queues that continue to store incoming messages from other LPs, the simulation kernel keeps on checking the empty queue of incoming events to each of the three LPs (thus maintaining the three LPs blocked in simulation time) because it is marked with the lowest timestamp.

A solution to this problem is based on using *null messages* [22, 15], namely messages characterized by an empty body and a timestamp. The objective of these messages is the one of indicating that no events with smaller timestamp (than the one of the null message) will ever travel along a channel. This allows coping with deadlocks since a null message can break the cycle of waits for incoming information along channels. However, the volume of null messages can be significant in scenarios where LPs can send no real events to others for a significant portion of the simulation run. The management of null messages can lead to runtime costs, which are an additional source of performance loss. We recall that such performance loss can be experienced in conservative synchronization also because of block phases of the LPs, especially when the number of non-blocked LPs at any time is lower than the actual number of CPU-cores we may have available in the computing system. This is a scalability problem affecting conservative synchronization [23].

On the other hand, the specific peculiarities of the simulation model can help tackling the aforementioned problems. In particular, for several models it can be reasonable to think that the new events generated by the execution of an event e at time ts, cannot be marked with a timestamp smaller than $ts + L$. This means that, for the specific model, there is a minimum simulation time interval between an event occurrence and the occurrence times of the ones generated by it [24]. If this is true then the indications that travel along channels, particularly the timestamp of some null message, not only indicates that nothing will income up to that time, but rather that nothing will income up to that time plus L. In fact, null messages are typically marked with the minimum time for the occurrence of something not yet occurred (some not yet processed event) at the source LP.

For example, considering again the area evacuation model where there are entities moving to reach the exit point, it is possible to assume that, when some entity enters in a region at time ts (an action that is materialized by the execution of an event at the LP representing that region), it will take some minimum time L before reaching the next region. This value L takes the name of *lookahead* and can be exploited in order to identify, at any time, more events incoming from input channels as safe to be executed, since they will not eventually be hit by some straggler (related to the entity entrance into the region in our example). This reduces the risk of blocking scenarios, and also leads to the need for a reduced volume of null messages to travel across the LPs.

However, it is not always possible to define a lookahead value for the simulation model. In fact, in some applications we may have events occurring at some time that generate new events occurring at the same time (like a kind of instantaneous effect propagation). Also, the lookahead value could be small, thus having limited impact on avoiding blocking scenarios.

2.2.2 Optimistic virtual-time synchronization

As discussed, conservative virtual-time synchronization can lead to limited exploitation of the hardware resources provided by the underlying architecture, especially because of blocking scenarios leading running processes/threads to the impossibility to carry out, for a while, the execution of event handlers for events hitting specific LPs.

The optimistic (speculative) virtual-time synchronization approach, originally conceived in [19] under the name *Time Warp*, is based on the idea that, if there are available resources, rather than leaving them unused, these should be employed to execute tasks that could be (they have the potential of being) correct, thus useful. Basically, there is a bet done before on the fact that this work can be really useful, discovering only later if it was actually rewarding, thus if the gamble paid off. If the work done is discovered not to be correct, the achieved outcomes are discarded, as well as the side effects on the model-execution trajectory, thus resuming the execution flow from a correct (say, causally consistent) point. Otherwise, they are confirmed by simply continuing the execution and making the side effects of the done work on the simulation model state permanent. The main advantage of this approach is related to the fact that the resources used to perform the speculative execution would not be used otherwise by conservative synchronization in the case of LPs blocks.

We note that speculative processing is not a prerogative of optimistic virtual-time synchronization. In fact, it is exploited in a number of other fields ranging from superscalar pipeline processors [25]—where speculation (also termed as out-of-order pipeline) is used to fully exploit all the components of a CPU without waiting the outcome of the previous instructions—to file prefetching [26], and to concurrency control algorithms in transactional systems [27]. In any case, speculation in PDES has its own peculiarities.

When executing the simulation model under speculative synchronization, the next event already available (at the simulation kernel) for processing at some LP is with no delay eligible for CPU-dispatching. Clearly, this is the event with the

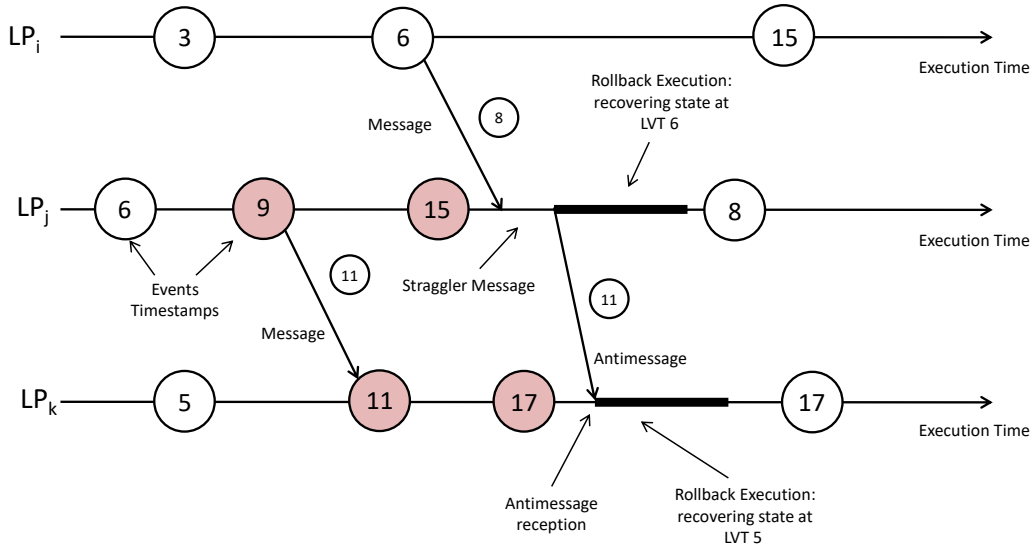

Figure 2.5. Rollback Operation

minimum timestamp the simulation kernel is already aware of, which is not yet processed by that LP. We note that such an event might be a straggler—it might have a timestamp lower than the current LVT of the destination LP—just because event processing is not subject to previous assessment of event safety with respect to local timestamp ordering. When the simulation kernel needs to CPU-dispatch the execution of a straggler, the state of the target simulation object has to be restored to a consistent value, meaning that the corresponding LVT can no longer be greater than the timestamp of the straggler event.

Let us consider the scenario in Figure 2.5. Here, when LP_j receives from LP_i the message containing the event e with timestamp $T_e = 8$, the simulation kernel checks whether it is smaller with respect to LVT_j and, if such condition is true, actions are taken to rewind the execution trajectory of LP_j to the time of the previously executed event r having the higher timestamp t_r such that $t_r < 8$, that is $t_r = 6$. In this way, the execution of the event e is materialized as occurring in timestamp order locally to LP_j. In fact, performing such a rewind operation, commonly termed as *rollback* of LP_j, the effects on the state of LP_j produced by the events between the LVT_j value prior to the rollback and t_r have been discarded. However, during their executions these events could have dynamically generated other events, possibly sent towards other LPs via messages. In particular, the execution of the event with timestamp 9 at LP_j has generated a new event with timestamp 11 destined to LP_k, which has no more reason to exist since its parent has disappeared from the execution trajectory of LP_j as soon as the rollback is performed. Of course, the execution of the event with timestamp 9 could still generate an event with timestamp 11 once the execution of LP_j is resumed from the correct state and then continued, but in such case it will represent a completely new incarnation of the event.

We also note that, the rollback operation involving LP_j, has produced in turn an inconsistent simulation state also by the side of LP_k. Thus, LP_k has to be informed

by LP_j that the event (or the specific incarnation of it) with timestamp 11 no longer exists. In order to fulfill this objective, the optimistic synchronization mechanism relies on the notion of *anti-message*. This is a copy (or a digest) of the originally sent message, but with a kind of negative sign, indicating that the originally sent message needs to be annihilated.

On the recipient side there are two possible scenarios: the event to be annihilated has not yet been executed (it is stored by the simulation kernel but no CPU-dispatch involving it took place), or the event has been already processed. These scenarios can be distinguished one from the other by relying on the LVT of the recipient LP, similarly to what happens upon the arrival of a straggler event. If the timestamp associated with the received anti-message is greater than LVT_k, the original event figures as not yet been executed along the current LP_k's trajectory, thus is enough to discard (annihilate) its positive incarnation. On the other hand, if such timestamp is smaller than LVT_k, the corresponding event figures as being executed, thus bringing the evolution of LP_k on a wrong trajectory. In this case, LP_k has to rollback too to a consistent state (as it happens in the example in Figure 2.5). This phenomenon, namely the fact that a rollback of an LP has produced a rollback on other LPs, is named *cascading rollback*.

The operation of bringing back the state of an LP to a previous value is crucial in optimistic virtual-time synchronization. For this reason, state restoration in speculative PDES is one of the most studied topics. In the literature there exist two main approaches to support rollback operations of the state of the LP. The first one relies on the concept of *save state & restore*, and it is the original rollback approach proposed in [19]. The second one relies on the concept of *reverse computation* of the executed simulation events, and assumes that, starting from an event, it is possible to generate a *negative event*, able to completely revert the execution of the original one, thus completely reversing its side effects on the state of the LP. These approaches are discussed in the following paragraphs.

State saving

This technique is based on the idea that is possible to store, after the execution of each event e destined to LP_i, a copy of its simulation state, associated with the event timestamp t_e, $S_{LP_i}(t_e)$, called *snapshot* (or also *checkpoint*) [19]. So, when a straggler event or an anti-message is received with timestamp t_s, a rollback can be performed by identifying and restoring the simulation state $S_{LP_i}(t_r)$ such that t_r is the highest timestamp among the ones associated with the different snapshots and such that $t_r \leq t_s$.

The state saving operation can be performed transparently by the simulation kernel, or can be done involving the simulation model code. In any case, this operation requires to identify the portion of memory where the simulation state of the LP is stored and to execute specific software modules for creating the state copy when taking a snapshot (or reloading the taken copy when rolling back the LP). We note that copying the whole memory region where the LP state is stored after the execution of each event can be a time and memory consuming operation, especially for large size of the state image. In order to reduce such costs two different orthogonal approaches have been studied: 1) reducing the amount of snapshots

taken during the execution; 2) reducing the amount of information stored for each snapshot.

As for the first approach, it is important to consider that, in order to restore a state that is consistent with respect to timestamp t_s, any state $S_{LP_i}(t_r)$ such that $t_r \leq t_s$ is a good candidate, providing that the sequence of events executed between t_r and t_s is still available (e.g., it is still stored by the simulation kernel). In fact, it could be possible to restore any state in the simulation just starting form whichever past snapshot and re-executing again all the events with simulation time in the interval between the snapshot time and the time of the state to be restored. Therefore, in order to reduce the cost associated with taking a snapshot and, at the same time, reduce the memory footprint, it is possible to reduce the frequency with which the state saving operation is performed, as an example increasing the number of events executed between two consecutive snapshots. With this approach, named *Sparse State Saving* [28], when a consistent state at timestamp t_s has to be restored, the last snapshot $S_{LP_i}(t_r)$ such that $t_r \leq t_s$ is reloaded and then all the events e such that $t_r \leq t_e \leq t_s$ are executed in timestamp order, performing the so called *costing forward* phase of the rollback. It is important to take in mind that the events re-executed during the costing forward phase are not part of a different trajectory, with respect the previously followed one. Therefore, such re-execution is only an artifact of the state restore operation, and needs not to actually re-generate new events that were already produced in the original execution. In other words, the coasting forward phase is a kind of *silent execution* only aimed at realigning the LP state to the correct final value in a rollback operation. This can be reached by having the simulation kernel to filter these new events during the costing forward phase, thus avoiding resending them towards the destination LPs.

Moreover, since the costing forward phase has to produce the same simulation state reached during the original execution, it is important to have a deterministic behavior in the event execution while performing costing forward operations. This means that the whole software architecture needs to provide piece-wise-deterministic behavior, meaning that if some execution step may be subject to non-determinism, then some action needs to be taken to make this deterministic, e.g., via logging of the information that lead to the specific final outcome for that step. This aspect is crucial when the model developer relies on some *pseudo-random number generator* (PRNG) to drive the updates of the LP state. In particular, such pseudo-random number generator has to support rollback operations by producing the same sequence of random values while executing the same simulation trajectory multiple times, e.g., in coasting forward phases. This can be done either storing the seed associated with the PRNG within the snapshot of the LP state, or implementing a generator able to undo changes of its internal state.

The period among two consecutive snapshots (or the specific points at which state snapshots need to be taken) can be fixed (periodic state saving) or variable (adaptive state saving). This allows for tuning the state snapshot subsystem in order to optimize the tradeoff between the costs spent in taking the state snapshots and the costs for reconstructing a state that was not saved via coasting forward [29, 30, 31, 32, 33, 34].

Another technique that can be used to reduce time and memory footprint of snapshot operations is the one of reducing the amount of information saved with

each snapshot. This technique is based on the observation that, depending on the specific logic implemented by the simulation model, the most part of the LP state could be left untouched (e.g., only accessed in read mode) by the execution of some event. So, it would be possible to save only the touched portion to achieve backward state-reconstruction in a rollback phase. This technique, known as *Incremental State Saving* [35], aims therefore at reducing the size of the single snapshot by storing only the modified portion of the state, also reducing the execution time to perform the save and restore operations. The state portion modified by the execution of an event can be explicitly notified by the model developer or can be transparently identified by the simulation kernel. In particular, effective techniques are the ones relying on transparent code instrumentation [32, 36, 37, 38, 39, 40]. These approaches parse the (assembly or source) code of the simulation model to identify memory updating instructions (or statements), in order to add a call to a module that makes a copy of the old value stored into the LP state before the actual update takes place. Thus, when a straggler event or an anti-message arrives triggering a rollback operation, the chain of logged values is backward scanned, realigning the memory to some previous consistent state. This technique can be made completely transparent for the programmer, who has not to modify the original simulation model implementation since this mechanism can be automatically supported by the simulation kernel and by the compiling tools associated with it for generating executable versions of the simulation model code.

Incremental and non-incremental snapshots have been also combined with each other. An example is provided in [41] where, relying on the instrumentation of the assembly code of the simulation model, each time a memory update is identified, the old value is not directly stored, rather, a dirty bitmap is updated in order to keep track of the portion of memory modified. Thus, periodically a state snapshot is taken by saving the memory areas updated with respect to the last (non-incremental) taken snapshot. In [32] the combination of incremental and non-incremental snapshots is based on a dynamic and simulation-model transparent switching technique, which selects the best suited operation mode depending on the specific phase of the LP execution.

Reverse computation

The reverse computation approach is based on the idea that it is possible to automatically generate for each event e a *reverse event* r_e able to restore the previous state value by executing in reverse order the operations of its corresponding positive version [42]. Thus, storing a chain of revers events for each LP_i while advancing in simulation time, it is possible to restore a consistent simulation state S_{LP_i} with respect to a timestamp t_s just backward scrolling such chain of reverse events (and processing them) until the first reverse event r_e such that $t_{r_e} \leq t_s$ is found.

A reverse event can be seen as a reverse ordered copy of the positive one, with each operation replaced by its inverse incarnation within the event handler (e.g., an addiction is replaced by a subtraction of the same value). However, a series of problems have to be faced in order to reproduce the exact reverse behavior. For example, by simply reversing the operations' order it is not possible to cover scenarios with variables driving branch conditions, since the values used by the

condition could be still to be computed. This problem can be easily faced by storing the result of the branching condition in the corresponding reverse event (as a minimal form of snapshot information associated with the reverse event).

Another problem is represented by the fact that not all the operations are reversible. This is the case of disruptive operations, namely operations whose execution fully overwrites via simple assignment the previous value, such as a memory copy operation. These situations have to be managed instrumenting the original code to store each time the previous value, generating an incremental fine-grain snapshot for each memory update [36].

Also if the reverse computation technique could introduce a significant overhead in terms of memory footprint (falling back to fine-grain state saving in case of events characterized by a large amount of disruptive operations) and rollback cost (requiring to execute the whole reverse event chain until the state to be restored), it can give an important reduction in the time needed by rollback operations if the state to be rolled back is near to the actual logical time of the LP. A solution coping with this problem, which is based on the dynamic generation of so called *undo-code-blocks*, has been recently presented in [43]. In this approach, reverse computation is mixed with classical snapshot-based approaches in an optimized manner. Also, machine code for reverse events is not based on the implementation of the reverse logic, but rather on the implementation of instructions that only revert the side effects in memory, packing the values to be restored as (immediate) operands of the reverse event code block.

2.2.3 Adaptive approaches

Conservative and optimistic virtual-time synchronization are opposite solutions that have been comparatively assessed in many works. While the conservative approach has proved to be able to efficiently parallelize models provided with good lookahead, the optimistic approach has been shown to offer good results even in the absence of lookahead information, as long as the rate of causality violations is not too high (going to frustrate the benefits of speculation). A third possibility in the literature has been offered by adaptive schemes, which try to combine conservative and optimistic virtual-time synchronization in various ways.

One of the most promising approaches in this direction is offered by the Local Time Warp approach [44]. In this solution, when an event is found to be unsafe it can be speculatively executed in two different ways. In particular, the two different execution modes are distinguished according to the locality of the event, namely the amount of LP possibly involved in a rollback due to the arrival of a straggler event. The most conservative mode speculatively executes events without forwarding the ones produced by these executions and destined to other LPs, until safety of these executions is certain. The second mode instead spreads the events produced during the speculative execution in a constrained manner, particularly by defining a window (in terms of simulation time or amount of executed events) within which the events are allowed to be sent.

The reliance on virtual-time windows to temporarily stop the execution of events at the LPs that already hit the upper bound of the window while processing events is another scheme used in, e.g., [45]. The idea behind these solutions is the one of

reintroducing block phases (proper of conservative synchronization) in a speculative processing scheme, with the aim at avoiding a large divergence of the LVT across the different LPs. This should favor reducing the volume of causality errors—in fact the LVTs can diverge at most by the selected side of the window.

Other solutions are based on the so called *throttling* technique, which is based on introducing delays in the speculative processing of the events whose "value" (in terms of their likelihood of being actually revealed as causally correct after their execution) is low. Clearly, unbounded delays would lead to processing events under pure conservative synchronization, at the price of uncontrolled performance degradation. In this class of solutions we find the Elastic Time technique [46], or other similar approaches [47, 48, 49, 50], where the speculative behavior of each LP is constrained by delaying the execution of events that are too far in simulation time—this is done in the hope to process them with increased likelihood of their causal correctness.

In all the above approaches we see how the baseline reference scheme is optimistic synchronization, so conservative synchronization plays the role of brake for speculative processing. A different approach is explored in [51], where a protocol natively embedding safe and unsafe processing of events is presented, although standing as a pure theoretical proposal with no actual implementation.

2.2.4 The role of Global Virtual Time

Speculative virtual-time synchronization is based on the idea that it is possible to go forward along the simulation trajectory without any previous assessment of causal consistency of the events processed at the LPs, restoring some previous correct state if a causality violation is detected. As explained in the previous section, in order to enable state restore at any point along simulation time, state snapshots are taken and processed events are kept stored even after their execution, so as to use them in any coasting forward phase. However, while the simulation advances, such amount of stored information can lead to significant memory footprint, an aspect that can definitely impact performance. Further, as an extreme, memory unavailability caused by address space exhaustion could lead to aborting the simulation run. Thus, it is mandatory to perform periodic *fossil-collection* operations to reclaim memory used to store obsolete information that is sure to be no longer needed for supporting any rollback [52]. As for this aspect, we recall that reverse computing—where we can simply use reverse event handlers for backward reconstruction of the LP states—tackles this problem only partially, given that, as discussed, to be really effective it needs to be complemented with some form of checkpointing (see, e.g., [42, 43]).

To cope with this problem, optimistic PDES relies on the concept of *Global Virtual Time* (GVT). It represents a kind of time-barrier separating the portion of simulation that can still be retracted from the one that can no longer be affected by causality violations—namely the committed portion. GVT is defined as the minimum timestamp of any event being processed, or still to be processed, and any in-transit message/anti-message. Although the GVT value characterizes the status of the simulation run at any WCT instant, algorithms for (periodically) computing the GVT value [53, 54, 55, 56] typically provide "lower bound" approximations

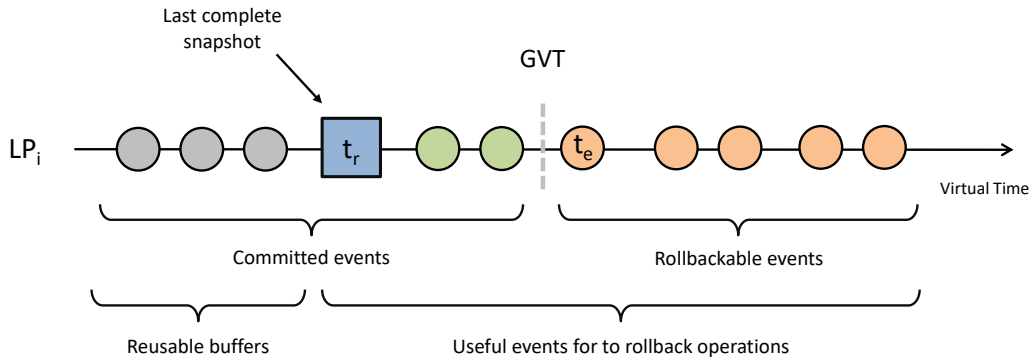

Figure 2.6. Valid memory buffers for fossil collection

of the real GVT value at the current WCT. However, these approximations are still useful to discard stored information related to the execution portion that has become irrevocable.

In some sense, the GVT value represents the *commit horizon* of the simulation run, and it is also exploited to carry out other tasks (beyond memory recovery) such as the displaying (or auditing) of intermediate simulation results that are already known to be correct—they are guaranteed to be not affected by virtual-time synchronization errors. Concerning such output production, the frequency with which the GVT is computed (and an updated value is actually determined) is relevant especially considering interactive simulations where I/O operations providing new data to end-users must be committed as soon as possible. On the other hand, the GVT algorithm has a runtime cost that needs to be considered when tuning the frequency of its execution.

Clearly, depending on the actual configuration of the state restoration support, it could be possible that some stored information related to simulation time preceding the GVT value still needs to be kept upon fossil collection. Considering for example periodic state saving, starting from the last computed GVT value, it is possible to define, for each LP, a timestamp t_r corresponding to the oldest state snapshot that could need to be reloaded by a rollback. In particular, identified for each LP a timestamp t_e associated with the oldest event to be possibly rolled back, namely the first one with timestamp greater than the GVT value, t_r corresponds to the latest taken snapshot such that $t_r \leq t_e$. Thus, at each WCT instant all the saved snapshots s of the LP state such that $t_s < t_r$ can be discarded (see Figure 2.6), together with all the events belonging to this virtual-time portion. Obviously, the fossil collection operation can be performed after a new GVT value is computed to make memory recovery prompt, and also in this case it is important to find a correct tradeoff between the frequency of GVT computation, and the advantage from memory recovery at that frequency (such as improved locality).

As a final important note, the concept of GVT has been used in optimistic PDES mostly to drive the aforementioned housekeeping tasks. The solutions provided in this thesis will exploit the GVT value as core information not only for housekeeping, but also for driving the scheduling of the core activities (such as event processing) along the threads that will run the share-everything PDES system, as we will clarify in Chapter 4.

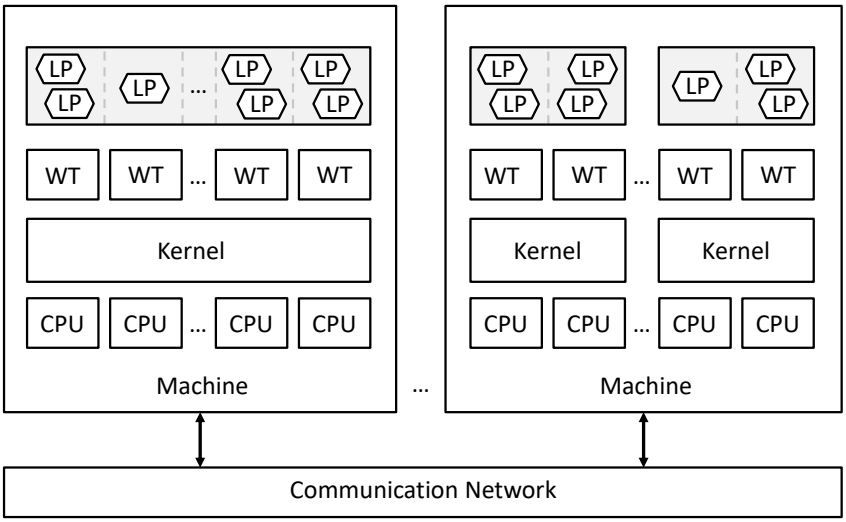

Figure 2.7. PDES Multi-threaded Architecture

2.2.5 Software architecture aspects

As the reader may have noted, PDES and its flavors (conservative and optimistic) have been presented in terms of their baseline methods. However, in real PDES systems these methods need to be supported by specific algorithmic steps, and therefore by the design and implementation of the simulator architecture driving these steps. Since this thesis is exactly focused on new algorithms and implementations for supporting the PDES methodology on multi-core machines, in this section we outline the common literature way of building the architecture of PDES systems. This description will constitute an additional important background for understanding the innovative contributions by this thesis. Given that our focus is on speculative PDES (although initially on limited speculation capabilities), our description will be essentially tailored to PDES systems adhering to the optimistic virtual-time synchronization approach.

In classical architectural implementations of these systems (see, e.g., [57, 58, 59, 60, 61]), the simulation kernel is partitioned in multiple kernel instances, each one managing a subset of LPs (so each LP is hosted by exactly one kernel instance) as if these LPs and the hosting kernel instance were a unique system in charge of simulating the evolution of a sub-model of the original simulation model—although we know that they are not really independent because of the possibility of cross-LP scheduling of events. These kernel instances have been historically devised as different single-thread user-space processes, so that they can be run on different single CPU-core machines interconnected via a network.

More recent research trends have addressed the topic of reshuffling the traditional PDES architecture, to realize multi-threaded simulation kernels [62, 63, 64, 65]. The new architectural organization has one or more kernel instances deployed on a single or more machines and each kernel instance is in charge of managing multiple processing units (multiple CPU-cores), relying on the *worker thread* (WT) paradigm (see Figure 2.7), where each thread implements a main simulation loop. The main loop can be evenly defined for each WT or can be differentiated in order

to assign to each of the WTs a specific task (e.g., message passing, event processing or housekeeping tasks) [57, 66, 67, 68].

Nevertheless, in these solutions, the notion of binding between a thread—say a WT—and a set of LPs is still in place. In particular, a WT does not directly share the management of the LPs with the other WTs running the PDES application on the multi-core machine. The presence of shared-memory in the hardware, and address space sharing across WTs, has been in fact exploited for optimizing cross-thread message passing (reflecting the message-passing operations triggered by the dynamic creation of new events when some LP processes an event) as well as message-passing operations related to housekeeping algorithms such as GVT computation [54].

Overall, a WT is the only responsible of managing the event queues of its associated LPs—including the output queues storing the events dynamically produced by the LPs, which are useful if we need to produce anti-messages, namely negative copies of these sent messages. It also manages all the per-LP data structures needed to guarantee state restoration (such as state snapshots). At the same time, the WT schedules the events for execution as if it was a standalone simulator, so it always selects the event with the minimum timestamp among those not yet processed and destined to its bound LPs—this is also known to as Lowest Timestamp First scheduling [69, 70]. Anyhow, this does not save from causality errors since LPs bound to other WTs can produce events destined to local LPs with timestamps in the past of their LVTs.

Within this panorama, a few works started devising kernel level algorithms and protocols specifically suited for shared-memory, thus not only based on some shared-memory implementations of the underlying message-passing operations. Along this path we find approaches for shared-memory computation of GVT [54, 55], as well as approaches for shared-memory suited load-balancing [71, 72, 73].

The last concept, namely load-balancing, introduces an additional core point of interest for us. More in detail, even if the suite of all algorithms and related implementations for putting in place the speculative PDES paradigms were largely optimized—we can think of messaging, state restoration as well as GVT—, a core impairment might lead speculative PDES to (partially) fail in its objectives of high performance and scalability.

This impairment is essentially related to the existence of the aforementioned binding between WTs and LPs, which is the motivation for the need of load-balancing policies aimed at dynamically changing such binding for keeping the advancement of LVTs balanced (along model execution) across the overall set of LPs. If a "perfectly" balanced advancement would not be guaranteed, then causality errors might become the predominant factor negatively affecting performance (even under extreme efficiency while managing each single rollback operation). However, under load-balancing, an LP bound to some WT cannot be arbitrarily (at any point in time) CPU-scheduled for executing its events by other WTs, even though in principles the LP (and its state) as well as its support data structures—such as event queues and state restoration information—are at any time accessible to all the WTs running the PDES application on top of the multi-core machine. This imposes a kind of limitation on how the PDES system architecture uses the computing power in the underlying multi-core machine—and delivers it towards the simulation model

so to achieve the "perfect" balance of the LPs' advancements—a drawback which is directly tackled by the share-everything paradigm we explore in this thesis.

Overall, in such literature PDES systems, WTs do not share all the core data representing the state of the model execution at arbitrary points in time. Such a sharing, with all its benefits, is explored in this thesis via optimized non-blocking shared-data management approaches suited for PDES. The concept of non-blocking management of shared data structures across concurrent threads, known in literature as *non-blocking synchronization*, and the related main literature result, are recalled in the next chapter so as to finalize the background information required for the understanding of the PDES-suited technical solutions provided in this thesis.

Chapter 3

Non-blocking synchronization

To ensure correctness (data consistency) while executing a multi-threaded application where threads access share-data structures, some form of thread synchronization (coordination) along wall-clock time must be adopted. This aspect is related to the notion of *safety* of a (concurrent) application, a property implying that *nothing bad happens* along the execution of any thread. In other words, a program satisfies the safety property if in all its possible executions (*histories*) its result is guaranteed to be correct. In fact, we can consider a shared resource (a shared data-structure) as an object characterized by a set of possible values it can assume, and a set of operations to manipulate such object making it transit from an admitted value to another one. These operations are characterized by pre-conditions and post-conditions that have to be respected by the object implementation. While in a sequential application it is quite easy to proof that such conditions actually hold, for a concurrent application, a way to detect correctness has to be provided.

This is done by checking if the performed operations (identified by an *invocation* and a *response*) during a concurrent history can be reordered in order to build an equivalent sequential one. In particular, a concurrent history (characterized by the interleaving of invocations and responses along different threads) is equivalent to a sequential one if it is possible to reorder it, obtaining the sequential one, while keeping all the histories associated with each single thread (thread sub-history) equivalent. Thus, safety can be defined by the *linearizability* property [74]: a concurrent algorithm is said to be linearizable, if, for any possible history it is possible to generate an equivalent sequential history such that the semantic of the objects on which the algorithm insists is respected. In other words, an algorithm is considered linearizable if, for each invocation of an operation, it appears as if the operation takes effect instantaneously (atomically) between its invocation and its response, while the total ordering of these operations corresponds to a valid sequential one.

To guarantee correctness, the most diffused technique is *mutual exclusion*, which relies on the *locking* primitive to enable one thread instance in a parallel program to execute a *critical section*. In other words, a thread is enabled to perform read/-modify operations on share-data only if no-one else is performing the same action at the same time. Such solution looks as a paradigm suited for low-to-medium scale systems/application-deploys, but does not look to exhibit the adequate level

Table 3.1. Progress conditions based classification

	Independent Non-Blocking	**Dependent Non-Blocking**	**Dependent Blocking**
Every method makes progress (maximal)	Wait free	Obstruction Free	Starvation Free
Some method makes progress (minimal)	Lock free	Clash free	Deadlock free

of scalability for efficient exploitation of larger multi-core machines, likely causing a significant deterioration of performance in scenarios with high concurrency and non-partitioned data accesses—leading to higher likelihood of wait phases for lock acquisition by the threads.

In order to overcome such problems, *non-blocking synchronization* has been devised. This technique avoids the execution of classical critical sections by relying on *fine-grain synchronization*, at the level of a single machine instruction. This approach, which has been devised in the past decades [74, 75, 76, 77], aims to reduce the critical section length to the single processor instruction. In particular, it allows asynchronous and concurrent accesses to data objects, yet guaranteeing consistency in the updates, through the exploitation of atomic instructions offered by the underlying hardware.

The one of non-blocking algorithms can be considered as a class into the classification based on the *progress condition* provided in [78], which is symmetrically opposed to the blocking class. The blocking class contains all the algorithms where a delay of a thread can prevent another one to make progress in its execution. Within this class we find of course all the lock-based algorithms, since if a thread executing the critical section is delayed or blocked, all the threads waiting to acquire the busy lock are delayed. However, all the solutions where a thread can be blocked waiting for another one (e.g., waiting for the result produced by another thread) fall into this class. On the opposite side, an algorithm is considered non-blocking if threads are enabled to make progress independently of the behavior of the other threads. Further, depending on the number of threads enabled to make progress, namely a few or all of them, it is possible to distinguish *minimal* and *maximal* progress conditions.

Depending on whether minimal or maximal progress conditions are respected, blocking synchronization algorithms can be grouped in:

- *Deadlock free* – given a set of concurrent threads all trying to access the same critical section, *at least one* eventually succeeds. This is the weakest property that can be requested by an algorithm and guarantees that the whole system makes progress, but does not guarantee progress of each single thread.

- *Starvation free* – given a set of concurrent threads all trying to access the same critical section, *all* of them eventually succeed.

Such properties make sense only under the scenario where a thread that is executing in a critical section eventually leaves it releasing the corresponding lock. This means that, in order to guarantee such properties, a fair scheduler is requested, namely a scheduler that does not suspend a thread running within a critical section. As for this aspect, a progress condition is considered *dependent* if it requires the support of a fair scheduler to guarantee minimal progress, otherwise it is considered *independent*.

As for non-blocking algorithms, depending on the progress guarantee they provide, three categories are available:

- *Lock free* [77] – given a set of concurrent threads all contending on the same set of data objects, *at least one* progresses in finite execution steps. This property ensures the absence of deadlocks, but does not guarantee the absence of *starvation*.

- *Wait free* [77] – given a set of concurrent threads all contending on the same set of data objects, they *all* progress in finite execution steps. This is the strongest property for a non-blocking algorithm.

- *Obstruction free* [79] – given a set of concurrent threads all contending on the same set of data objects, one progresses in finite execution steps only if it executes solo for long enough. This property ensures the absence of deadlock, but *livelocks* may occur if a set of threads keep preempting or aborting each other's atomic operations.

In Table 3.1 we show the classification based on progress conditions. Looking at the horizontal organization, the first line contains maximal progress conditions, while the second one contains minimal progress conditions. Instead, the first column contains non-blocking independent progress conditions, the second one contains the non-blocking dependent conditions, while the last one contains the blocking dependent progress conditions. Here, a non-blocking dependent property that guarantees minimal progress is reported, named the *Clash free*. Such condition is considered the Einsteinium of the periodical table, since it not exists in any natural algorithm.

An efficient implementation of coordination tasks, done by using appropriate algorithmic techniques based on lock-free/wait-free solutions, can provide a significant scalability improvement. This has a direct twofold effect on the execution of applications relying on these synchronization tasks: programs run faster, so the results can be obtained in shorter time, by increasing the number of processing units while maintaining fixed the problem size (this property is often referred to as *strong scalability*). At the same time, problems of *larger size* can be made tractable, by increasing at the same time the problem size and the number of processing units (a property often referred to as *weak scalability*).

However, it is not straightforward that a non-blocking implementation provides the expected increase of performance compared to its blocking counterpart, even at relatively high thread (CPU-core) counts. One motivation is that, while executing

non-blocking synchronization algorithms, in particular lock-free ones, usually there is a try-until-succeed cyclic pattern that can be summarized in three main steps: read the current state of the object; perform update locally; try to commit such update verifying that the state is still compliant with the initial one. Thus, if the state is changed during such steps, the performed work has to be squashed restarting from the beginning. In this scenario, if a large amount of threads works on the same (portion of an) object, there is a high probability to conflict (and fail), nullifying the advantages potentially provided by the non-blocking approach. Moreover, the update phase of the object attempted at the third step listed above needs to be carried out via complex atomic machine instructions that impose costs related to firmware level protocols (such as protocols for bringing cache lines into exclusive states for the atomic manipulation by the machine instructions - like, e.g., the MESI protocol [80]). These aspects need to be taken into account when providing solutions falling in the non-blocking class. In the next section we discuss in detail the common memory consistency model provided by off-the-shelf machines, which clearly plays a relevant role in the way atomic memory manipulations executed by machine instructions are actually carried out in multi-core machines. This is important to understand the type of instructions, and their effects, we will exploit for building our share-everything PDES oriented algorithms and the corresponding implementations of multi-threaded PDES platforms.

3.1 Memory consistency model

While devising concurrent algorithms to be deployed on a nowadays machine, it is important to fully understand the behavior of the underlying hardware, in particular referring to the view that each thread has about the memory shared with other threads working on the same address space. Indeed, multi-processor/multi-core shared memory systems offer *memory consistency models* [81] as kind of "contracts" among software developers and hardware manufacturers, discriminating what software can expect to be guaranteed by the underlying hardware. A variety of consistency models exist, each one often presented as a set of rules.

The simplest memory consistency model is *sequential consistency*. In this model the results of any execution are the same as if the operations of all the processing units were executed in some sequential order, and the operations of each individual processing unit appear in this sequence in the order specified by the program it is running [82]. This model ensures that all read and write instructions executed by any processing unit (a CPU-core) are observed in the same order by all the processing units in the system.

In this thesis, we assume a weaker consistency model, namely *Total Store Order* (TSO) [81], which is used by most off-the-shelf platforms, such as SPARC and x86, thus making our solutions of general applicability. With TSO, CPU-cores usually use *store buffers* to hold the stores committed by the overlying pipeline until the underlying memory hierarchy is able to process them (Figure 3.1). In particular, a store leaves the buffer whenever the cache line to be written is in a coherence state such that the update can be safely performed. TSO allows what is called *store*

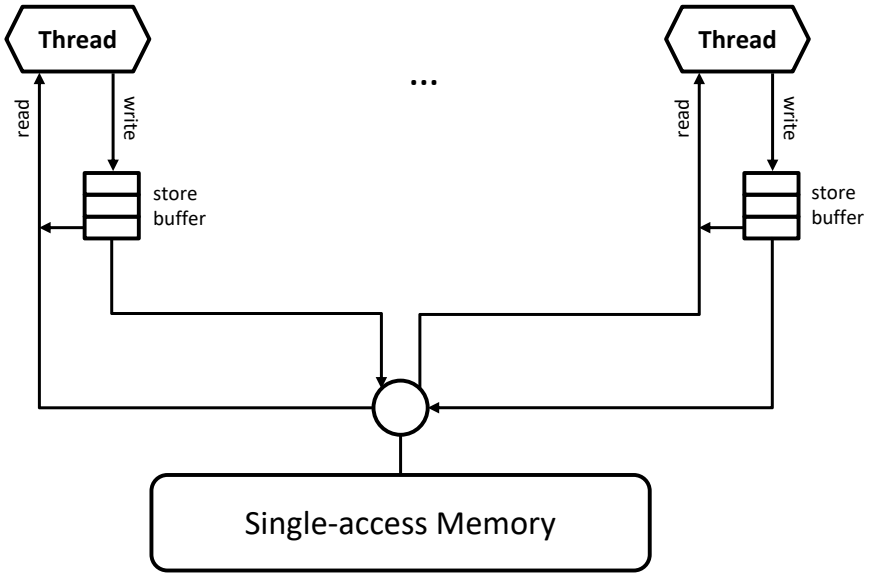

Figure 3.1. Total Store Order consistency model logical representation

bypass: even if a CPU-core outputs a write before a read, their order on memory (as seen by other CPU-cores) can be reversed.

While TSO produces no damage in many applications—rather, it can provide a significant speedup due to a reduced latency on the memory hierarchy—(non-blocking) synchronization based on atomic memory operations on shared data must explicitly cope with this scenario. In fact, store bypasses can affect the correctness of synchronization algorithms for concurrent threads only relying on individual read/write operations (just like [83]). On the other hand, TSO-based architectures offer particular instructions in their ISA, referred to as *memory fences*, which enable recovering sequential consistency by explicitly flushing store buffers before executing any other memory operation, thus allowing to preserve the ordering across subsequent read/write operations.

However, for scenarios where synchronization among threads requires to atomically perform pairs of memory operations (or more)—as an example a read and a write with an updated value on the same memory location—memory fences do not suffice. To cope with this issue, TSO-based architectures offer *Read-Modify-Write* (RMW) instructions, whose execution directly interacts with cache controllers so as to ensure that cache lines keeping *synchronization variables* are held in an exclusive state until a couple of read/write operations are executed atomically [81]. This means that no other cache can keep the same line in read mode until the couple of operations completes.

In our work we widely rely on these instructions in order to guarantee correctness while devising non-blocking algorithms suited for share-everything PDES. Specifically, some of the classical RMW instructions, supported by off-the-shelf processors, and employed in the algorithmic proposals in this thesis are: `Compare&Swap`, `Fetch&Add` and `Fetch&Or`.

`Compare&Swap (CAS)` is one of the most important instructions employed in concurrent programming. It receives in input a memory address (`addr`) and a couple of values (`old` and `new`) in order to compare the content stored in `addr` with the value `old` (the expected one to be found) and, only if they are the same, updates the content of that memory location with the value `new`. The atomicity of this RMW instruction guarantees that the new value is computed starting from up-to-date information; if the value has been updated by another thread in the meantime, the write would fail. Regarding the result returned by the instruction, it indicates whether the update has been performed. In particular two different API are exposed corresponding to this operation, one returning a boolean value and another one returning the value found in this memory location—thus it is possible to check the outcome of the operation by also comparing the result value with `old`.

In many synchronization algorithms, the `CAS` instruction is used to perform an update only if nothing has changed in a given memory location. Unfortunately, the thread executing the `CAS` instruction has not information about what is happened from the moment in which the `old` value has been read. Thus, in the meanwhile the `addr` memory location could have been updated multiple times returning at the end to its original value. This is also known to as the ABA problem. While in some cases ABA is not a real problem—thinking about arithmetic operations—in many cases this can affect the correctness of the execution. Depending on the context in which the `CAS` is employed, different solutions are available, like e.g., deferred memory reclamation—while working on memory addresses—or tags.

While `CAS` is usually employed in retry cycles, since it is not guaranteed to succeed, other instructions are devised to perform their update with no failure, which we describe below. The simplest one is the `AtomicExchange` which atomically reads the content of a memory location and updates its value.

The `Fetch&Add` instruction receives in input a memory address `addr` and a value `val`. This instruction atomically increments the value stored at the memory address `addr` by the specified value `val`—namely retrieves the memory content, performs the arithmetic operation and finally replaces the memory content with the obtained value—returning the value stored inside `addr` before its execution. There exist variants of this instruction aimed at performing different arithmetic operations or at returning the value stored after its execution (`Add&Fetch`).

The `Fetch&Or` instruction, similarly to the previous one, receives in input a memory address `addr` and a value `val` in order to gather the memory content and atomically apply the logical update. In particular this instruction performs a bitwise-or operation between the content of the memory location `addr` and the value `val`. Also, in this case variants are defined to perform different logic operations, like the bitwise-and one (`Fetch&And`), or to shift the returned value before the instruction execution (`Or&Fetch`).

3.1.1 Harris' non-blocking linked-list

In the literature, several non-blocking algorithms have been presented in order to manage various types of well-known data structures. Among them we can mention hash tables [84, 85], trees [86, 87, 88], priority queues [89, 90, 91, 92, 93] and software-implemented registers [83, 94, 95].

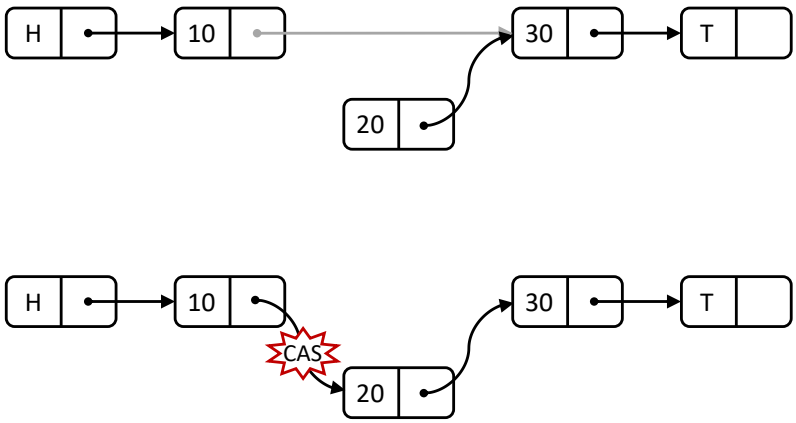

Figure 3.2. Harris' list insert operation

In this section we provide the reader with details on the non-blocking linked-list presented in [96], which consists of an ordered chain of nodes supporting linearizable insertions and deletions. This data structure is used in this thesis as the very baseline for building more complex ones—and more complex non-blocking algorithms—for managing shared data in the share-everything PDES context. Thereby, its description will help the reader better understanding the contribution by this thesis.

Each node stores a pointer to the next one in the list, a key value (e.g., a timestamp) on which a total order can be defined, and possibly a pointer to a generic element. Two special nodes are defined, *head* and *tail*, marking the begin and the end of the list in such a way that all the other nodes are placed between them. Three operations are defined: *insert*, which adds an element, with its key, in the proper position in the list; *delete*, which removes the element with the specified key from the list; *find*, which check if the list contains a given key. All these operations are supported by a *search* procedure which returns the pointers to a couple of adjacent nodes—right and left—such that, the right one has a key greater than or equal to the searched one, while the left one has a key strictly smaller.

With this organization inserting a node with a key k is a quite simple operation. Indeed, it is enough to perform a search operation to retrieve the nodes that should surround the new one and, if the right one has a key different by k (the Harris' linked-list is designed to contains only one instance of a given key), tries to insert it by performing a `CAS` operation on the next field of the left node, as shown in Figure 3.2. In this case, atomicity of the `CAS` instruction ensures that, if it has been succeeding, the left and right nodes have remained adjacent. Contrarily, if the `CAS` instruction fails, something is changed in the target portion of the list, thus the insert operation has to be restarted, just according to the aforementioned abort-retry paradigm.

Unfortunately, a delete operation cannot be carried out in a similar way. In fact, removing a node—the right one returned by the *search* procedure—by updating the next field of the left node in order to point to the next one, with respect the right

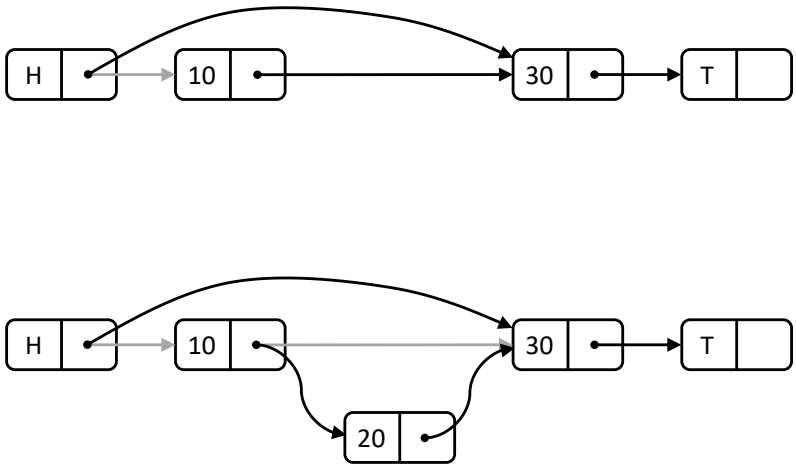

Figure 3.3. A possible wrong implementation of the Harris' list delete operation

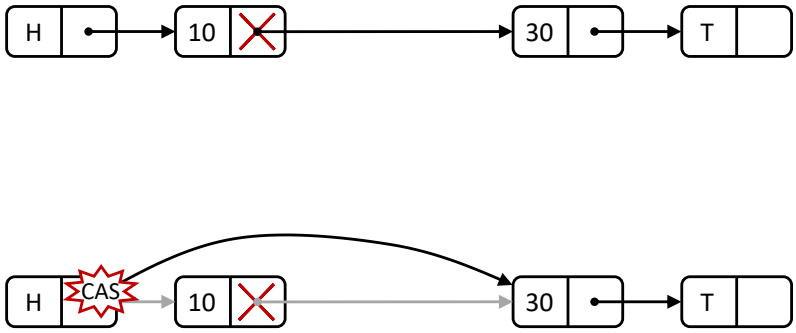

Figure 3.4. Harris' list delete operation

node, we cannot guarantee that no new nodes were inserted between the left and the right ones. In this case the new nodes would be "lost", as shown in Figure 3.3.

In order to prevent such problem, the *delete* procedure is carried out in two phases, each one performed via `CAS` (Figure 3.4). In the first phase, the node to be removed is marked as logically deleted. This operation is carried out by performing a `CAS` instruction on its next field in order to set the last bit to 1—note that using one bit as a flag in the next filed requires that nodes are aligned to a multiple of 16 bits. After this operation the node can be still traversed by concurrent threads, but it is no longer considered as a valid one. Moreover, since its next field is explicitly marked, a concurrent operation working on it will notice it and/or will fail—insertions are not enabled to introduce nodes after a deleted one—avoiding the scenario shown in Figure 3.3. At this point, since a logically deleted node can be considered in a final state—nothing can happen on its next field—the delete operation simply tries to shift the next pointer of the left node to the next one, in order to physically remove it.

As mentioned, this design relies on a *search* procedure that has to return two adjacent and still not deleted nodes. Thus, the search procedure is in charge of removing logically deleted nodes between the ones to be returned: this means that, the physical removal of the logically deleted node can be carried out by any thread performing a search procedure.

As we will see in Chapter 5, we have adopted this design modifying it in order to improve its expressiveness in terms of states possibly assumed by the nodes, and improving its efficiency by including conflict-resilient capabilities. This led us to achieve a baseline algorithmic approach for share-everything PDES on multi-core machines. On the other hand, in the solutions provided in the subsequent chapters we definitely slide towards data structures with intrinsically higher semantics, exactly tailored to the needs of the PDES scenario.

Chapter 4

Share-everything PDES: the basics

By the discussion in Chapter 2, it should be clear that an uneven virtual-time distribution of events across LPs can be experienced depending on the specific dynamics of the executed model, which—depending in turn on the way such workload is actually processed by the WTs within the PDES environment—may produce, at a given wall-clock-time instant, a misalignment across the LPs in simulation time. A similar effect can also appear when we have events with (very) different computational requirements, since more demanding ones may lead the target LPs to remain back along simulation time, with respect to LPs targeted by less demanding events, which are promptly processed one after the other.

Such an effect, actually well-known in the literature, is globally recognized as a negative phenomenon in speculative PDES. In fact, the probability according to which an event e speculatively executed on LP_i is revealed to be actually useful (causally consistent) is inversely proportional to the event distance from the commit horizon, namely the current GVT value at the time of processing the event [49]. Also, even if this statement does not represent a rule, intuitively, pushing LP_i far from the commit horizon may increase the amount of events that will be eventually executed by other LPs, having timestamps in the simulation time interval between the GVT value and LVT_i. This, in turn, may increase the likelihood to generate an instance of event e' destined to LP_i with timestamp $t_{e'}$, such that $t_{e'} < LVT_i$—this is a straggler event requiring the rollback of LP_i. A problem that can be amplified considering that the rollback of LP_i could cause in turn the rollback of other LPs. Moreover, performing a rollback, not only vanishes the amount of carried-out speculatively work, but incurs in additional costs to restore a causal-consistent state. Under such considerations, an unbalanced advancement of the simulation produces an increased amount of rollbacks, reducing in turn the percentage of time spent in useful execution.

Along its life, the research on PDES has been integrated with techniques and solutions aimed at continuously improving its capabilities of exploiting the underlying hardware with the goal to enhance performance and scalability. This includes approaches to cope with the aforementioned problem of divergence between the LVTs

of the different LPs along model-execution. However, as hinted, most of the literature work has been focused on the classical PDES architectural organization, where a set of LPs is bound to WTs/CPU-cores. In such a scenario, in order to mitigate such problem, the binding between LPs and WTs is periodically re-evaluated, relying, as noted in Chapter 2, on load-balancing mechanisms: in case an unbalance is detected, the WT hosting an "excessive workload" passes one or more LPs to other WTs, depending on the statistics collected in the previous observation period, by migrating such LPs together with the associated data (e.g., events to be processed) across processing units.

In such organization, the LP is considered as the *work unit* to reason on in order to optimize the workload distribution, and the advancement of the overall simulation run. The events bound to the LP—and their clustering along virtual time, as well as their average computational requirements—are used only as an index of the amount of to-be-done work—the *weight*—associated with the LP in order to determine how to manage it within the load-balancing scheme. Overall, an individual event does not play a role in the planning of resources (CPU-cores), but rather it has a role only when grouped together with other events destined to the same LP, exactly for the definition of the weight of the LP.

Load balancing—and its baseline concept of considering the LP as the work unit—can be considered as a good means for medium to long term planning of resource usage, which clearly fits simulations characterized by a quite regular evolution of the simulation model over time, with infrequent scenarios of spikes of the (dynamically generated) workload associated with the LPs. Overall, load balancing appears not to be fully suited for punctual fluctuations in the "weights" of the different LPs, namely in the density and computational requirements of the events destined to the LPs along simulation time. We can think of this drawback as a kind of "imperfection" in how the computing power is delivered towards the model to be executed.

More precisely, skews may arise in the advancement of the LPs along virtual time—although we might still consider a simulation execution as balanced along a non-minimal observation window, e.g., the one used by the load-balancing algorithm. In these scenarios classical PDES-oriented load-balancing approaches, based on medium/long-term binding between LPs and threads, have scarce capability to react to the sudden unbalances that may materialize, which can lead to an increase of the likelihood of wasted computation in case of speculative processing—or blocking event processing waiting for the events to become safe in the conservative one.

An example of such sudden imbalance is shown in Figure 4.1. In this scenario, although a load balancer might have bound LP_i and LP_j to WT_x and LP_k and LP_w to WT_y—because the (estimated) average weight of these two couples of LPs might be similar in the current load-balancing period—for a subinterval of the load-balancing period, we may experience anyhow an imbalance since in this subinterval LP_i and LP_j will likely remain back in simulation time while the other two LPs will run ahead. This may generate the scenario where a newly produced event by LP_j at some point in time will become a straggler, invalidating work done by WT_y.

Here comes the share-everything PDES paradigm for multi-core machines. It stands as an alternative way of devising the architecture of a PDES system, which is aimed at the optimized (fruitful) delivery of the computing power towards the model

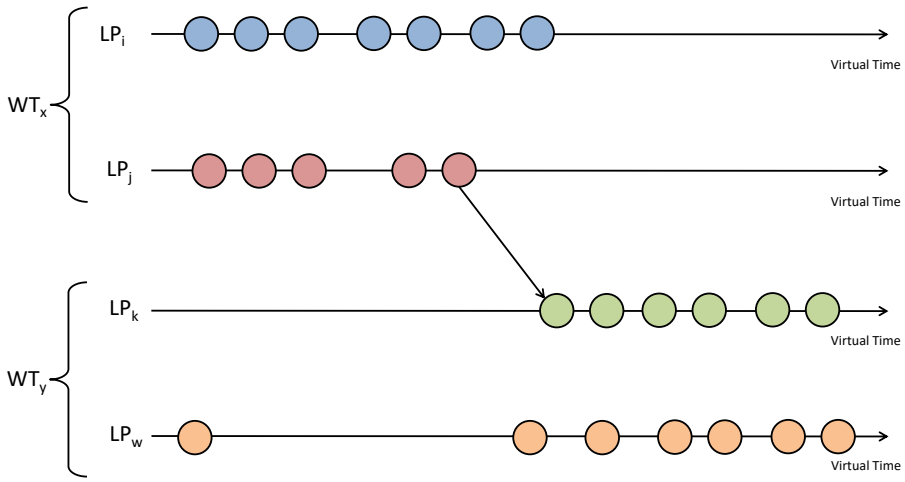

Figure 4.1. Example of sudden imbalance along a subinterval of the load-balancing period

to be executed "at any time instant". This optimization is based on the idea that all the threads (so all the CPU-cores) running the PDES application can fully share *finer grain* work units, namely individual events possibly bound to different LPs. Under this perspective, the LPs become simple "containers" of work, from which threads pick such fine grain work units for processing. The LPs, and their weights, are no longer the actual workload measure to be used for driving the planning of resources' usage.

At high level, the core concept that stands behind share-everything PDES is the one of always delivering the computing power to the not-yet-processed events having higher priority, namely the ones with lowest timestamps. In other words, with this kind of approach the main aim is the one of avoiding that a thread picks for processing an event e that is more far than another not-yet-processed event e' from the current commit horizon. Also, such an avoidance should not be an average-behavior property—meaning that, in the end, some thread should not take care of pushing some LP significantly far ahead in simulation time from the others along some window of the model-execution—but rather the core property to be systematically guaranteed at any time instant along the simulation run.

The main ingredient for reaching this target is the absence, in share-everything PDES, of the limitation on what LP can be dispatched for execution by any thread at any time instant—a limitation that is instead typical of common PDES systems based on (temporary) binding of LPs and WTs. Here, in fact, a thread can CPU-dispatch at any time only the events destined to its currently bound LPs.

Clearly, the share-everything PDES architectural paradigms has the potential to cope with the hard workloads mentioned in Chapter 1, where we may have (sudden) skews in the distribution of the events across LPs along virtual time (see Figure 4.1). These skews possibly create relatively short bursts of events to be processed at a subset of the LPs, while other LPs have no (or few) events to be processed along that same virtual-time window—a phenomenon that remains unchallenged by classical medium/long term resource usage planners like literature load balancers for PDES systems. The share-everything PDES paradigm considers events as fully-shared

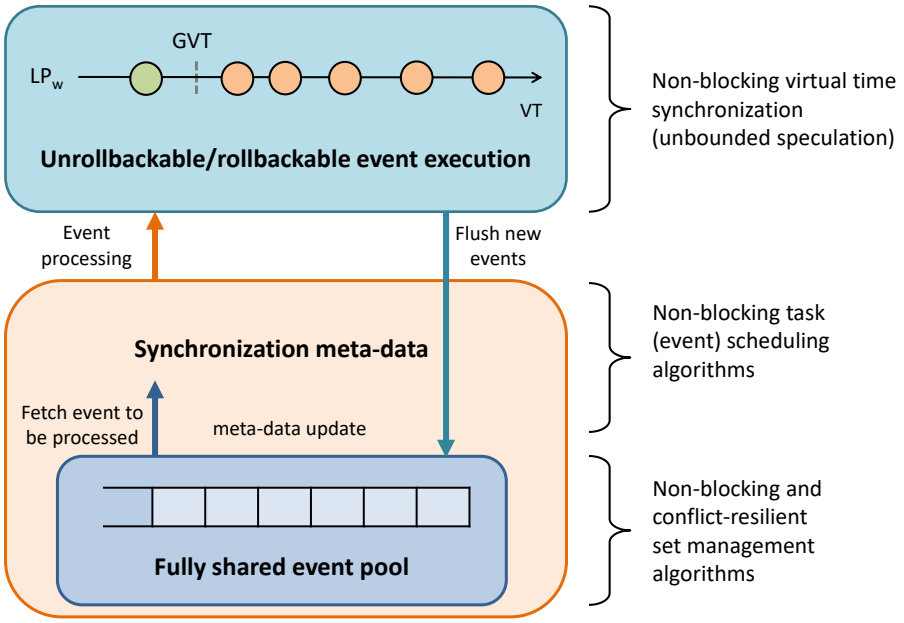

Figure 4.2. Building blocks and related properties for share-everything PDES

workload units, thus being able to concentrate the computing power on any burst of events that materializes among subsets of LPs. Indeed, all the WTs can take care of processing any event in these bursts—hence the events in the bursts that are closer to the current GVT—thus contributing to promptly advance the commit horizon.

The last expressed concept is relevant for making a linkage with a few approaches we already mentioned in Chapter 2, which were based on "throttling" schemes (see, e.g., [21, 50]) just to avoid that some LP, whose events are processed speculatively, will run much ahead along simulation time. In share-everything PDES we slide towards the opposite approach where we do not try to impose waits to LPs that are far ahead from others in simulation time—namely, the ones that are far ahead from the current GVT. Rather, we directly concentrate the computing power on the task of pushing ahead the actual GVT. This still leads to avoid a (big) distance between the LVTs of the LPs and the GVT, achieving the same target of reducing the incidence of causality violations in speculative PDES via an orthogonal means. Indirectly, this approach may also lead advantages for other aspects, such as when we have interactive simulations and we need to show to the end-user the outcome of the committed portion as soon as possible or at regular wall-clock-time intervals.

Such a new share-everything PDES architectural approach is simple in principle. In the essence, it can be envisioned as one based on the concept of a fully-shared event pool containing the events destined to whichever simulation object, from which all the threads can extract the higher priority events (those with timestamps closer to the current commit horizon) for processing, and into which the same threads put newly generated events. On the other hand, as mentioned in Chapter 1, fully sharing the workload of events across all threads poses hard problems in terms of managing the computing power in a truly scalable and efficient way. In fact, while

in classical PDES architectures there is a unique synchronization dimension, the one of virtual time—here threads coordinate themselves by means of the message-passing abstraction—, share-everything PDES requires synchronization along wall-clock time too, managing concurrent accesses by threads to shared resources, like, e.g., event pool and LP states.

As hinted, during the last decades virtual-time synchronization [19, 97] and wall-clock-time synchronization [77] have been extensively investigated topics. However, there is a lack of literature results—or very few works exist that we will discuss in Chapter 8, which show anyhow differences with respect to what we present in this thesis—on mixing such approaches in an effective fashion exactly for the purpose of hosting (speculative) PDES applications on multi-core machines. Optimistic virtual-time synchronization has been widely studied by relying on the aforementioned LP-centric data partitioning scheme, not according to the idea of shifting towards an event-centric scheme to be supported, in its turn, by advanced wall-clock-time thread synchronization mechanisms, as instead we do in this thesis.

Overall, in order to fully exploit the resources offered by the underlying multi-core hardware, an ideal general-purpose share-everything PDES platform should be designed and implemented via:

- the definition of non-blocking and conflict-resilient shared event pool management algorithms hopefully providing constant-time complexity both for insertion and extraction operations, in face of generic distributions of the events' volume and timestamps—both these features are in fact application specific. We also note that extreme efficiency in the management of concurrent accesses to the shared pool is also a means for coping with another relevant workload feature we mentioned, namely the one of fine-grain events, for which threads may frequently slide towards executing housekeeping operations, rather than persisting into the simulation model code for long CPU bursts;

- the definition of non-blocking algorithms for dispatching events to be processed across threads in such a way that threads never collide on a same LP state—but the computing power is still concentrated as much as possible on the task of advancing the actual GVT value—and causal consistency/safety is detected on-the-fly leading to the possibility to optimize the way events are actually processed (e.g., with or without the need for state restoration support);

- the capabilities to process available events by relying on transparent support for optimistic execution—again with the core focus of delivering anyhow the computing power to the task of GVT advancement.

The relation between these desirable properties and the high level description of the architectural blocks/layers of the share-everything PDES system we envision is illustrated in Figure 4.2.

In the next chapters of this thesis we incrementally address the above points, thus incrementally providing algorithms and implementations guaranteeing such desirable properties to be exploited synergistically, or even as standalone properties, of share-everything PDES on multi-core machines.

Chapter 5

Non-blocking event pool management

At the bottom-most layer of the share-everything PDES architectural organization we find an event pool that is globally shared by all the WTs running within the PDES environment. This event pool should support highly concurrent non-blocking event extractions/insertions. In fact, failing in this type of support would unavoidably impair the overall architectural design. Just to mention again a few hard to cope workloads, if the granularity of the events being processed at the LPs is (very) fine—a thing that may happen depending on the model we are simulating and on its software implementation—there is a higher likelihood that frequently, along wall-clock time, WTs will slide towards operations that require event-pool management. Hence, they will likely produce significant volumes of actually concurrent accesses to the event pool.

As we noted in Chapter 3, some literature studies have already pointed to the relevance of supporting non-blocking event pool (e.g., priority queue) algorithms based on data structures ranging from lists to calendar queues [13, 91, 98]. However, all the proposals follow the baseline non-blocking paradigm were concurrent operations that conflict on a same portion of the data structure—a thing that we discover by the failure of some RMW instruction—are such that the "wound" one is simply aborted and then retried from scratch.

In this chapter we raise the bar by tackling an endemic problem related to the usage of non-blocking event pools based on such abort/retry paradigm when employed in share-everything PDES platforms and hard to cope workloads need to be faced, such as fine grain event ones. Such phenomenon is the likelihood of high conflicts on specific portions of the event pool data structure, as we shall better clarify in a while. At the same time, our event pool, which is a non-blocking calendar queue, still offers the property of guaranteeing total ordering of the events kept within the pool based on their timestamps, just like the proposal in [91]. Hence, in our approach we do not slide towards trading off the ordering level of the elements and the runtime efficiency of the event pool in face of concurrent accesses [89]. This is quite important since total ordering allows to discriminate what event—upon an extraction from the pool—really has the highest priority with respect to others, and

therefore is more suited for being processed with the aim at actually concentrating the computing power on the task of advancing the GVT value.

Overall, we present the design and the implementation of an innovative version of the non-blocking calendar queue that has the property of being conflict resilient[1]. Conflict resilience is achieved in relation to concurrent extractions, which are highly likely bound to the same bucket of the calendar—thus touching the same portion of the whole data structure—namely the bucket containing the event with the minimum timestamp. This bucket is somehow the "hot" one in the whole calendar, since any thread will attempt to pick its next event to process exactly from that bucket. On the other hand, newly-produced events will more likely carry timestamps associated with different buckets (i.e., buckets in the future), which automatically decreases the likelihood of conflicting on concurrent insertion operations.

As opposed to the approach in [91], which does not entail conflict resilience, the lock-free calendar queue we present in this chapter makes an extraction operation still valid (i.e., committable) in scenarios where some concurrent extraction has changed the snapshot of the currently hot bucket. Rather, the extraction will be aborted only if the hot bucket becomes (logically) empty while attempting to extract from it. This is the scenario where a new bucket becomes the hot one, meaning that the locality of the processing activities within the simulation model has moved to a subsequent simulation time interval. The complexity of this type of approach stands in how we still allow linearizability of the concurrent operations on the calendar queue, including the possible mix of dequeues and enqueues bound to the same bucket, as we shall discuss.

We also address (and investigate experimentally) how to dynamically resize the width of the buckets in the calendar, which in turn affects the length of the chain of events that are expected to fall within the hot bucket. As we will show, when allowing conflict-resilient lock-free concurrent accesses to the calendar queue, classical policies for resizing the buckets (like the one proposed in the context of sequential accesses to the classical calendar queue presented in [12]) are no longer optimal. This is because the length of the chain of events associated with the buckets affects both:

- the time complexity for the access to the bucket chain, which in our case still stands as amortized constant time;
- the actual number of concurrent extractions from the same bucket which are allowed to be still committable, even if the snapshot of that bucket-chain concurrently changes; this aspect is intrinsically new and specifically related to our conflict-resilient lock-free calendar queue proposal.

5.1 The conflict-resilient non-blocking calendar queue

We have built our lock-free pending event inspiring ourselves to the classical calendar queue [12]. More in detail, our conflict resilient non-blocking version is composed of an array of entries referred to as *physical buckets*, each of which is the head

[1]The source code of our implementation can be found at `https://github.com/HPDCS/CRCQ`

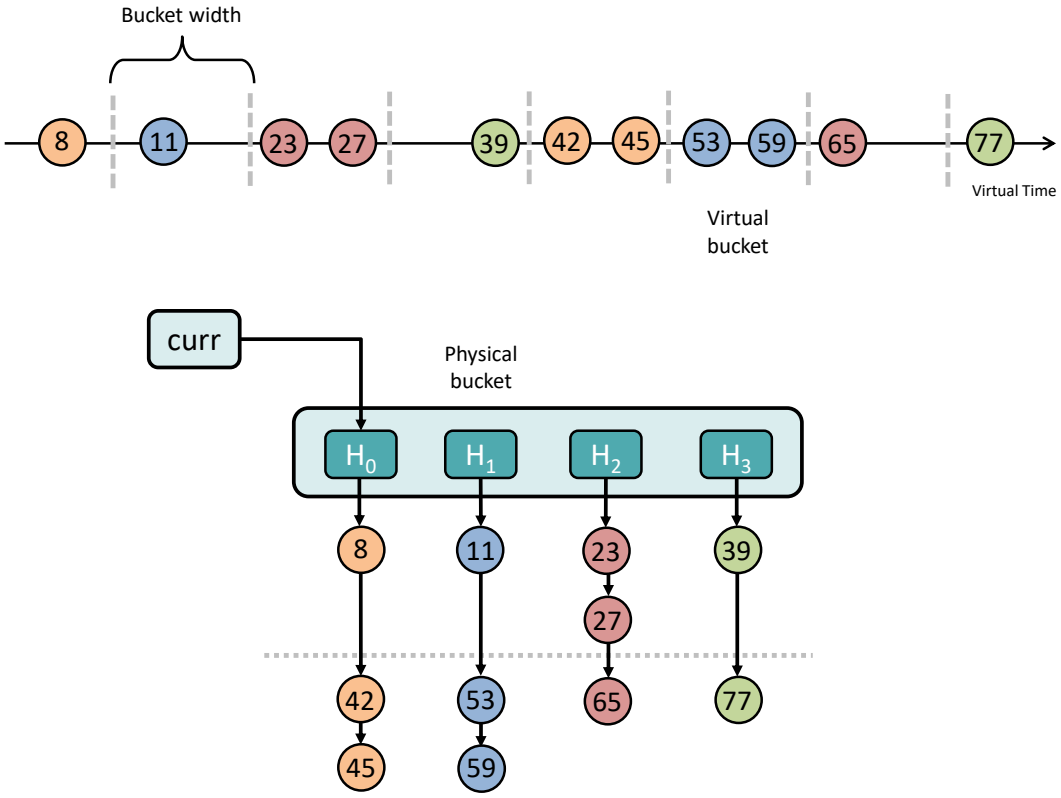

Figure 5.1. Time axis organization in the calendar queue

element of a lock-free ordered linked-list implemented according to [96], with some modifications. As seen in Section 3.1.1, in the original linked-list organization, one bit in the pointer to the next node is used to indicate whether a node has been logically deleted—it is still linked to the list, but it must be considered as invalid, say already extracted by some concurrent (or already finalized) operation. The list presented here relies on the two least-significant bits of the pointer to the next node[2] to represent four different states of a node: one bit is used in a way similar to [96], while the second one, in combination with the first, is used to indicate whether a node is involved in a resize of the calendar. Such a scenario is representative of non-blocking dynamic reorganizations of the calendar bucket size, which is enabled in our solution while still permitting regular operations (extractions/insertions) to be concurrently processed.

The time axis is divided into equal slots, called *virtual buckets*, each of which covers a time interval *bw* called bucket width. Each virtual bucket is associated with a physical bucket in a circular fashion. Thus, each physical bucket contains multiple virtual buckets covering time spans periodically repeated, as shown in Figure 5.1.

The basic organization of the data structure representing our conflict resilient non-blocking calendar queue is provided in Figure 5.2. Essentially it consists of a

[2] 4-bytes alignment ensures that such bits are always set to zero.

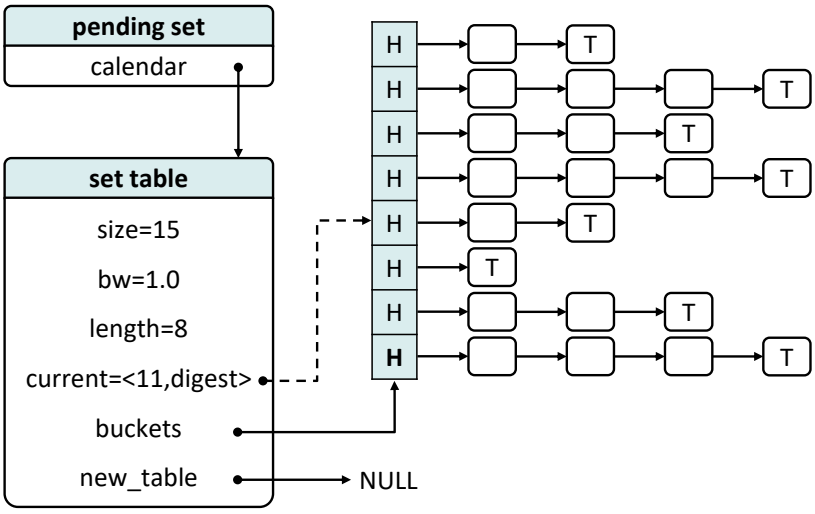

Figure 5.2. Scheme of the data structure

pointer to a table called *Set Table* which maintains the metadata required for its management. In particular, the *Set Table* contains:

i. `bw` stores the actual bucket width of the data structure;

ii. `length` is the number of buckets in the calendar;

iii. `size` keeps the number of stored events;

iv. `current` is a pair $\langle index, epoch \rangle$ such that *index* is the index of the virtual bucket containing the minimum and *epoch* is a value whose role will be explained later;

v. `buckets` is a pointer to the array that keeps the physical buckets;

vi. `new_table` is a reference initialized to `null` which is used only during the resize phase of the calendar.

To ensure consistency of concurrent accesses to our data structure, we rely on three RMW instructions, namely `Compare&Swap (CAS)`, `Fetch&Add` and `Fetch&Or`.

To avoid the effects of conflicts while concurrently extracting elements from the current bucket, we have introduced a set of capabilities oriented to reduce the need to fully restart the operations, reducing in this way the amount of wasted work—here is the conflict resilience property. Indeed, reducing the CPU-time required to actually extract a node, by avoiding fully restarting the extraction in case of conflicts, we can reduce the negative performance effects possibly caused by a larger number of conflicting threads.

To achieve this target, we have exploited the *epoch* field contained in `current`. In particular, the epoch is used to identify the instant in time at which an operation is performed, in order to reconstruct a partial temporal order among operations. A new epoch starts each time a new node is inserted in the past or in the current virtual bucket. This represents a critical instant in the state trajectory of the

calendar queue since it means that a change in the minimum, or in the recorded timestamps that can be close to the minimum, is happened.

When enqueue or dequeue operations are invoked, the ReadTable() procedure is performed to retrieve a pointer to a valid set table. The objective of this procedure is to check if there is a resize operation currently in place on the calendar queue with the goal to assist the threads involved in this operation until it is completed. When no resize operation is in place, the reference to the current valid version of the table can be returned for actually performing extractions or insertions. All the operations on the non-blocking calendar queue, including the resize, are described in detail in the next sections.

5.1.1 Enqueue operation

The pseudo-code for the Enqueue() operation is shown in Algorithm 5.1. This operation takes in input an event e happening at time T_e. Once retrieved a valid set table reference using the ReadTable() procedure, the enqueue determines the correct physical bucket associated with the timestamp T_e, computing it as $i = (\lfloor \frac{T_e}{BW} \rfloor \mod B)$ where BW is the (current) bucket width and B is the size of the calendar, namely the number of physical buckets. Since the i-th bucket is a lock-free linked-list implemented according to the indications in [96], we can use the provided search procedure to identify the right point for the insertion of the new node, with minimal changes due to the different states a node can pass through in our implementation. Executing the search procedure we retrieve a couple of nodes, called *left* and *right*, that would surround the new one. When performed, the search procedure tries to compact the nodes marked as deleted that are between the left and the right one, with the aim to return a coherent snapshot of the list. To manage events with the same timestamp, each event stores an increasing sequence number, unique with respect to the nodes with the same timestamp. In this way the couple $\langle timestamp, sequence\_number \rangle$ provides a total order among all stored nodes.

Once retrieved the surrounding nodes, the node to be inserted is updated with a reference to the *right* node and with the epoch number retrieved from the `current` field. Finally, the node is physically inserted into the list by performing a `CAS` operation on the next field of the *left* node making it point to the new one. If the `CAS` fails, the whole operation starts again from scratch. However, concurrent insertion operations, as we discussed, are typically less critical in terms of conflicts since they likely span different buckets of the calendar, or different surrounding nodes within a bucket.

To complete the update of the structure, the procedure checks if the inserted node belongs to a virtual bucket preceding or equal to the one pointed by `current`, namely if the new event stands in the past or around the minimum. In the positive case, it updates the index kept by `current`. As hinted before, the enqueue of a node in the past modifies the minimum element of the queue. Therefore, the epoch number kept by `current` is incremented by one when updating this variable, starting in this way a new epoch of the whole calendar queue. As last operation, the `size` field is incremented by one to reflect the fact that the queue has been increased, in terms of kept elements. Atomicity of all these operations is guaranteed by relying on `CAS` and `Fetch&Add` instructions.

Algorithm 5.1 lock-free Enqueue

```
E1:  procedure ENQUEUE(event e)
E2:      tmp ← new node(e)
E3:      repeat
E4:          h ← READTABLE( )
E5:          nc ← ⌊ e.ts / h.bw ⌋
E6:          bucket ← h.table[nc mod h.length]
E7:          ⟨left, right⟩ ← bucket.SEARCH(tmp.t, VAL| MOV)
E8:          tmp.next ← right
E9:          old ← h.current
E10:         tmp.epoch ← old.epoch
E11:     until CAS(&left.next, UNMARK(right), tmp)
E12:     repeat
E13:         old ← h.current
E14:         if nc > old.value then
E15:             break
E16:         end if
E17:     until CAS(&h.current, old, ⟨nc, old.epoch + 1⟩)
E18:     Fetch&Add(&h.size, 1)
E19: end procedure
```

5.1.2 Dequeue operation

The pseudo-code of the Dequeue() operation is shown in Algorithm 5.2. Similarly to the enqueue operation, a dequeue starts by issuing a call to ReadTable(), in order to retrieve a valid table reference. It then fetches the `current` field, in order to extract the *index* of the virtual bucket storing the minimum-timestamp event and the current *epoch*. The physical bucket from which to start the extraction procedure is determined as $i = (index \mod B)$, where B is the size of the calendar, namely, the number of physical buckets. Once the correct physical bucket is identified, it is scanned from the head, looking for a node valid for extraction.

However, we must ensure that the pointed node belongs to the current virtual bucket, since the list associated with a physical bucket might cover multiple virtual buckets. To this end, for each node, we check if the timestamp of the node belongs to the current virtual bucket. Moreover, to avoid the extraction of a node which has just been deleted or marked as invalid, we check whether the two last bits of the next field of the node keep some mark. Finally, according to the conflict-resilient organization of the queue, we also need to check whether the node is valid with respect to the semantic of the dequeue operation. To this end, we must verify that the node belongs to an epoch earlier than the one read from `current` at the begin of the dequeue operation, or it belongs to the same one. If this condition is true, then the node was inserted before the beginning of the dequeue operation or, alternatively, it was placed in the future (with respect to the current bucket). In this scenario the state of the node is compliant to the semantic of the dequeue operation, thus it can be safely dequeued. In fact, its insertion was correctly linearized before the

Algorithm 5.2 lock-free Dequeue

```
D1:  procedure Dequeue( )
D2:      while true do
D3:          h ← ReadTable()
D4:          oldCur ← h.current
D5:          cur ← oldCur.value
D6:          myEpoch ← oldCur.epoch
D7:          bucket ← h.table[cur++ mod h.t_size]
D8:          left ← bucket.next
D9:          min_next ← left
D10:         if ISMARKED(left,MOV) then
D11:             continue
D12:         end if
D13:         while left.local_epoch ≤ myEpoch do
D14:             right ← left.next
D15:             if ¬ISMARKED(right) then
D16:                 if left.ts < cur · h.bw then
D17:                     right ← Fetch&Or(&left.next, DEL)
D18:                     if ¬ISMARKED(right) then
D19:                         Fetch&Add(&h.size, -1)
D20:                         return left.event
D21:                     end if
D22:                 else
D23:                     if left = tail ∧ h.t_size = 1 then
D24:                         return null
D25:                     end if
D26:                     CAS(&bucket.next, min_next, left)
D27:                     CAS(&h.current, oldCur, ⟨cur, myEpoch⟩)
D28:                     break
D29:                 end if
D30:             end if
D31:             if ISMARKED(right,MOV) then
D32:                 break
D33:             end if
D34:             left ← UNMARK(right)
D35:         end while
D36:     end while
D37: end procedure
```

current (concurrent) extraction. Differently, if the above condition is not verified, then the node was inserted after the beginning of the current dequeue procedure, in particular after the initialization of a new epoch. Therefore, its insertion might have occurred after that a new minimum has been enqueued. In this latter case, the node cannot be extracted, and the operation is restarted from the beginning.

If the end of the current virtual bucket is reached without finding a valid node, either the current physical bucket is empty or the present nodes belong to a future

time slot, the procedure must continue the search for the minimum in the next bucket. Hence, `current` is atomically updated using a `CAS`. Regardless of the outcome of the `CAS` machine instruction, the procedure is restarted from the beginning. In this way, if `current` is concurrently updated, the new value becomes visible, even if it identifies a bucket in the past. Moreover, before updating the current bucket, the procedure tries to compact the virtual bucket, by cutting off (unlinking) all the traversed nodes that are found to be marked as invalid.

The search for a valid node halts when it finds a node which respects all the following properties: i) it is not marked, ii) it belongs to a correct epoch, iii) it is associated with the minimum timestamp in the current virtual bucket. Then, the dequeue operation tries to mark it as logically deleted (`DEL`), in order to finally extract and deliver it to the requesting worker thread. Differently from the classical non-blocking linked-list implementation [96], the node is marked by relying on `Fetch&Or`. The advantage coming from the reliance on this RMW instruction, rather than the classical `CAS`, is that it successfully completes even if the address of the node is concurrently updated (e.g., due to a concurrent enqueue of a new node). This is clearly favorable to concurrency since we are interested just in marking the node, regardless of a possible next node's update.

`Fetch&Or` returns as well the value stored in the target memory location, therefore it is possible to verify the outcome of the marking attempt: if the original value is unmarked, the procedure has successfully marked the node obtaining the event, otherwise the worker thread detects that a concurrent takeover by another thread took place. In this case, the dequeue operation doesn't have to be restarted. Indeed, it is enough to continue the bucket list traversal, since the epoch field can tell whether the enqueue of a node is serialized before or after the current dequeue. Finally, the `size` field is decremented to signal the extraction of an element from the queue.

5.1.3 Resizing the queue

In a way similar to the original calendar queue, $O(1)$ amortized time complexity is guaranteed by the fact that, on average, the number of elements within each bucket is balanced—namely, the number of elements inside each bucket is bounded by a constant, limiting the number of elements to be traversed during an enqueue, and there are not too much empty buckets to be scanned during a dequeue. As said before, every time an operation on the queue is performed, a ReadTable() procedure is called to retrieve a reference to a valid table. During this procedure, the number of elements per bucket is checked to determine whether it is still balanced. If this is not the case, a *resize* operation is executed. In particular, the resize is executed if `size` oversteps a certain threshold. The value of this threshold is a function of the desired number of events per bucket (denoted as $DEPB$), and the percentage of non-empty buckets.

The pseudocode of the ReadTable() operation (where the resize operation is implemented) is shown in Algorithm 5.3. When the *resize* condition is met, to announce its upcoming execution, the `new_table` field of the old set table is pointed to a new (just-allocated) set table. This somehow "freezes" the old table, preventing any new insertion/extraction operation into/from it. From now on, any

Algorithm 5.3 lock-free ReadTable

```
R1:  procedure ReadTable( )
R2:      h ← array
R3:      curSize ← h.size
R4:      if h.new = null ∧ resize is NOT required then
R5:          return h
R6:      end if
R7:      compute newSize
R8:      CAS(&h.new, null, new array(newSize))
R9:      newH ← h.new
R10:     if newH.bw≤ 0 then
R11:         begin ← random()
R12:         for j ← 0 to h.t_size−1 do
R13:             i ← (begin + j) mod curSize
R14:             retry-loop to mark i-th head as MOV
R15:             retry-loop to mark first node of i-th bucket as MOV
R16:         end for
R17:         MST ← compute bucket width
R18:         CAS(&newH.bw, −1.0, MST)
R19:     end if
```

thread executing a ReadTable() operation will be aware that a resize operation is taking place, and will start to participate.

Once the reference to the new table is published (preventing any thread from using the old one), before starting to migrate the nodes from the old to the new table, we must mark as `MOV` (exploiting the aforementioned 2-bit status information within the pointer to the next node) every entry of the bucket array, and all the first valid nodes of the associated lock-free lists. This is necessary to abort the execution of any dequeue operation. In fact, dequeue operations are restarted any time a node marked as `MOV` is found. In particular, a dequeue operation is restarted (and possibly joins the queue resize operation) if it attempts to dequeue any first valid node. In the same way, an enqueue operation is restarted if a marked node is found, meaning that a resize operation is in place.

We are therefore sure that no one will extract nodes from the queue, making stable the portion of the time axis close to the minimum. This assumption allows us to safely determine the new bucket width, and the length of the bucket array—how to determine the bucket width will be discussed in Section 5.1.4. To this end, in a way similar to [12], a certain amount of events (starting from the minimum) is inspected to compute the *mean timestamp separation*. This is the average distance between the timestamps of consecutive events along the time axis. This value is multiplied by the desired number of events per bucket, in order to compute the bucket width. The result is stored in the `bw` field of `new_table`, by relying on a single-shot `CAS` (i.e., the operation is not retried if it fails), because, if it fails, some other WT has succeeded at publishing a new bucket width.

In order to ensure lock-freedom during the resize operation (without introducing multiple copies of the same node), we rely on a mark-and-clone strategy. Before

```
R20:    for i ← 0 to h.length−1 do
R21:        while j-th bucket of h is non-empty do
R22:            i ← (begin + j) mod curSize
R23:            get first node of bucket i as right
R24:            get the next node of right as right_next
R25:            if right = tail then
R26:                break
R27:            end if
R28:            if right_next ≠ tail then
R29:                retry-loop to mark it as MOV
R30:            end if
R31:            create a copy of the right node
R32:            while true do
R33:                search for right.ts in a virtual bucket vb of newH
R34:                if found node n with same key then
R35:                    release copy
R36:                    copy ← n
R37:                    break
R38:                else if successful to insert copy as INV with a CAS then
R39:                    break
R40:                end if
R41:            end while
R42:            if CAS(&right.replica, null, copy) then
R43:                Fetch&Add(&newH.size, 1)
R44:            else if right.replica≠ copy then
R45:                try-loop to mark copy as DEL
R46:            end if
R47:            retry-loop to ensure that newH.current.value ≤ vb
R48:            retry-loop to mark right.replica as VAL
R49:            retry-loop to mark right as DEL
R50:        end while
R51:    end for
R52:    CAS(&q.array, h, newH)
R53:    return newH
R54: end procedure
```

migrating a node, a thread must ensure that the node itself and its successor are marked as MOV. If they are not, it tries to do so by relying on CAS. This is done in order to avoid any interference with concurrent threads, executing any enqueue/ dequeue operation, which did not notice that a resize operation is taking place. If the CAS succeeds (or if the nodes are already marked as MOV), the thread allocates a copy of the node, this time marked as invalid (INV). This copy is placed in the new structure, and the node (marked as INV) is ignored by any other dequeue operation until it is validated. By using a copy of the node, it is guaranteed that no node is lost, e.g., due to an unscheduled thread. When placing the node's copy into new_table, if a copy of the same node is already present, it means that some other

thread has already performed this operation. In this case, the copy is released and a reference to the one already installed into `new_table` is returned. Then, the node in the old structure is atomically updated so as to keep a reference to the found new copy. By traversing this reference, the node's copy in `new_table` is transitioned to the valid state `VAL`, and the original node is marked as logically deleted (`DEL`). It will be later physically deleted (i.e., unlinked from the list), still using `CAS`. To understand the reason behind this protocol, we should consider the insertion of a copy of one node performed by a delayed thread (e.g., one unscheduled by the operating system). In this scenario, the resize operation might be already finished, and a copy of the node could be already placed in `new_table`, and thus extracted. Thanks to the validation of the new copy, realized by publishing a pointer to it in the original node, we are sure that the new copy will be not validated. In fact, until the original node is not removed, it will reference its copy. This copy-based pattern allows to enforce lock-freedom. In fact, any thread can take on the job of concurrently migrating the same node, trying to finally flag the original node and its new copy.

5.1.4 Varying the bucket width

In the original, non-concurrent, calendar queue [12], a strategy is defined to compute the bucket width, which estimates the mean separation time (MST) between two consecutive events. The obtained MST is then multiplied by a constant equal to 3, which represents the desired number of events per bucket (denoted as $DEPB$). This should guarantee that the number of events in a bucket is bounded by a constant and thus the calendar queue should deliver O(1) amortized time complexity for its operations. However, the value of $DEPB$ is chosen according to an experimental evaluation performed by Brown, which shows that such a value delivers good performance under different distributions of the timestamps.

Moreover, the original calendar queue might suffer from reduced performance in at least two pathological scenarios. The first one occurs when the queue has reached a steady number of contained events, but the priority increment distribution changes over time, leading to a different MST compared to the one measured during the last resize. In such scenario we have that the bucket width is inappropriate, and performance deteriorates. In the second case, the priority increment distribution makes events be clustered into two buckets distant from each other and reduces the efficiency, since enqueues traverse a significant amount of events during insertions and dequeues scan a large number of empty buckets.

There are several works that try to resolve these issues by triggering the data-structure reshuffle more frequently and designing new strategies to individuate a more accurate bucket width. In particular, the authors in [99] state that the sampling of events should be obtained from the most dense buckets, namely, those that contain the highest percentage of events stored into the queue. An analytical model is presented in [100] for computing a scale factor for the actual bucket width such that the new bucket width minimizes (in the model) the cost of queue operations. The final bucket width computation resorts to a combination of the sampling technique described in [99] and a minimization technique. Moreover, the condition triggering the data-structure reshuffle is checked periodically. The time period is

selected accurately in order to maintain $O(1)$ amortized access time. Instead, the authors in [92] define a new data structure able to recursively split individual and dense buckets, making the bucket width be chosen properly for each bucket.

Anyhow, these works define heuristics and algorithms for computing MST, $DEPB$ and the bucket width for the case of sequential accesses. So, they do not account for the effectiveness of the proposals with concurrent, scalable, conflict-resilient multi-list event pools, like the one we are presenting. They try to minimize the average cost of queue operations by modeling properties of events, in particular MST, so they cannot capture dynamics and behaviors connected to interactions between concurrent threads. In order to cope with such aspects, so as to determine a suited bucket width for our concurrent event pool, we had to take into account the impact of retries on the cost of queue operations, which lead to repeat some computational steps, such as atomic instructions. In particular, when a dequeue finds an empty bucket, it tries to increase `current` with a `CAS` instruction. Updating such variable is an expensive operation, since it is likely contended among threads, and might at worst lead to thrashing behaviors. Consequently, it is reasonable that the number of events in a bucket should be at least T_d in our approach, where T_d is the number of CPU-cores expected to concurrently access a bucket for dequeue operations, which can be clearly less than the total number of CPU-cores available for running the share everything PDES system, which we denote as n. Anyhow, T_d can be much greater than the value of $DEPB$ used to compute the bucket width according to the rule proposed with, e.g., the original calendar queue. On one hand, this should not affect the asymptotic cost of dequeue operations, since a wider bucket width decreases the probability to scan a large amount of empty buckets. On the other hand, longer bucket lists affect the enqueue access time, since the enqueue has to scan an increased number of events before finding the position for the insertion. However, as shown in [90], a highly concurrent event pool based on a multi-list approach can have $O(N^2)$ times longer bucket lists, where N is the number of CPU-cores, than a baseline solution relying on spinlocks, and obtain comparable performance under high concurrency levels. Therefore, it follows that an average number $DEPB$ of elements within a bucket such that $T_d < DEPB << N^2$ can be a desirable value for optimizing the cost of both enqueue and dequeue operations in our proposal. Given that $T_d \leq N$ independently of the actual access pattern to the queue by concurrent threads, we suggest a value $DEPB \approx N$ as a suited one. In any case, in the experimental study of our proposal, we report data with different configurations of the parameter $DEPB$, just to capture the effects of possibly different parameterizations while determining the bucket width.

5.2 Experimental data

We experimentally evaluated the performance of our data structure with two different test settings. In the first one we exploited a synthetic workload based on the well-known *Hold-Model*, where *hold* operations, namely a dequeue followed by an enqueue, are continuously performed on top the queue, which is pre-populated with a given number of items at startup (referred to as queue size). This test allows evaluating the steady-state behavior of the queue and, when executed in a

Table 5.1. Employed Timestamp Increment Distributions

Probability Distribution	Formula
Uniform	$2 \cdot \texttt{rand}$
Triangular	$\frac{3}{2} \cdot \sqrt{\texttt{rand}}$
Negative Triangular	$3 \cdot (1 - \sqrt{\texttt{rand}})$
Exponential	$-\ln(\texttt{rand})$

multi-threaded fashion, its scalability and resilience to performance degradation in scenarios with scaled up volumes of concurrent accesses. The second test setting is related to the usage of the presented lock-free queue within a baseline share-everything PDES architecture, on top of which we run both the classical PHOLD benchmark for PDES systems and a multi-robot exploration model, called TCAR. The platform used in all the experiments is a 32-core HP ProLiant machine running Linux (kernel 2.6) equipped with 64 GB of RAM. The number of threads running the test-bed programs has been varied from 1 to 32.

5.2.1 Results with the Hold-Model

With the Hold-Model workload, the event pool is gradually populated with a given number of elements (queue size) and then each concurrent thread performs dequeue/enqueue operations with equal probability set to 0.5. Each run terminates when the total number of operations executed by the threads reaches 10^6. We run experiments with 4 different distributions of the priority increment for the generation of new elements to be inserted into the queue, which are shown in Table 5.1. For each distribution, we executed 4 tests with various queue sizes, namely 25, 400, 4000 and 32000. The performance of the proposed conflict-resilient calendar queue (CRCQ) is compared with the classical calendar queue [12] (SLCQ), whose concurrent accesses are synchronized via spin-locking, and the lock-free $O(1)$ event pool, still inspired to the classical calendar queue structure, presented in [91] (NBCQ).

The results are shown in Figures 5.3-5.6, where each sample is obtained as the average of 10 different runs of the same configuration. We report the wall-clock times required to perform the target number of operations while varying the number of threads from 1 to 32. As expected the evaluated data structures have $O(1)$ access time, in fact, once the number of running threads is fixed, the wall-clock time is constant for different combinations of queue sizes and priority increment distributions. Both the two tested lock-free solutions outperform SLCQ. However, our new proposal CRCQ shows an improved performance wrt the NBCQ in almost every scenario, just thanks to conflict resilience on dequeues. Moreover, any number of threads larger than 4 is enough to make CRCQ be the most efficient implementation among all the tested alternatives.

Although the wall-clock time shown by CRCQ slightly increases with more than 12 threads, it is reasonable to think that the flat line of NBCQ in Figures 5.5 and 5.6 does not reveal a behavior similar to the ones in Figures 5.3 and 5.4 just because

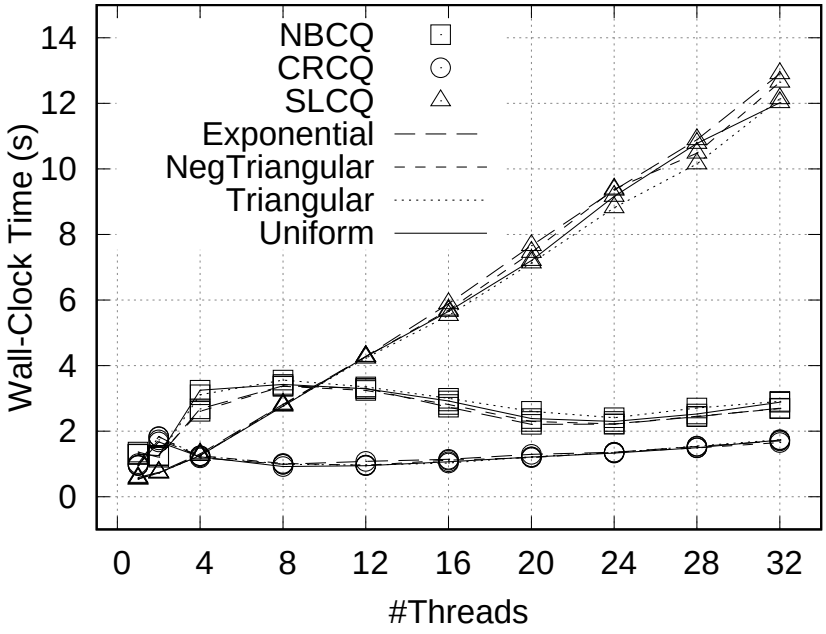

Figure 5.3. Hold-Model wall-clock times with average queue size equal to 25

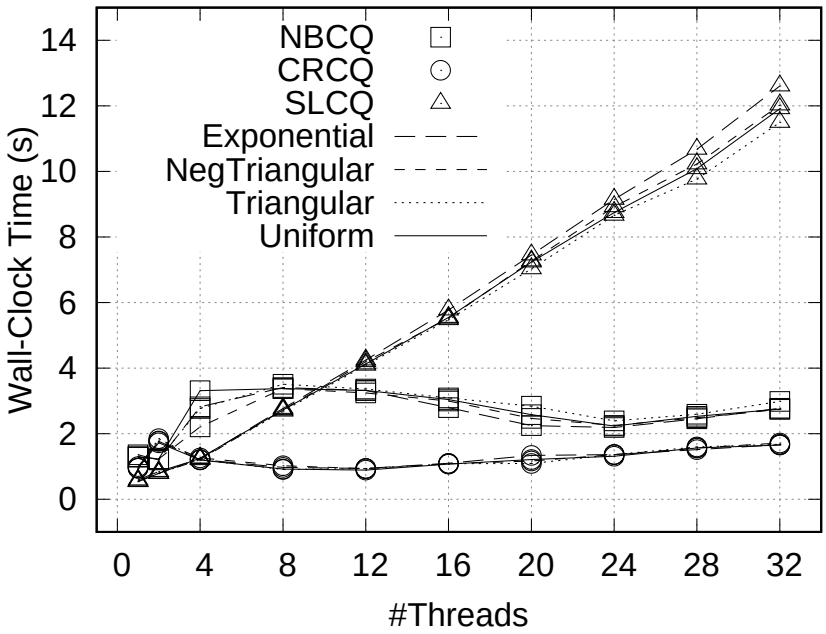

Figure 5.4. Hold-Model wall-clock times with average queue size equal to 400

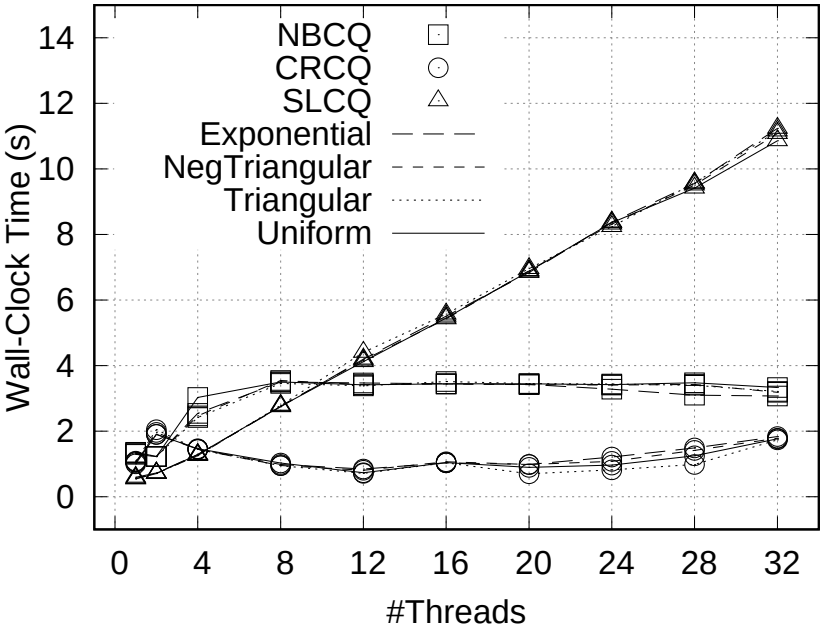

Figure 5.5. Hold-Model wall-clock times with average queue size equal to 4000

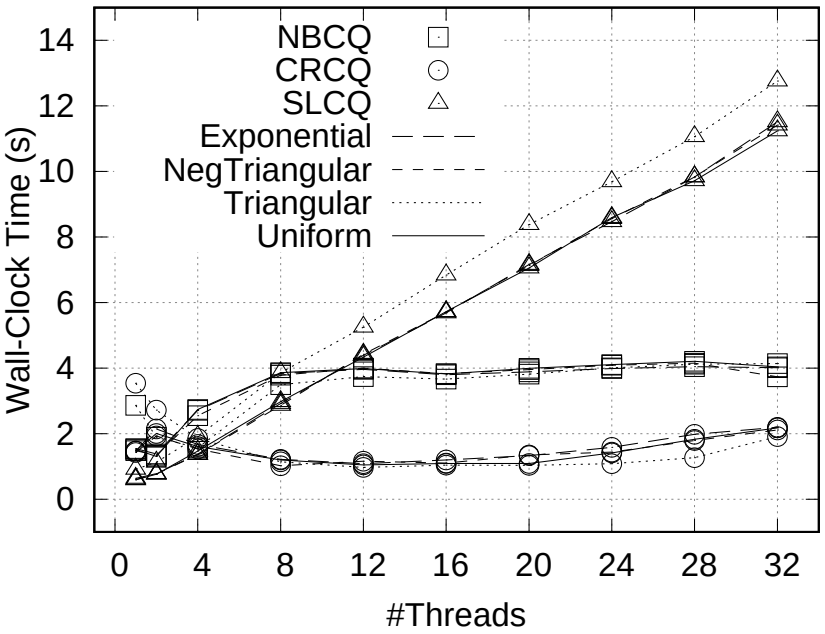

Figure 5.6. Hold-Model wall-clock times with average queue size equal to 32000

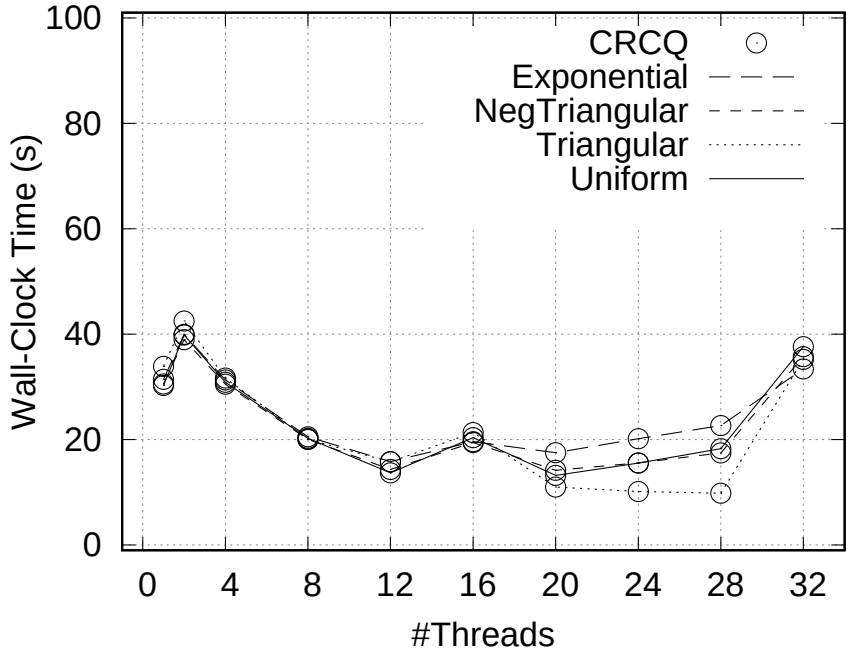

Figure 5.7. Hold-Model wall-clock times with average queue size equal to 32000 and $DEPB$ equal to 3

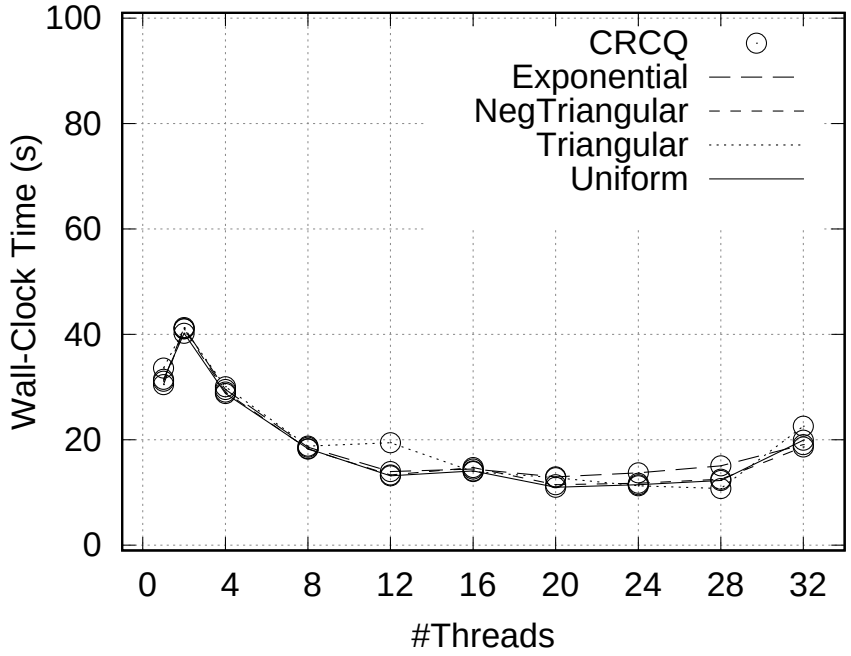

Figure 5.8. Hold-Model wall-clock times with average queue size equal to 32000 and $DEPB$ equal to 6

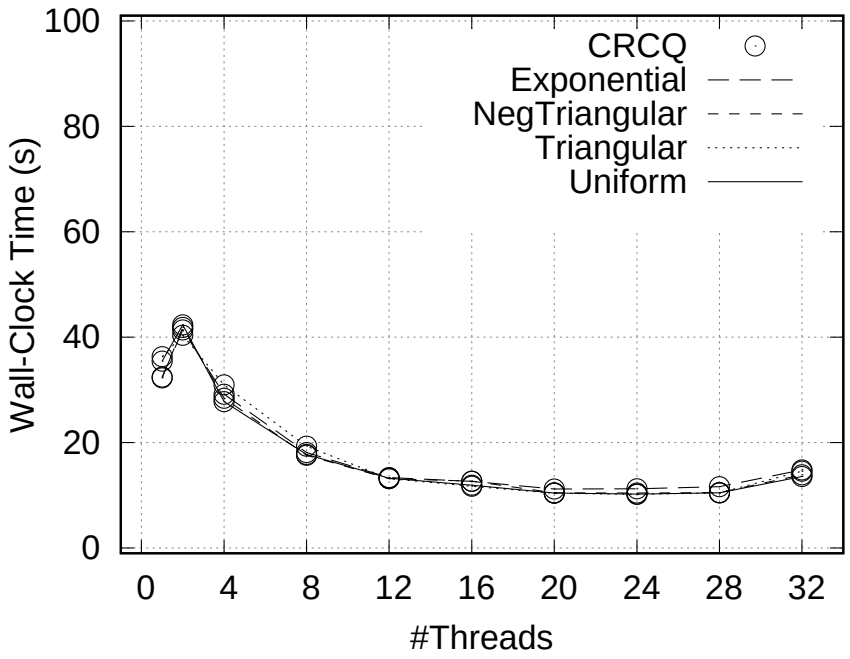

Figure 5.9. Hold-Model wall-clock times with average queue size equal to 32000 and $DEPB$ equal to 12

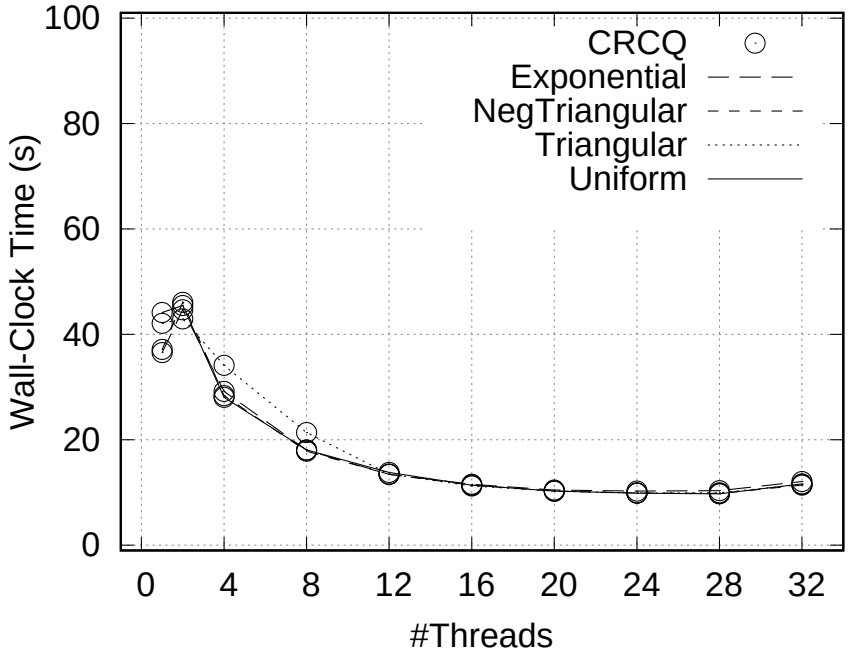

Figure 5.10. Hold-Model wall-clock times with average queue size equal to 32000 and $DEPB$ equal to 24

the minimum wall-clock time is expected with a higher number of threads compared to the maximum we could experiment with[3]. In other words, the higher efficiency of CRCQ allows to sense earlier the inherent bottleneck represented by retrieving the minimum from the event pool since, with reduced queue size, each bucket contains few items and consequently the probability that a dequeue finds an empty bucket is higher.

To confirm this hypothesis, we have run a synthetic test with a queue size equal to 32000, but this time with different values of $DEPB$, namely 3, 6, 12, 24. As shown in Figures 5.7-5.10, an increased number of items in a bucket allows to increase the performance improvement obtained by a scaled up number of threads with CRCQ, since this reduces the probability to find an empty bucket. Consequently, the amount of "wasted" local work of a thread is also reduced, thanks to conflict resilience of the proposed data structure, and at the same time there are less conflicts in updating/reading the value of `current`. Moreover, longer buckets act as an implicit back-off mechanism, exalting the resilience capability of our proposal vs conflicting concurrent accesses to the hot spot of the queue, namely the minimum timestamp event. We recall again this is materialized in only one item at any time. This also confirms the deductions in [90] related to the fact that the optimal bucket width in scenarios with concurrent accesses can be significantly different from the ones characterizing sequential implementations.

5.2.2 Share-everything PDES results

In order to test our proposal no longer as a standalone component, but rather when exploited in share-everything PDES, we have realized an early design of a share-everything PDES engine whose flow chart is shown in Figure 5.11. The ordering of the elements into the calendar queue is based on events' timestamp. Each event also keeps information regarding which is the target LP and the actual event's type and payload.

In this version of the share-everything PDES system, speculation along simulation time plays a limited role. More in detail, all the events kept in the calendar queue are *schedule-committed*, with the meaning that they will never be retracted (e.g., via negative copies) because of causality errors involving their parent events. In fact, in this early design of the share-everything PDES engine, events that are dynamically produced during the processing of an event which can be reverted are flushed to the calendar queue only if the speculatively executed event is eventually detected to be *safe* (i.e., causal consistent). This, in its turn, implies that the event is no longer rollbackable, therefore the output it has produced will never need to be undone. On the other hand, the undo on the updates of the state of the LPs by events that are eventually detected as causally inconsistent are undone by exploiting the reverse computing literature technique presented in [43].

We target the scenario where the maximum number of worker threads employed in the speculative PDES engine is upper bounded by the number of available CPU-

[3]In fact, experimenting with a number of threads greater than the number of available CPU-cores in the underlying platform would have generated scenarios of interference on CPU usage, possibly leading to unclear motivations for winning or losing on the comparison among the different solutions.

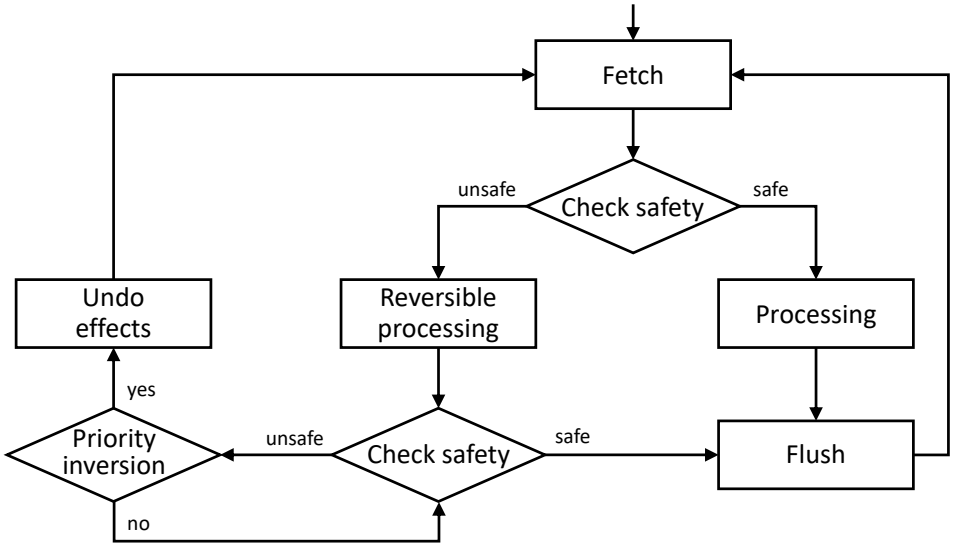

Figure 5.11. Main Loop Flow Chart of the employed share-everything PDES platform

cores. This is a classical configuration avoiding interference by deschedule/reschedule operations in parallel applications [101, 102, 103] at least for cases where the platform is temporarily dedicated to a specific application.

Due to the multi-threaded nature and the speculative flavor of the simulation environment, no two different worker threads can execute at the same time multiple events which entail reading/updating the same memory regions. Therefore, to enforce data separation, the simulation relies on an array of spinlocks, one for each LP. Whenever WT_i has to execute an event e, it tries to acquire the lock for the given recipient LP. In case the lock cannot be taken, it means that another worker thread is currently executing operations on the region's state. In this case, the worker thread spins on the lock, until the other worker thread completes its operations. As already noted, the achievement of joint non-blocking capabilities at the level of both event pool management (as we already do in the present solution) and LP state management is the object of the proposal in Chapter 6. On the other hand, this configuration of the share-everything PDES engine can still reveal highly effective in scenarios with non-excessively unbalanced distribution of the events across the LPs, so that the core data structure with conflicting operations by the worker threads is exactly the fully-shared event pool. Also, it can cope with (very) fine grain events, by avoiding the costs for managing output queues and retractions of events, which characterize classical speculative PDES.

A meta-data layer is used to track the LP currently being run by any thread and the timestamp of the corresponding event. When a worker thread has no LP to take care of (for event processing), it tries to extract an event to be processed by the calendar queue by performing a fetch operation. Specifically, it extracts the event e with minimum timestamp that is currently registered into the calendar queue (if any), and records atomically the extracted timestamp value and the corresponding LP id, associated with WT_i, into the meta-data layer. Thus, such layer allows to

compute the *commit horizon* of the simulation and to distinguish if an event can be affected by other events in the past or not. In the first case, the extracted event can be safely executed by triggering the native version of the application code. Otherwise, the worker thread speculatively processes the event by running a modified version of the event handler obtained thanks to a transparent instrumentation (an ad-hoc compile/link procedure) of the native code. The instrumented code generates at runtime undo blocks of machine instructions, which can be used to rollback updates performed by event processing according to the technique in [43]. Safety of the events in this baseline configuration of the engine is computed relying on the aforementioned meta-data layer. Differently by the solution exposed in the next chapter, the one of computing the safety of an event to be processed is a blocking operation.

The events produced by a speculative execution are locally stored waiting for the commitment of the event that generated them. If that event is not eventually committed because of a causality error, the local buffer of events is simply discarded without affecting the pending event set. Otherwise, in case it eventually become schedule-committed, then a *flush* procedure is called, which atomically inserts them into the calendar queue. Concerning execution liveness, if the LP lock is taken by any worker thread speculatively processing an event e associated with $T(e)$, and during its execution a new event e' associated with $T(e') < T(e)$ is flushed (e.g., by a worker thread executing in non-speculative mode) into the calendar queue, we might incur in a livelock. Thus, if a worker thread has executed speculatively an event which is (not yet) safe, it checks whether any other worker thread has extracted an event with higher priority destined to the same LP. In the positive case, the effects of the event's execution are undone (by simply jumping to the generated reverse window) and the region lock is released. In this case, the event stays bound to the worker thread (for re-processing).

The first test-bed application we run on top of the share-everything PDES engine is the classical PHOLD benchmark [14], whose high level representation of the classical LP behavior is schematized in Figure 5.12. We configured it with 1024 LPs. In PHOLD the execution of an event leads to update the state of the target LP, in particular statistics related to the advancement of the simulation, such as the number of processed events and the average values of the time advancement experienced by the LPs. It also leads to executing a classical CPU busy-loop for the emulation of a given event granularity. There are two types of events: i) *regular* events, whose processing generates new events of any type; ii) *diffusion* events that do not generate new events when being processed.

Thus, PHOLD can be configured in terms of event granularity (the amount of work performed in the busy-loop) and in terms of the number of events destined to other LPs. In our evaluation, the number of diffusion events generated by regular ones (denoted as Fan-Out) is set to 1 and 50. These two configurations of the event pattern lead to scenarios where the average number of events in the event pool is stable, but there are punctual fluctuations, which are more or less intense. In turn, these can allow assessing the effects of our proposal in scenarios where the actual locality of the activity (in particular extraction activities) bound to the hot bucket of the calendar queue can be more or less intense.

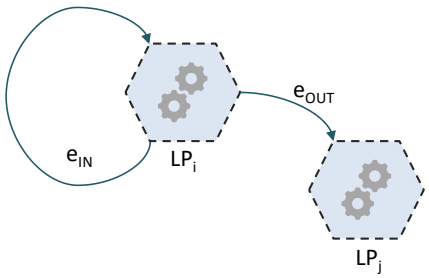

Figure 5.12. PHOLD model representation

The timestamp increments are chosen according to an exponential distribution with mean set to one simulation-time unit. We selected two different lookahead values, respectively 10% and 0.1% of the average timestamp increment, in order to observe the impact of this parameter on the run-time dynamics. Finally, the busy loop while processing an event is set to generate different event granularity in different tests, namely 60, 40 and 22 microseconds, in order to emulate medium to low granularity events proper of a large variety of discrete event models.

5.2.2.1 Results with PHOLD

In Figures 5.13-5.15 and Figures 5.16-5.18 we show the speedup achieved by the parallel runs wrt the sequential execution of the same configurations of PHOLD. Speedup values are shown while varying in the share-everything PDES system the number of threads from 1 to 32, and with different lookahead values (respectively 10% and 0.1% of the average timestamp increment). Whenever the event granularity is larger and the lookahead is large enough to make threads process events safely with high probability (Figure 5.15), we observe that both the lock-free event pools considered in our study allow an almost linear speedup with coefficient 1 (ideal speedup). This is observed up to 16 threads. When running with more threads, the speedup coefficient is 0.8 for CRCQ and 0.7 for NBCQ. Conversely, the spin-locked calendar queue has scalability problems with more than 8 threads and its efficiency is further negatively affected by an increased Fan-Out value.

Reduced lookahead (Figure 5.18) does not affect significantly the behavior of lock-free solutions and makes the spin-lock protected calendar queue to achieve an increased speedup when the Fan-Out is set to 1. The reason behind this behavior is that a smaller lookahead increases the probability to extract an unsafe event, that has to be executed speculatively and requires an explicit synchronization (possibly with rollback) in the worst case, reducing the actual concurrency in accessing the shared event pool.

A reduced event granularity leads to a higher pressure on the shared event pool by enlarging the volume of concurrent accesses. In particular, event granularity set to 40 microseconds and lookahead set to 10% (Figure 5.14) make the spin-locked calendar queue deteriorate after 8 threads and NBPQ after 24 threads (mostly because of conflicts and retries on dequeue operations),while CRCQ still delivers linear speedup.

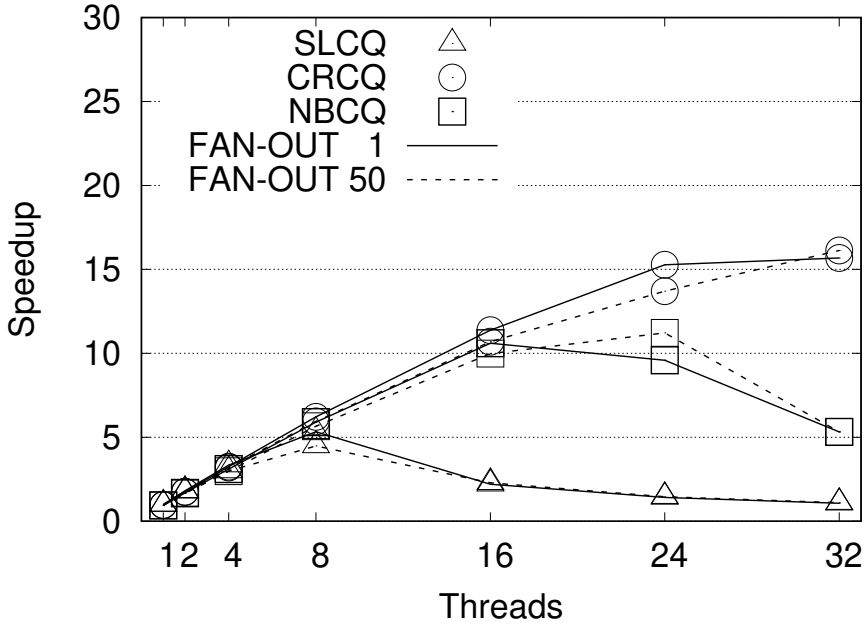

Figure 5.13. PHOLD speedup results, lookahead equal to 10% of the average timestamp increment and event granularity equal to 22 μs

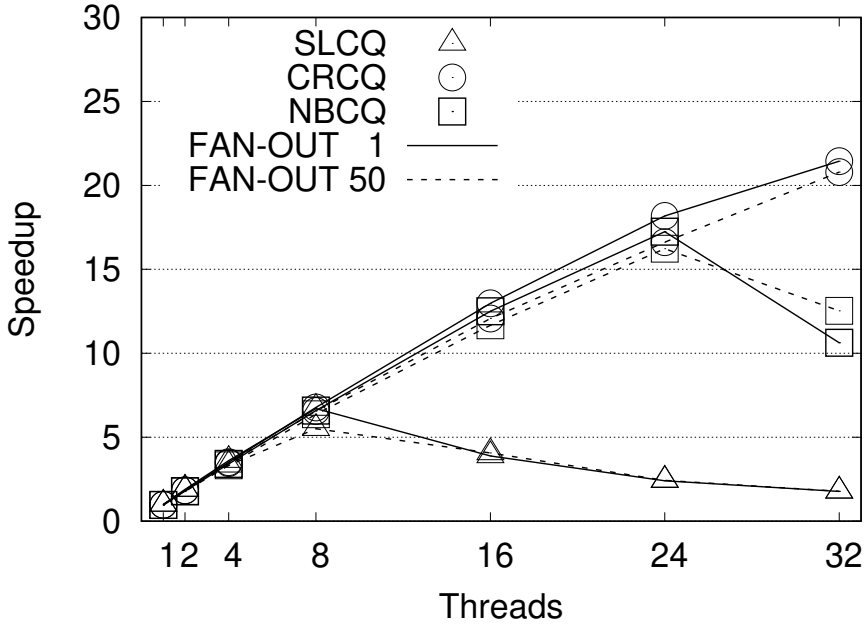

Figure 5.14. PHOLD speedup results, lookahead equal to 10% of the average timestamp increment and event granularity equal to 40 μs

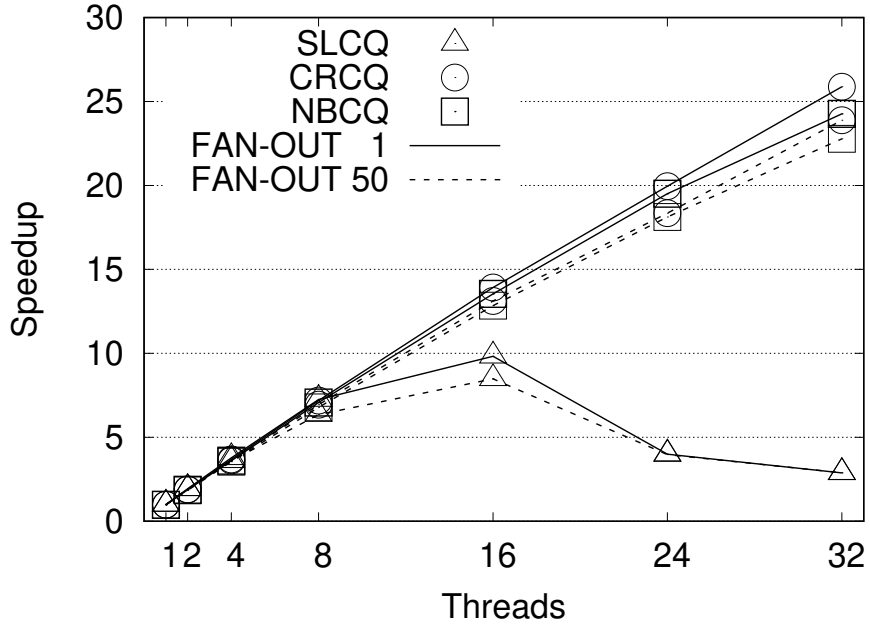

Figure 5.15. PHOLD speedup results, lookahead equal to 10% of the average timestamp increment and event granularity equal to 60 μs

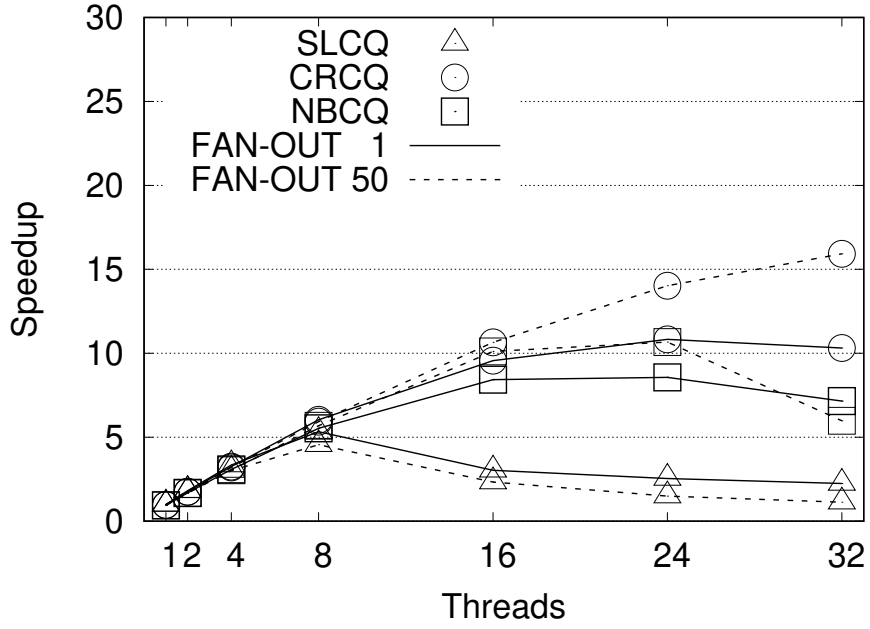

Figure 5.16. PHOLD speedup results, lookahead equal to 0.1% of the average timestamp increment and event granularity equal to 22 μs

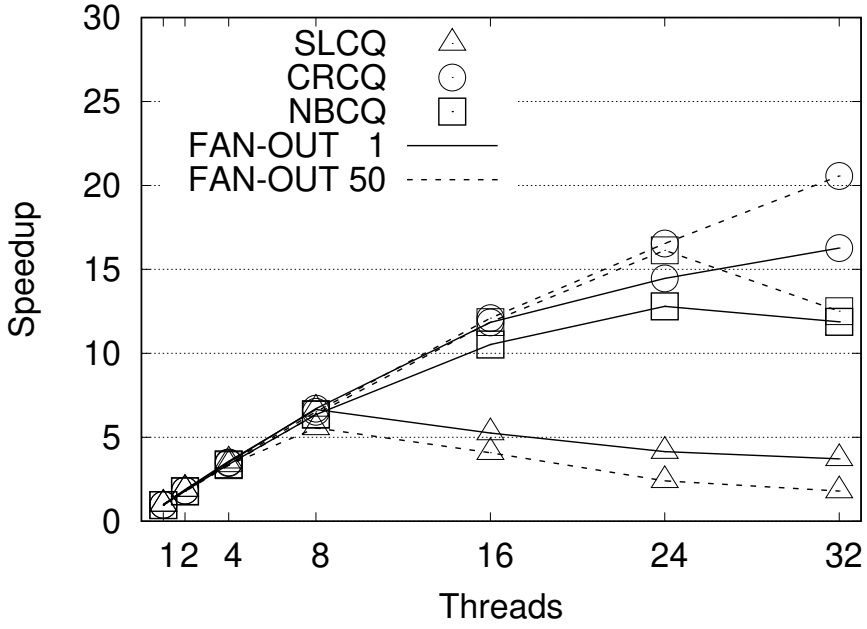

Figure 5.17. PHOLD speedup results, lookahead equal to 0.1% of the average timestamp increment and event granularity equal to 40 μs

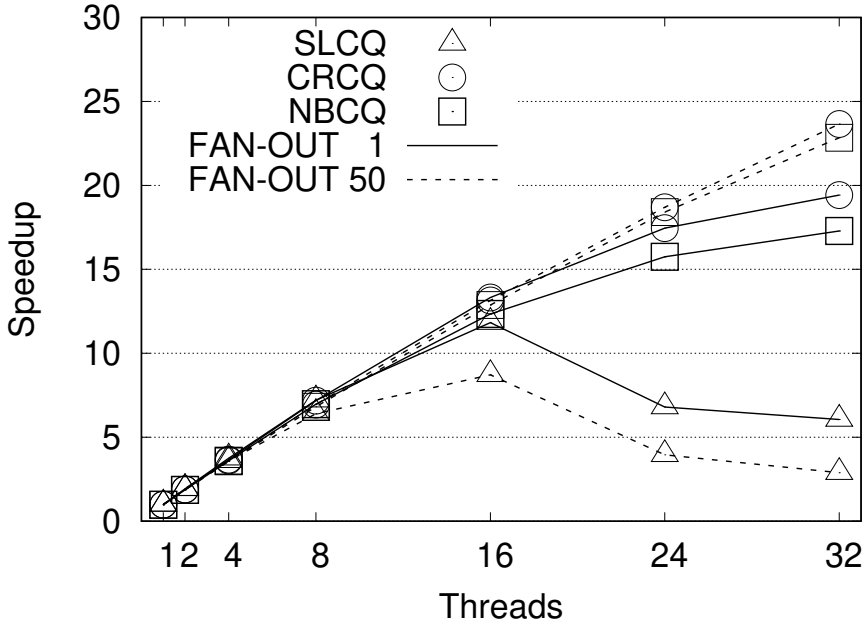

Figure 5.18. PHOLD speedup results, lookahead equal to 0.1% of the average timestamp increment and event granularity equal to 60 μs

Reduced lookahead (Figure 5.17) leaves almost unaltered the speedup of both the lock-free queues, but leads to a trend reversal. In particular, a larger Fan-Out makes lock-free solutions achieve higher performance (conversely the spin-locked calendar goes worse) since it balances the trend of reduction of concurrency in accessing the shared event pool due to synchronization at the meta-data layer (thanks to more intense bursts of enqueue operations). Anyhow also in this adverse scenario the proposed solution still delivers an almost linear speedup.

Finally, a very fine grain event, namely 22 microseconds, leads to a scenario quite similar to the tests with the Hold-Model, where increasing the number of threads leaves unaltered the wall-clock time spent into the conflict-resilient calendar queue. In Figure 5.13 it is shown how the speedup slope is reduced when moving from 24 to 32 threads in our solution (anyhow the speedup is still 0.5 of the ideal one with 32 threads), while a smaller lookahead (see Figure 5.16) makes our proposal still achieve the same speedup of larger lookahead scenarios when Fan-Out set to 50, and gives no speedup advantages when moving from 24 to 32 number of threads with the smallest Fan-Out, thus indicating how CRCQ does not negatively impact the absolute performance when moving towards the usage of the maximum admitted parallelism level.

5.2.2.2 Results with TCAR

As the second PDES test-bed, we used a variant of the Terrain-Covering Ant Robots (TCAR) model presented in [104]. In this model, multiple robots (agents) are located into a region (the terrain) in order to fully explore it.

TCAR simulations are usually exploited for the what-if analysis in rescue scenarios. In particular, if some kind of accident occurs in a region which is either unknown by the rescuers or altered by the accident itself (e.g., due to explosions or collapses), the first action in order to actually rescue the victims is to explore the whole region to determine a plan. In this simulation scenario there is a non-negligible trade-off: the higher the number of robots (agents) injected in the rescue terrain, the faster is the full exploration of the region, but (at the same time) the higher the cost. Simulation can provide rescuers with the optimal number of ant robots which must be unleashed in the terrain to fully cover it in a given time.

In our implementation of TCAR, the terrain is represented as an undirected graph, therefore a robot (i.e., an ant robot) is able to move from one space region to another in both directions. This mapping is created by imposing a specific grid on the space region. The robots are then required to visit the entire space (i.e., cover the whole graph) by visiting each cell (i.e., graph node) once or multiple times. Differently from the original model in [104], we have used hexagonal cells, rather than square ones. This allows for a better representation of the robots' mobility featuring real world scenarios since real ant robots (e.g., as physically realized in [105]) have the ability to steer to any direction during the exploration.

Robots start from specific border cells in the terrain, and from each cell a given number of robots starts moving around (mimicking the fact that rescue teams start from specific positions and unleash robots for discovery). Overall, the simulation scenario is depicted in Figure 5.19.

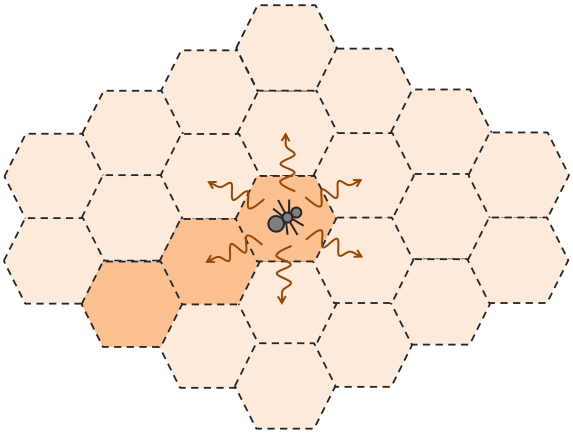

Figure 5.19. TCAR representation

The TCAR model relies on a node-counting algorithm, where each cell is assigned a counter that gets incremented whenever any robot visits it. So, the counter tracks the number of *pheromones* left by ants, to notify other ones of their transit. Whenever a robot (i.e., an ant robot) reaches a cell, it increments the counter and determines its new destination. Choosing a destination is a very important factor to efficiently cover the whole region, and to support this the trail counter is used. In particular, a greedy approach is used such that, when a robot is in a particular cell, it targets the neighbor with the minimum trail count. A random choice takes place if multiple cells have the same (minimum) trail count. Although this greedy approach might not be optimal, it allows for a complete coverage of the region taking into account the simplicity of the agents, which may have a very limited and noisy sensing capability. In the original model, whenever a robot is in a given cell, it accesses the information stored in the neighbor cells (i.e., trail counters) to make its decision.

The original TCAR model adopts a *pull* approach for gathering trail counters from adjacent cells. Considering the traditional PDES programming model, based on data separation across the LPs, such an approach would result in sending query/reply events across objects modeling adjacent cells each time a robot needs to move to some new destination cell.

To reduce the interactions across the LPs (by reducing the volume of events to be scheduled along the model lifetime) we adopted a *push* approach, relying on a notification event (message) which is used to inform all neighbors of the newly updated trail counter whenever a robot enters a cell. Then, each LP modeling a cell stores in its own simulation state the neighbors' trail-counters values, making them available to compute the destination when simulating the transit of a robot. In the used TCAR configuration, we included the evaluation of a new state value for the cell whenever a robot enters it, so as to mimic the evolution of a given phenomenon within the cells.

This computation has been based on a linear combination of exponential functions (like it occurs for example when evaluating fading on wireless communication systems due to environmental conditions). Further, to model the delay robots experience when entering a cell for correctly aligning itself spatially, a lookahead of

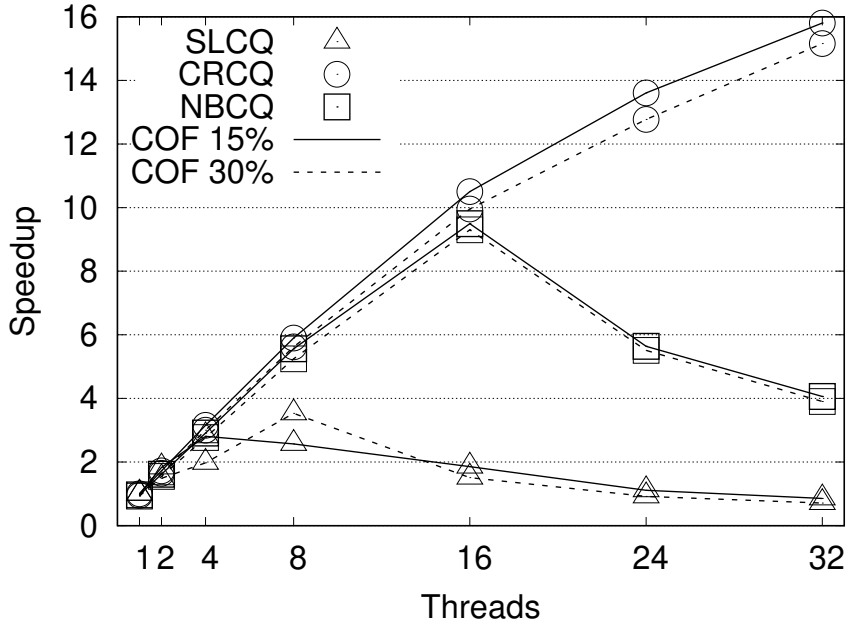

Figure 5.20. TCAR speedup results

10% of the average cell residence time has been added when generating new events used to notify the update of the local trail counter to the neighbors. We configured TCAR with 1024 cells, and we studied two alternative scenarios with different ratios between the number of robots exploring the terrain and the number of cells, say 15% and 30%. We refer to this parameter as Cell Occupancy Factor (COF).

Speedup results are shown in Figure 5.20, where we still keep the PDES engine configuration with spin-lock protected calendar queue as the reference. By the plot we see how, also for this application, the maximum thread count leading the spin-lock protected calendar queue to be competitive is between 4 and 8, depending on the value of COF. Beyond 8 threads, such configuration rapidly degrades. NBCQ scales well up to 16 threads, but then degrades, while CRCQ provides close to linear speedup up to 32 threads, jointly guaranteeing at least 50% of the ideal speedup value even for larger thread counts.

Figure 5.21 provides a finer grain representation of the CPU cycles spent by the different configurations of the PDES engine in the different layers/operations. Here the cycles spent in waiting to access to the spin-lock based calendar queue is considered as part of the enqueue/dequeue operations. By the results we observe that CRCQ allows reducing the time spent in enqueue and dequeue operations significantly, compared to SLCQ, with scaled up gains at larger thread counts. This is also reflected into a reduction of the amount of clock cycles spent at the meta-data synchronization layer when using non-blocking approaches, and in particular CRCQ. In more detail, delaying the enqueue of new events upon committing a processed event, as it occurs with SLCQ, leads to reducing the speed of advancement of the commit horizon of the simulation, since the delayed threads cannot participate with new extracted event-timestamps to the reduction leading to the commit horizon

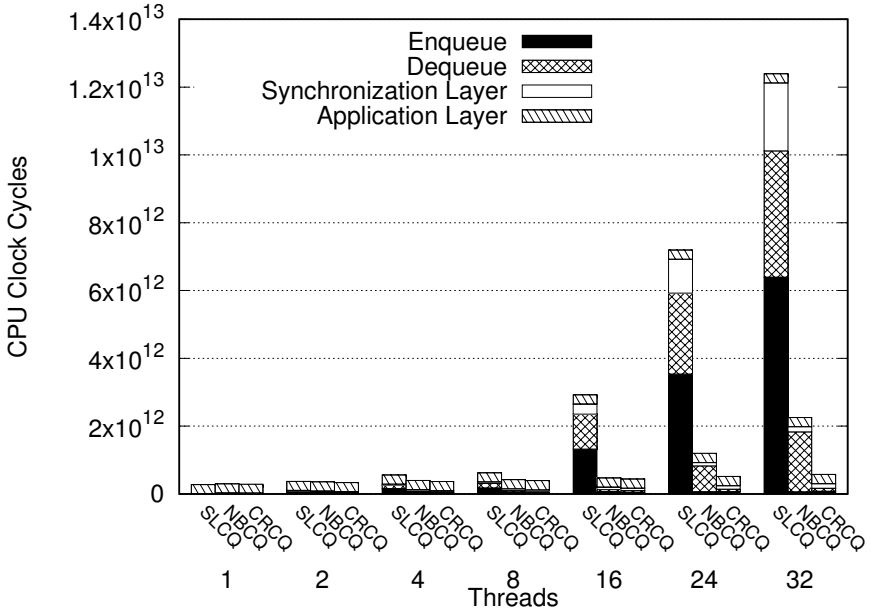

Figure 5.21. TCAR execution profile, COF set to 30%

advancement. Moreover, the increased queue latency due to spin-locking activity leads to scenarios where the probability to extract an unsafe event is much higher. This leads threads to stall for longer time before being able to commit the events they just executed speculatively, a problem which is avoided with non-blocking solutions.

Comparing the two tested lock-free event pools it is possible to observe that the time spent in performing enqueue operations appears to be independent of the concurrency level. This is an expected result since enqueue operations, differently from dequeues, are expected to be distributed on a larger interval along the virtual-time axis, thus leading to reduced conflict and retry probability at any level of concurrency. Moreover, CRCQ shows a negligible rise of time spent in dequeue operations compared to the gain obtained thanks to the increase of the thread count, showing a constrained linear increment of cost. Contrarily, the NBCQ shows a superlinear increment of the cycles spent by the threads in dequeue operations, losing profits with larger thread counts. This is somehow expected since the bucket containing the highest priority (lowest timestamp) event is targeted by a higher volume of concurrent dequeues, all insisting on the head element of the bucket itself, leading the NBCQ to abort/retry a dequeue multiple times when conflicts materialize. Clearly, this phenomenon is also linked to the very fine grain nature of TCAR, in fact coarser grain models—like the 60 microseconds PHOLD configuration—lead to increase the percentage of time spent by threads in the application layer, thus reducing the volume of concurrent accesses for dequeue operations and the consequent likelihood of conflicts and retries. CRCQ directly tackled such a problem since the conflicts on a dequeue operation, namely `Fetch&Or` updates on a just deleted node, do not bring to retry the whole dequeue task. Moreover, leaving deleted nodes on the current bucket, combined with longer lists, act as a natural back-off mechanism, tending to reduce the collision probability.

Chapter 6

Non-blocking task scheduling

In the previous chapter we have provided a solution for the engine level share-everything PDES coordination by devising a highly concurrent shared-event pool supporting conflict resilient non-blocking operations. This solution allows achieving scalability of engine-level tasks, in particular by preventing extraction and insertion operations from/to the event pool to become a bottleneck in the share-everything PDES scenario.

However, this solution alone does not allow to cope with thread coordination issues related to the access by threads to the states of the LPs involved in the simulation model. More in detail, if two or more threads concurrently pick different events destined to the same LP, they incur the risk of blocks and sequentialization in the access to the LP state (as it occurs in the baseline version of the share-everything PDES engine provided in the previous chapter). In fact, the common programming model for PDES is such that the application code is designed in such a way to have an LP representing a sequential entity, on whose state image a single thread is enabled to work at any given point in time in order to not impair safety of software actions because of concurrent conflicting accesses.

Overall, two threads that request the access to the state of a same LP would need to explicitly coordinate to avoid inconsistent state manipulations. Also, the block is (hopefully) implemented via spin-locks in order to avoid operating-system thread-reschedule delays caused by kernel-level blocking synchronization services. As a consequence, an additional penalty comes out from the energy-waste generated by the corresponding busy-waiting CPU cycles.

In order to cope with these shortcomings, in this chapter we present the design of a share-everything PDES system where no thread is ever blocked because of a conflicting concurrent access to some engine/application data structure. So, we further raise the bar in order to focus not only on concurrent data manipulations targeting the fully-shared event pool, but rather on data manipulations targeting whatever data structure within the share-everything PDES system.

This result has been achieved by fully revising the CPU-dispatching rules that are put in place by worker threads, which are in this proposal no longer based on extracting the element with the current minimum timestamp from the fully-shared event pool (as an individual action) and then on locking the destination LP

(as a final action for CPU-dispatching the event). Rather, the pool is accessed in non-blocking fashion by selecting for processing an event destined to an LP that is currently not active—not already CPU-dispatched by any other worker thread. This is discovered exactly while traversing the event pool, still in non-blocking fashion, thanks to the introduction of new signaling mechanisms based on metadata that are used to indicate the state of each LP (CPU-dispatched or not), which are manipulated in combination with the access to the record representing an element of the event pool. These metadata are manipulated atomically in non-blocking mode via the `Compare&Swap (CAS)` machine instruction.

Overall, in this further enhanced proposal for the achievement of a non-blocking share-everything PDES engine we rely on a vertical approach where the state of the event pool, in combination with the augmented metadata, does not only keep into account what events need to occur at a given LP. Rather it also expresses the current state of the LP, namely whether it is already CPU-dispatched or not, and the whole information is accessed/manipulated in non-blocking mode.

This design is towards a fully non-blocking speculative PDES engine, where the actual need for block phases is only determined by the setup level of speculation. We set this level to one for each LP—thus each LP is allowed to perform a step ahead in logical time speculatively—still relying on a shared pool containing only schedule-committed events. This means that, in this organization an event cannot be annihilated, but rather it could be rolled back and executed again if found to be causally inconsistent.

The topic of enhancing the speculation level, and thus of providing fully non-blocking coordination of tasks along simulation time, will be faced in the next chapter. However, this facility (namely, non-blocking coordination in the virtual-time dimension) can be seen as orthogonal to the core CPU-dispatching mechanisms of events that we present in this chapter. In any case, even though we target speculative event processing to some extent, in this enhanced proposal we are still able to exploit lookahead information in order to detect whether some event is safe to be processed and does not require reversibility supporting actions. This allows us to enable LPs to perform as many steps ahead as possible in non-speculative mode before trapping into speculative execution of some event.

6.1 The task scheduling algorithm

As hinted, non-blocking task scheduling is achieved in this chapter by exploiting the conflict resilient event pool previously presented as the baseline, and modifying the representation of the corresponding data structure via the integration with additional information. Also, this time we do not only perform operations of insertion and deletion of records into/from the fully-shared event pool, but rather we jointly perform, still in non-blocking fashion the additional operations required for avoiding collisions of the different WTs on the state of a same LP. As it will be clear later in our explanation, the logic associated with the additional operations we put in place on the non-blocking event pool are actually independent of the structure of the event pool itself. Hence, we will see the non-blocking event pool can be seen as an abstraction on top of which we devise the additional operations. Consequently,

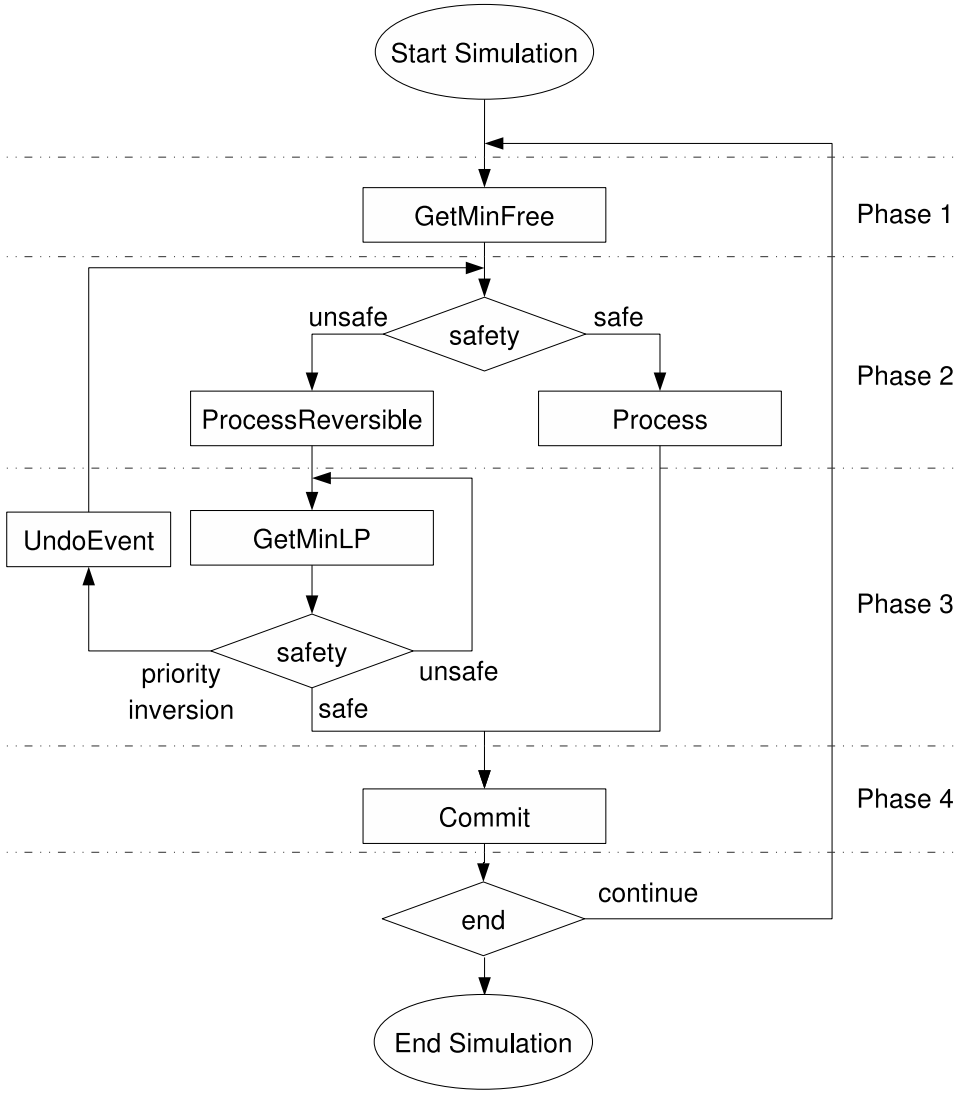

Figure 6.1. Diagram of the simulation loop executed by WTs

the new algorithmic proposals we provide in this chapter are combinable with—but still independent of—those presented in Chapter 5.

In the modified version, our non-blocking event pool is coupled with information related to the current CPU-dispatch of the LPs among WTs. In other words, the pool is associated with information related to a kind of *short-term* binding of LPs to WTs. So, we introduce in the reshuffled data structure (and in the associated management logic) the concept of booking of LPs by WTs, so that a non-blocking dequeue operation by some WT that tries to CPU-dispatch an event destined to some LP does not actually remove the corresponding event-record item from the shared pool, but rather it simply "books" the target LP, thus putting in place the aforementioned short-term binding.

The event records are not removed when the corresponding LP is booked and they are being processed since, as we shall explain, their presence in the event pool

Algorithm 6.1 Main Loop

```
N1:  procedure MainLoop( )
N2:      Set<event> newEvents ← ∅
N3:      bool safe ← FALSE
N4:      event e, e′ ← NULL
N5:      while ¬endSimulation do
N6:          ⟨e, safe⟩ ← GetMinFree( )
N7:          if e = NULL then
N8:              continue
N9:          end if
N10: execute:
N11:         if safe then
N12:             newEvents ← Process(e)
N13:         else
N14:             newEvents ← ProcessReversible(e)
N15:             do
N16:                 ⟨e′, safe⟩ ← GetMinLP(LP(e))
N17:                 if e ≠ e′ then
N18:                     UndoEvent(e)
N19:                     e ← e′
N20:                     newEvents ← ∅
N21:                     goto execute
N22:                 end if
N23:             while ¬safe ∧ ¬endSimulation
N24:         end if
N25:         if endSimulation then
N26:             break
N27:         end if
N28:         Commit(e, newEvents)
N29:     end while
N30: end procedure
```

is exploited for detecting the safety[1] (causal consistency) of the events themselves. So, the booking of an LP figures out as a logical removal of some corresponding event, which is not yet finalized into a definite removal. The latter takes place when event processing is committed. Consequently, the event pool data-structure has an additional API to effectively remove some item from the event pool, so as to perform garbage collection of event records whenever they are no longer needed—since their processing is related to a definitive committed action.

At a logical level, to hide away the complexity of the queue implementation, we abstract it as a generic non-blocking timestamp-ordered event pool. It offers anyhow a couple of additional API functions. The first one (GetMin) allows to retrieve the highest-priority event. The second one (GetNext) allows to pick the

[1] A formal definition of *safety* is provided in the next chapter.

Algorithm 6.2 GetMinFree procedure

```
L1:  procedure GetMinFree( )
L2:      Set S ← NULL
L3:      node n ← GetMin()
L4:      time min ← n.ts
L5:      while ¬TryLock(n.lp) do
L6:          S.add(n.lp)
L7:          n ← GetNext(n)
L8:      end while
L9:      if n.ts < min + LOOKAHEAD ∧ ¬n.lp ∈ S then
L10:         return ⟨n.event, TRUE ⟩
L11:     else
L12:         return ⟨n.event, FALSE ⟩
L13:     end if
L14: end procedure
```

event immediately following the one passed as argument. This latter function allows for a timestamp-ordered traversal of the pool.

The overall activities that are carried out by the WTs are schematized by the pseudocode shown in Algorithm 6.1. The loop executed by WTs can be logically divided in four different phases, as shown in Figure 6.1. Initially a call to the GetMinFree procedure is executed in order to retrieve from the shared event pool an event to process. As the name of the procedure suggests, in this phase we try to pick an event for processing destined to some "free" LP—say not currently CPU-dispatched. Also, the event should have the minimum possible timestamp, although it might not coincide with the absolute minimum in the event pool.

The pseudocode for the GetMinFree procedure is shown in Algorithm 6.2. This procedure traverses the event pool starting from the head. At each traversal step of an event, the WT tries to book the corresponding destination LP. Operatively, the booking is based on executing a `CAS` machine instruction that implements a non-blocking try-lock operation on a variable associate with the LP to be booked.

If booking fails it means that some other WT is already working on the LP. In this case, the WT continues traversing the event pool for a new try with some subsequent event. When an event whose corresponding LP can be booked is found, the event is returned, although it is not definitely removed from the pool. Moreover, while searching for an event to take care of, the procedure is able to determine the safety of such event. In more details, the procedure keeps track of the LPs met depending on the traversed events while scanning the pool, which were already found to be booked, and of the timestamp of the first event met during this traversing, namely the one with the highest priority in the event pool. Considering the possibility of having a lookahead specification for the simulation model, an event can be returned as safe for processing if the difference between its timestamp and the minimum one still recorded into the event pool (the GVT) is smaller than the lookahead value. If this is true, it means that no other event will ever be delivered—because of pending processing activities at any LP—to the same LP targeted by the returned event in its past.

Algorithm 6.3 GetMinLP procedure

```
S1:  procedure GetMinLp(int lp)
S2:      node n ← GetMin()
S3:      time min ← n.ts
S4:      while (n.lp ≠ lp) do
S5:          n ← GetNext(n)
S6:      end while
S7:      if n.ts < min+ LOOKAHEAD then
S8:          return ⟨n.event, TRUE ⟩
S9:      else
S10:         return ⟨n.event, FALSE ⟩
S11:     end if
S12: end procedure
```

On the other hand, concurrent processing of multiple events at a same LP is prohibited because of the booking mechanism, which leads to safety of processing—namely WT isolated processing on a given LP—even in scenarios where an event destined to the same LP will be successively flushed to the event pool with a timestamp lower than the currently selected event. This situation, if materialized, will be resolved via rollback, as we shall discuss. In any case the check in line 9 also covers the scenario where the booked LP corresponds to one that was previously attempted to book and that has been in the meanwhile released by some other WT. If this scenario materializes, the picked event cannot be considered safe, independently of the lookahead, since there still could be another event to be processed at the same LP which stands in the past of the picked one.

Once the GetMinFree procedure ends, it returns an event associated with an LP that is univocally bound to the WT, together with the indication of whether the event is safe or needs to be processed speculatively. Note again that the event is still available in the event pool. On the other hand, it can be considered as *logically extracted* since, until the WT will keep the corresponding LP booked, no other WT will extract events destined to the same LP.

At this point, according to the *safe* value returned by the GetMinFree procedure, two different execution paths are possible: safe and unsafe. A safe execution leads the WT to CPU-dispatch the non-instrumented event handler (the one not embedding reverse computing capabilities), which installs the updates on the LP state in non-reversible mode. Instead, for an unsafe execution, we dispatch the instrumented version, which entails undoing capabilities. At the end of the speculative execution of an event e, in order to preserve causal consistency and schedule-commitment for all the events inserted into the shared pool, the WT has to wait that the executed event becomes safe. This check is done via the GetMinLP procedure, whose pseudocode is shown in Algorithm 6.3. The execution path of the GetMinLP procedure is somehow similar to the one of the GetMinFree procedure. However, as the name suggests, this time we are not looking for a generic event within the pool, but rather we are trying to discover what is the minimum timestamp event destined to the very same LP that was targeted by the event e, in the hope such an event corresponds to e and has become safe in the meanwhile.

Algorithm 6.4 Commit procedure

```
C1: procedure COMMIT(event e, Set<event> E)
C2:     ∀e' ∈ E: INSERTINEVENTPOOL(e')
C3:     DELETE(e)
C4:     UNLOCK(LP(e))
C5: end procedure
```

In this procedure, the head of the pool is retrieved, and the pool is traversed searching for the first event targeted at the same LP of the event e. Once found, the event is returned together with its safety condition, this time based only on the distance from the commit horizon and the lookahead. It is important to note that we have no guarantee on which event is fetched this time, in fact, if the procedure returns an event different from e it means that a priority inversion has occurred—namely, event e has been executed out of timestamp order. In this case, the processed event e is rolled back executing undo-code blocks generated by instrumented handlers and the loop is restarted with the newly extracted event. If there is no priority inversion, the GetMinLP procedure is repeatedly called until the processed event is returned as safe. When this condition occurs, the event is finalized towards its commitment by proceeding to the next phase of the main loop in Algorithm 6.1.

The pseudocode of the Commit procedure is shown in Algorithm 6.4. The Commit procedure first places into the global pool all the newly produced events and successively eliminates the just committed event e making the commit horizon progress. Once the operations on the pool are completed, the LP's lock is released, and the loop is restarted checking each time if the simulation is completed.

We remark again that all the operations on the event pool (insertions/traversals/deletions) are implemented in non-blocking mode according to the solution presented in the previous chapter. Clearly, the smart dispatching mechanism we introduce based on LPs' booking adds a new dimension in terms of scalability by avoiding WTs' blocks for accessing the LPs' states in scenarios with full sharing of the LPs and of their workload.

6.1.1 Optimization

The GetMinFree operation is one of the most onerous in our PDES engine. In order to reduce the number of atomic operations, namely `CAS` instructions, to be performed to support the booking mechanism, nodes have been augmented with a *reserved* field. This is updated by a classical write once booked the corresponding LP. This field is not required for correctness, therefore if an update is lost—given that it is not performed using atomic operations—no problem actually arises since, in scenarios of false negatives for node reserving, the WT will find the corresponding LP already booked. On the other hand, if a WT finds a node reserved, it will not try to book the corresponding LP via `CAS`, thus avoiding at all the cost of this atomic operation.

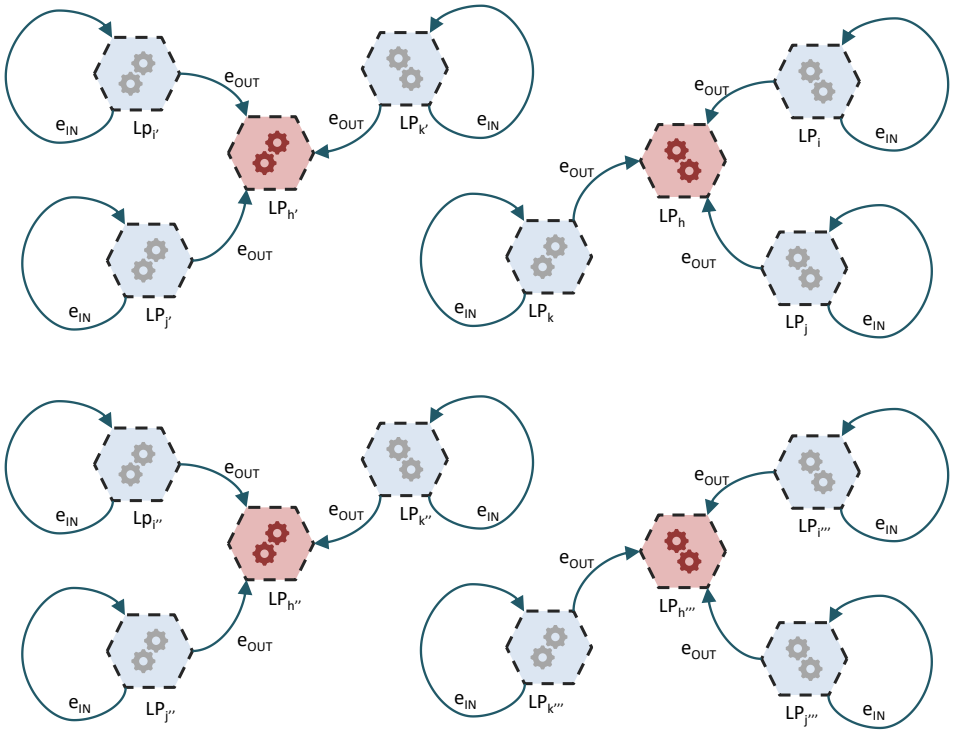

Figure 6.2. PHOLD hot spot model representation

6.2 Experimental results

We have integrated our proposal in a new platform release of our open source share-everything PDES project[2]. In this section we report a performance comparison of this proposal vs the previous release of that same engine based on non-blocking event pool operations, but still on blocking access to the LPs' states.

As test-bed application we used a variation of the classical PHOLD benchmark [14], previously exposed in Section 5.2.2.1, configured with 1024 LPs. Each LP schedules events for any other LP in the system, with an exponential timestamp increment. As usual for PHOLD, event processing leads to spending some CPU time, via a busy loop emulating a given event granularity. In our experiments we initially set the loop to give rise to events with granularity of the order of 60 microseconds, which can be considered as a mid-weight value.

In our PHOLD configuration we included 10 hot-spot LPs, towards which a given percentage of events injected by the other LPs are routed. This percentage has been varied from 25% to 100%, passing through 50% and 75%. A representation of this configuration of PHOD is provided in Figure 6.2.

It is known that PDES workloads with hot spots are difficult to manage since they might show unbalanced dynamics in case of traditional PDES platforms relying in the binding between LPs and WTs. Also, they are difficult to manage in share-everything PDES systems where WTs can block one another because of the need to process events on a same LP (the hot-spot one). We decided to experiment with

[2]Available at `https://github.com/HPDCS/USE/tree/DSRT2017`

this kind of complex workload just to study how our new approach could overcome such known limitations.

All the tests have been run on a HP ProLiant machine equipped with four 2GHz AMD Opteron 6128 processors. Each processor has 8 physical cores, for a total of 32 CPU-cores, which share a 12MB L3 cache (6 MB per each 4-cores set), and each CPU-core has a 512KB private L2 cache. The machine is equipped with 64 GB of RAM—organized in 8 NUMA nodes—and we used Linux (kernel 3.2) as the operating system. This is the same platform used for the experiments presented in the previous chapter. The number of WTs running within the PDES platform has been varied from 1 to 32 in order to perform a scalability study. All the reported data points have been computed as the average over 10 runs, executed with different seeds for the pseudo-random generation of event timestamps.

As a final preliminary note, we set the lookahead of the PHOLD model to the 10% of the average timestamp increment of newly generated events. This enabled us to study the effects of our non-blocking approach in scenarios where the need to wait for the safety of speculatively processed events does not become the major limiting factor to scalability.

In Figures 6.3-6.6 we show results related to the speedup observed while varying the number of WTs for the different configurations of the PHOLD model with hot spots. Speedup results have been computed over a sequential run of the same PHOLD model carried out on a classical calendar-queue scheduler. The two reported plots in each graph refer to our new non-blocking share-everything PDES engine (NBSE) and to the one based on LPs' states locks (SLSE) with sequentialization of WTs' conflicting accesses to the same LP state.

By the plots we see how our NBSE proposal is definitely resilient to performance degradation while increasing the number of used WTs. In fact, its speedup does not decrease for larger thread counts except for the scenario where the hot spots are hit with probability one. In any case, even for such extremely skewed workload of events, the decrease of the speedup when moving from 24 to 32 WTs is minor. Instead, the SLSE configuration shows a worse scalability, which leads performance to definitely decrease, especially when considering higher values of the probability to hit the hot-spot LPs with newly generated events.

The peek speedup achieved by NBSE is around 15 for all the configurations, while for largely skewed accesses SLSE does not provide more than 10 as the speedup. This is a relevant achievement of NBSE, showing not only scalability, but rather the capability to maintain similar scalability levels independently of the actual pattern of events (more or less clustered) across the LPs.

To further support the effectiveness of NBSE, we report results related to a few additional variations of the PHOLD configuration. In a first variation, we reduce the event granularity from 60 to 40 microseconds. In principle, these settings should be favorable to SLSE since the ratio between the time spent in non-blocking event-pool operations and the one spent processing events is increased. Hence, SLSE should suffer a bit less from the need for executing sequentialized accesses to the LPs' states while processing events. In any case, by the results in Figures 6.7-6.8 we see how NBSE still stands as definitely more performing than SLSE. In fact, NBSE still guarantees speedup of the order 15, while SLSE does not offer more than 10 of

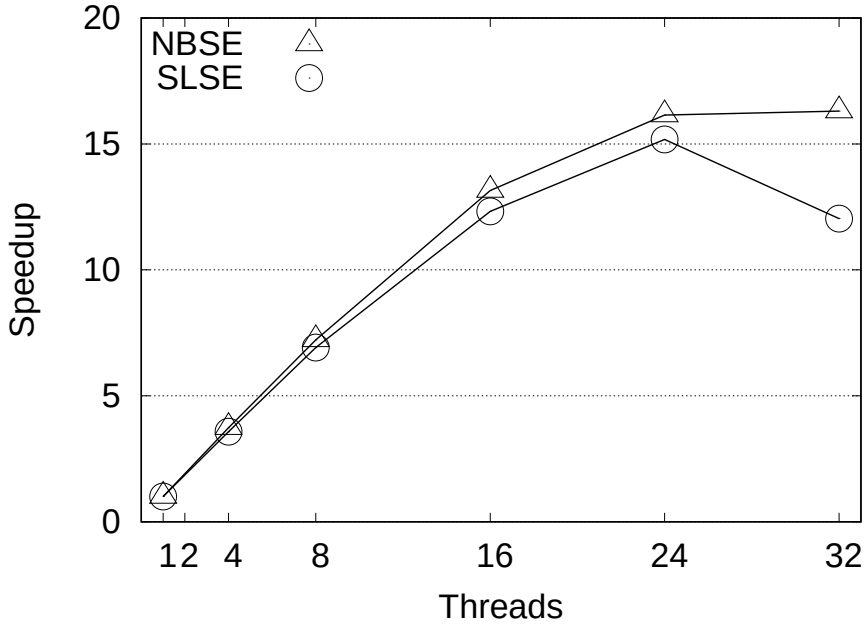

Figure 6.3. PHOLD (hot spot) - speedup results, number of spots equal to 10, probability of hitting the spot equal to 0.25 and event granularity equal to 60 μs

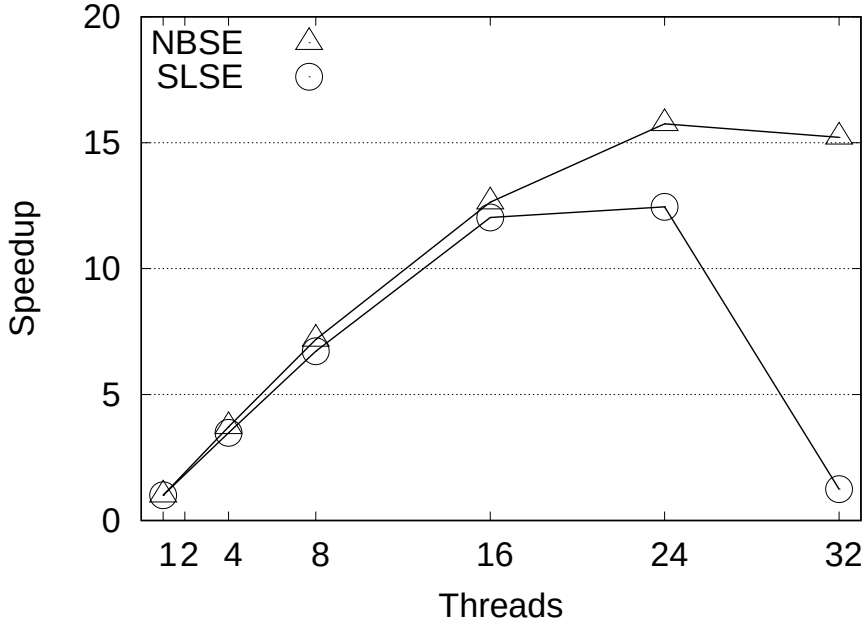

Figure 6.4. PHOLD (hot spot) speedup results, number of spots equal to 10, probability of hitting the spot equal to 0.5 and event granularity equal to 60 μs

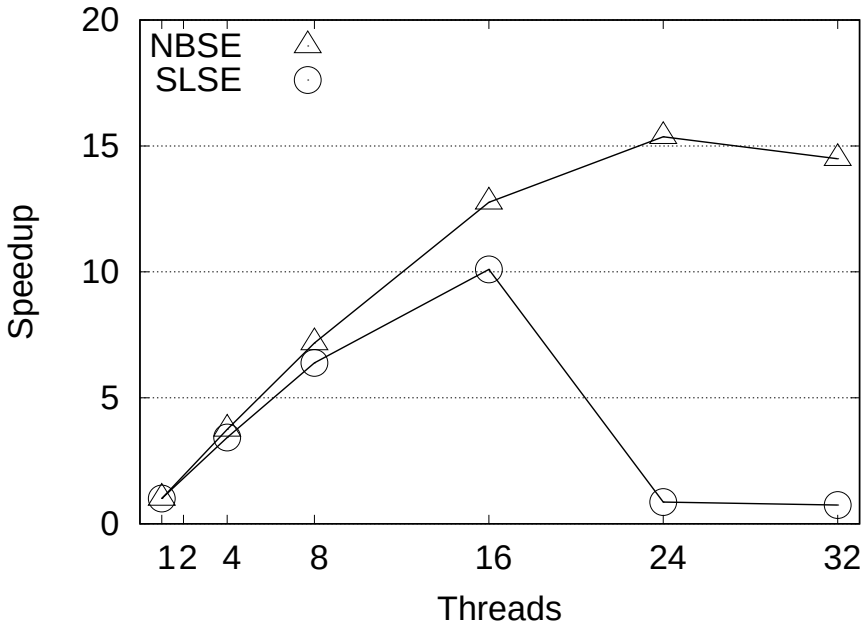

Figure 6.5. PHOLD (hot spot) speedup results, number of spots equal to 10, probability of hitting the spot equal to 0.75 and event granularity equal to 60 μs

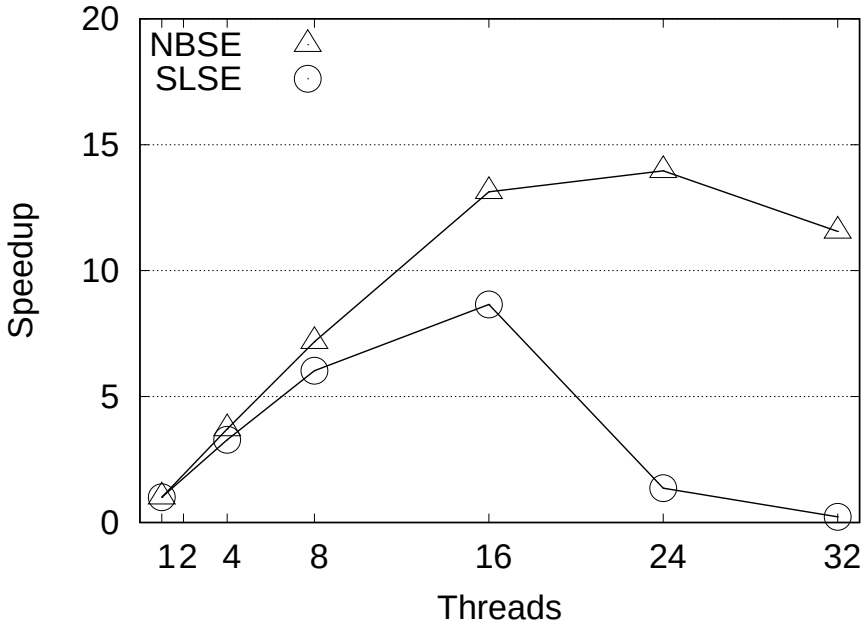

Figure 6.6. PHOLD (hot spot) speedup results, number of spots equal to 10, probability of hitting the spot equal to 1 and event granularity equal to 60 μs

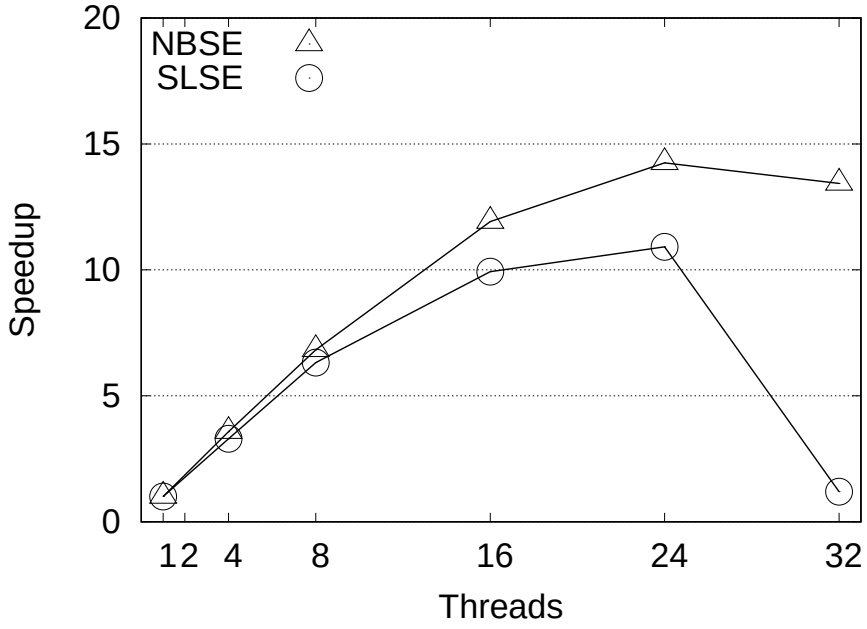

Figure 6.7. PHOLD (hot spot) speedup results, number of spots equal to 10, probability of hitting the spot equal to 0.5 and event granularity equal to 40 μs

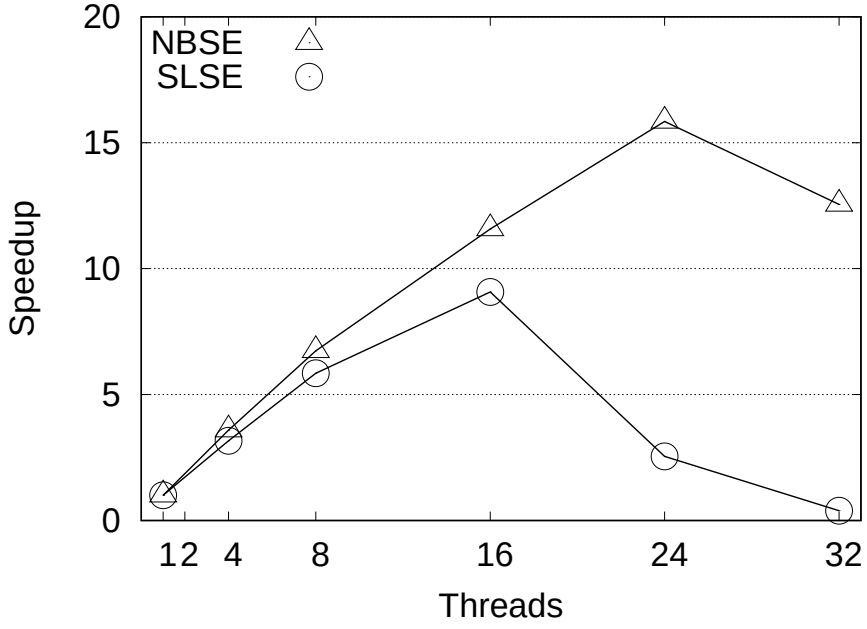

Figure 6.8. PHOLD (hot spot) speedup results, number of spots equal to 10, probability of hitting the spot equal to 0.75 and event granularity equal to 40 μs

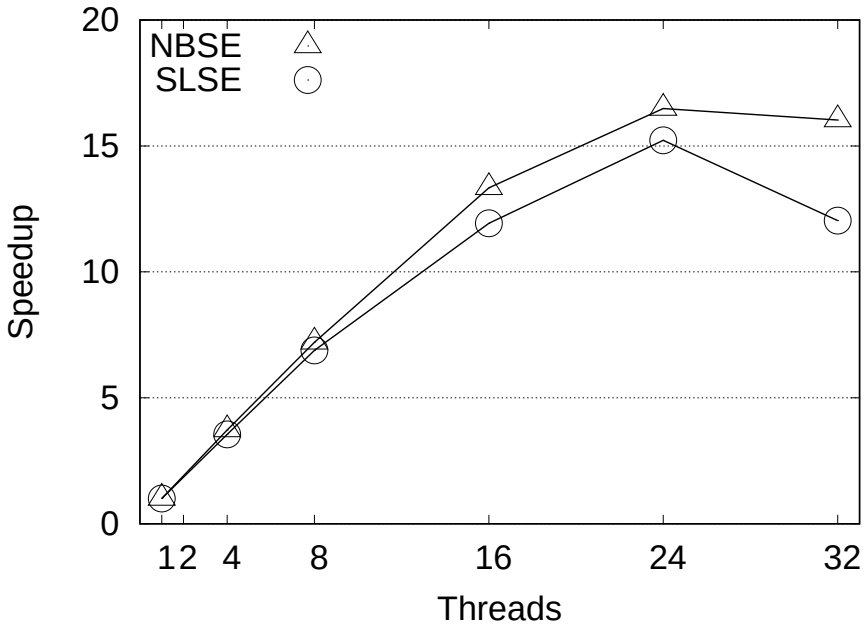

Figure 6.9. PHOLD (hot spot) speedup results, number of spots equal to 32, probability of hitting the spot equal to 0.5 and event granularity equal to 60 μs

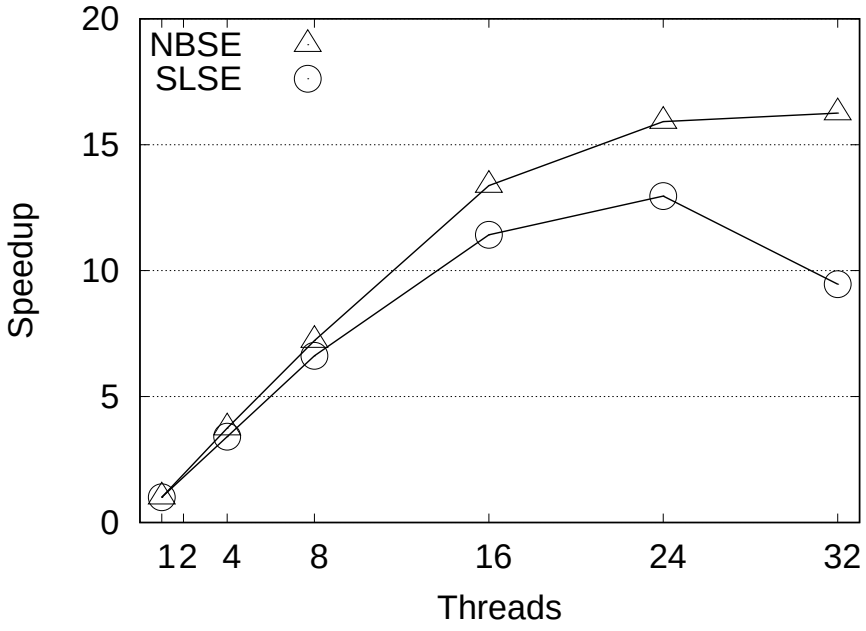

Figure 6.10. PHOLD (hot spot) speedup results, number of spots equal to 32, probability of hitting the spot equal to 1 and event granularity equal to 60 μs

speedup, exactly as with the previous configuration. These data refer to the cases of medium (0.5) and high (0.75) probability of event routing towards the hot spots.

The additional variation of PHOLD we consider is even more favorable to SLSE, since it is based on including 32 hot spots (which corresponds to the maximum number of WTs that are run within the PDES environment), rather than only 10. With these settings, the workload of events is less skewed, in terms of its distribution across the LPs, so that SLSE can find more opportunities of no-conflict between WTs that need to process events at the same destination LP. Hence, it should suffer less from lock-based sequentialization of the accesses to a same LP's state. The results for this variation of PHOLD are reported in Figures 6.9-6.10. Although the data show that SLSE suffers less from reduced scalability while increasing the number of WTs, the plots still show how NBSE is superior, especially with higher likelihood of hitting the spots. In fact, the maximum speedup achieved with NBSE is of the order of 16, while SLSE does not provide more than 12.5 speedup. Overall, also for this more favorable workload to SLSE, NBSE still achieves up to almost 30% better speedup values.

Chapter 7

The ultimate share-everything PDES system

As mentioned, an ideal share-everything PDES platform should guarantee scalability along the following two dimensions in a combined manner:

1) wall-clock-time coordination across threads—no one should block each other while accessing shared data structures (the event pool and the LP states);

2) virtual-time coordination across LPs (ultimately managed by threads)—no LP should be blocked along virtual time because of potential causality constraints between the simulation events.

While point 1) has been tackled in Chapter 5 and Chapter 6 providing non-blocking algorithms for managing the fully-shared event pool and share-everything-suited event-dispatching rules that avoid collisions across threads in the access to the state of the LPs, there is still a lack of support for fully non-blocking capabilities in virtual-time coordination. In fact, the provided solutions are able to speculate along simulation time for one event only (per LP), although lookahead is anyhow exploited for identifying the largest volume of tasks (events) that can be processed in parallel in non-blocking fashion.

In this chapter we present a design coping with both the above points in a combined manner providing fully speculative execution capabilities of the LPs—guaranteeing scalability in virtual-time coordination—and fully non-blocking wall-clock-time coordination across threads. While the solutions provided in the previous chapters are still valid, in this chapter these solutions are reshuffled and mixed together devising a holistic solution able to guarantee extreme efficiency under disparate workloads.

Technically, the proposal in this chapter provides solutions for a set of problems intrinsically related to the construction of speculative share-everything PDES systems. They are:

- the definition of non-blocking algorithms for managing a fully-shared pending-event pool that contains both schedule-committed events (those produced by

the execution of other events that have been detected to be safe and causally consistent) and non-committed ones (those that are the result of speculative, not yet committed, processing actions), which might need to be (logically) canceled. The latter aspect was not considered in the previously proposed solutions, which were limited to the management of schedule-committed events;

- the definition of non-blocking algorithms for dispatching the events to be processed across threads in such a way that threads never collide on a same LP, while detecting causal consistency along chains of speculatively processed events on-the-fly—also exploiting lookahead information. The latter aspect was not considered too in the previous solutions, where speculation did not allow to move LPs along arbitrary chains of possibly unsafe events.

Overall, we further raise the bar towards a fully featured solution targeting scalability of share-everything PDES on multi-core machines, whose implementation we again release as open source[1].

7.1 System architecture

When considering the level of speculation we admit, the solution we are going to present adheres to the well-known Time-Warp synchronization protocol [19]. More in detail, we support optimistic execution of (theoretically unbounded) chains of events along virtual time. To recover from an out-of-order event execution, this time we adopt the coasting forward [28, 106] technique based on state saving.

In this new design, the system relies on two main data structures:

(A) a set of *LP Control Blocks* (LPCBs), which are used to keep metadata representing the system-level view of the advancement of the LP in simulation time—this is a concept disjoint from the actual application level state of the LP;

(B) a set of events—either already processed, or to be processed, or logically canceled—maintained into the aforementioned fully-shared event pool which, from now on, we will refer to as *Scheduling Queue* (SQ).

At first approximation, the main execution loop carried out by all the WTs consists in: i) fetching some event to be processed from the SQ; ii) performing a rollback of the target LP, if required; iii) executing the event; iv) updating the LPCB; v) inserting newly generated events into the SQ.

As for point i), the fetch operation is contextual to the try-lock of the target LP. Hence, no WT will ever fetch an event bound to an LP that is currently locked by some other WT. Overall, no mutual block among WTs will ever occur in the attempt to access the same LP, since the WT that will experience a failure of its try-lock operation will simply go ahead scanning the event pool in order to take an event destined to some other LP, exactly in the same manner as seen in Chapter 6. This also guarantees isolation of WTs' accesses to a given LPCB and to the corresponding LP state, in both forward and rollback mode.

[1] Source code available at `https://github.com/HPDCS/USE`

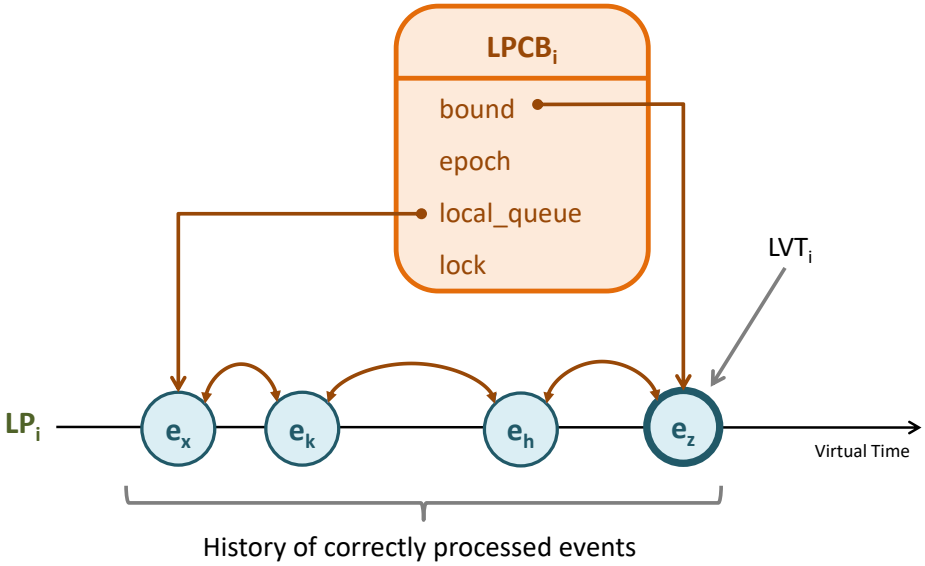

Figure 7.1. LPCB organization

Concerning the other operations accessing the SQ within the main loop, as we will discuss, they are all carried out in a non-blocking fashion—including secondary updates on individual nodes' data in order to correctly represent their state (e.g., logically cancelled because of a rollback) within the speculative processing scheme, as discussed in the following.

7.1.1 Architectural details

7.1.1.1 LP Control Blocks

Each LPCB is formed by a set of variables which hold metadata needed by the simulation engine to detect any relevant runtime condition related to the LP, including its involvement in causality errors. Its representation is shown in Figure 7.1. Noteworthy, among others, the LPCB keeps a pointer named `bound` to the last processed event, whose timestamp represents therefore the Local Virtual Time (LVT) of the LP associated with the LP. The actual buffer keeping the event pointed by `bound` still stands in the SQ, so that event processing leads to no actual removal from that queue. Un-linkage from this queue will occur when we detect that the event is either definitively causally consistent—it is a committed event—or it should not appear along the execution—it is an event generated by a rolled back one. Discriminating whether the event has been processed, or it is in a different state with respect to the state of the target LP and of all the other LPs, will take place via a proper state machine coded into the event-buffer metadata. The state diagram for this state machine will be discussed shortly.

The LPCB also maintains a non-negative integer named `epoch` which keeps track of the total number of rollback operations the LP has experienced. This information essentially tells in what incarnation the LP is currently executing along

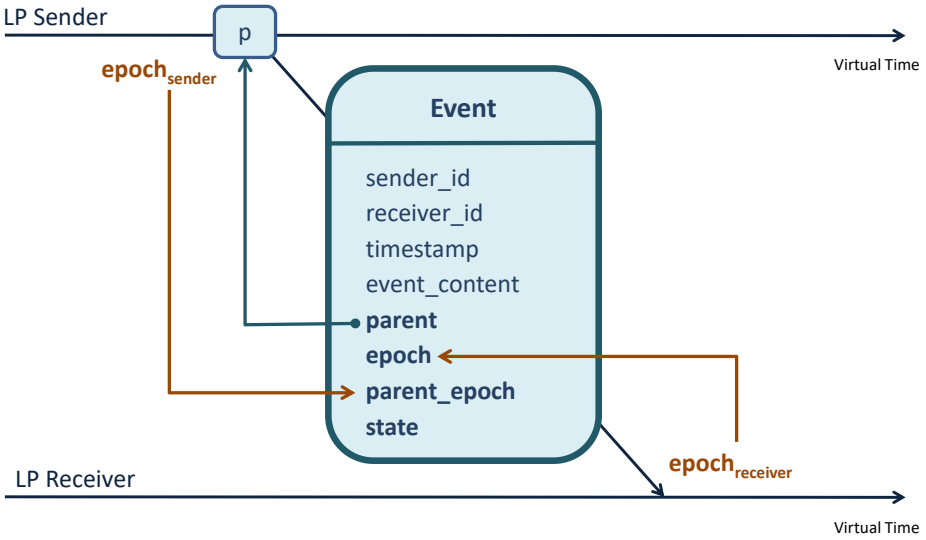

Figure 7.2. LPCB structure organization

its forward path, given that a rollback leads to a new incarnation—a new LP life after the causality error.

Moreover, each LPCB keeps metadata to retrieve (and to access) what we call the `local_queue` of the LP, which is used to maintain the history of processed events at that LP. Those that are rolled back are not included in this queue. Also, `local_queue` is built in our system as a view of event-buffers associated with the LP which are anyhow kept within the SQ. In other words, `local_queue` is built by relying on cross event-buffers' linkage standing aside of their linkage into the SQ. The reason for having such a view, rather than only relying on the global view of events in the SQ, stands in the management of both state reconstruction—which in our system is based on checkpointing and coasting forward—and CPU-dispatching of the next-to-be-processed event of the LP. Finally, the LPCB keeps the lock actually used to support try-lock operations when WTs try to take on the job of working on the LP.

7.1.1.2 Events representation

In our PDES system, an event is a simple memory buffer—the event-buffer—exchanged between two LPs, or sent from some LP to itself. The actual exchange takes place through insertion and extraction operations to/from the SQ. Each event originates on one LP, called *sender*, and targets another LP, called *receiver*, which could be the same event's source LP. The actual event representation is depicted in Figure 7.2.

Each event-buffer is made up by (a) metadata—used by the PDES system for treating the event—and by (b) the actual payload convoying the information to be delivered to the event-handler for application level processing. In this section we focus our attention on metadata, given that our system is application agnostic, and can support generic simulation models.

An event-buffer associated with the event e keeps the following metadata:

- the id of the LP which generates and sends the event, and the id of the LP which must receive and execute it, respectively hold by `sender` and `receiver` fields;
- the timestamp `ts` at which the event must occur along simulation time;
- a field `epoch`, which maintains the epoch of the receiver LP at the time when it processed the event;
- a pointer `parent` to the event p, whose execution has generated e;
- a field `parentEpoch`, which represents sender LP's epoch when p has been processed, namely the incarnation number of the sender LP at the generation time of e;
- a variable `state` used to represent the current state of the event within a finite-state machine, which drives the event management logic at the level of the PDES system.

Since our system entails speculative execution capabilities, possible violations of timestamp order might occur, and rollback operations are required in order to restore the correct execution trajectory of an LP (a timeline), as well as to undo the production of new events along the incorrect trajectory. In general, an event experiences several life stages. We define as *committed* the simulation trajectory of an LP that is observable at the end of a concurrent execution entailing no timestamp order violation. Every event that is visible within that simulation trajectory, is defined as *committed* or *safe*. On the contrary, events could be *retracted*, meaning that they can no longer exist in any time-line.

An event being processed flushes newly produced events to the SQ before the execution phase is over. Therefore, the unprocessed (or being-processed) event with the minimum timestamp is a safe event that corresponds to the commit horizon, namely the Global Virtual Time (GVT) of the speculative run.

Given that, thanks to the try-lock mechanism, two events destined to the same LP cannot be concurrently processed by WTs, we can exploit the lookahead of the simulation model to compute safety of whatever event to be processed (or being processed) according to the following expression:

$$\begin{aligned} is\_safe(e) \quad = \quad & (e.\texttt{ts} \in [GVT, GVT + LOOKAHEAD) \ \wedge \\ & \nexists\, e' : \ e.\texttt{lp} = e'.\texttt{lp} \ \wedge \ e'.\texttt{ts} < e.\texttt{ts}) \end{aligned} \tag{7.1}$$

If an event e is not safe—meaning that Equation 7.1 does not currently hold for it—and gets speculatively processed, its execution might be undone because of the arrival of a straggler destined to the same LP. However, in such a scenario, the event e could be still *valid*, meaning that it is requested to appear as executed along some timeline of the destination LP. On the other hand, if the event was generated by some other event that is undone, the former becomes *invalid*. In fact, it should no

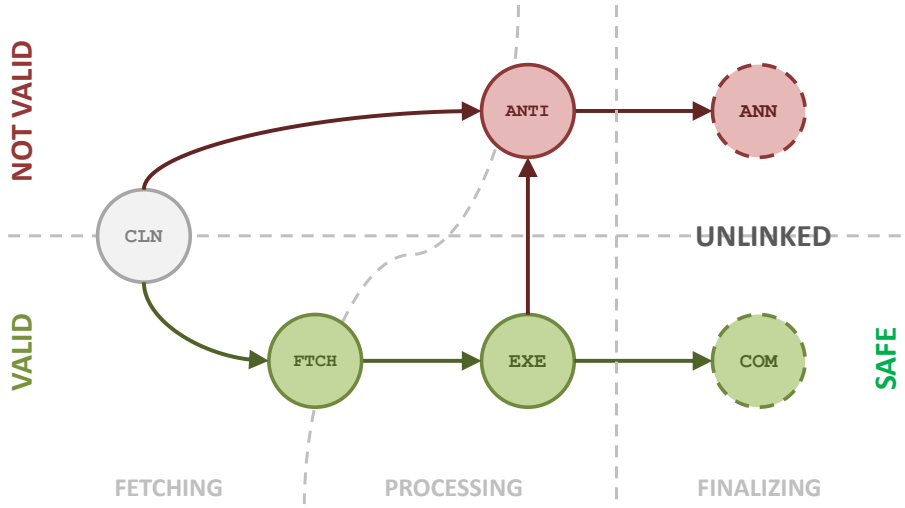

Figure 7.3. State diagram for the event's life-cycle

way appear in the correct timeline of the destination LP—in classical Time Warp these are events canceled by their corresponding anti-events.

As for safety, we detect if a particular event e has become invalid at a given point in wall-clock time by exploiting event-buffer metadata. In particular, an event e is currently valid if and only if (i) its parent p is currently valid as well and (ii) the execution of p that generated e has not been undone. Exploiting event metadata, the definition of validity can be formalized by the following recursive function:

$$is\_valid(e) = \begin{cases} true, & \text{if } e.\texttt{type} = \texttt{INIT} \\ (e.\texttt{parentEpoch} = e.\texttt{parent.epoch}) \wedge & \\ is\_valid(e.\texttt{parent})], & \text{otherwise} \end{cases} \tag{7.2}$$

The above formalization is based on having the validity of an event always depending on the validity of its parent. The unique exception is the `INIT` event —used to just setup the simulation initial state, including the states of the LPs— which is safe (hence valid) by construction.

To check whether the parent of an event e has been re-executed or not, we harness the `parentEpoch` information kept by the event-buffer, which is compared to the epoch of its parent p. As said before, e.`parentEpoch` and p.`epoch` have been set with the epoch of the sender LP at the time the event p has been executed. Since the LPs' epochs are updated after a rollback takes place, if an event is re-processed, its epoch number will be updated with the new LP's epoch. If p.`epoch` is not equal to—more precisely greater than—e.`epoch`, it means that the parent is living within a new and different timeline, to which the child event e does no longer belong. Consequently, if e.`parentEpoch` $= p$.`epoch` we can infer that the event p has not been re-executed, and therefore still stands on the original timeline that generated e.

We do not immediately unlink events from the SQ when they become invalid. This is because we manage the queue via non-blocking algorithms. As a conse-

quence, a node in the queue—namely, an event-buffer—might be required to still stand into the queue to facilitate the execution of non-blocking queue traversals by WTs. In fact, they can use that node as a link between others, even though the corresponding event appears to be as no longer relevant for the execution of any LPs' timeline. Also, temporarily keeping invalid event-buffers into the SQ is a way to asynchronously notify the other WTs currently traversing the SQ that something has changed along the timeline of some LP—since invalid event-buffers expose updated metadata—which may in its turn drive the actions by these same WTs. In other words, each single node in the SQ is associated with a state machine that helps supporting a fine grain coordination across WTs, implemented according to the non-blocking paradigm.

Figure 7.3 shows the actual state machine within which each event-buffer lives. How state transitions occur based in the pseudo-code executed by WTs will be discussed in Section 7.1.2, while in this section we illustrate the "meaning" of each state. A newly produced event, just inserted into the SQ, is born with a clean state (`CLN`) representing that no operation has been performed yet on the event-buffer—except its insertion, clearly with the correct metadata such as the epoch of the sender LP. Note however that, starting from the wall-clock-time instant the event appears to be incorporated into the queue, any WT can be already traversing it, possibly updating its state. When a WT finds an event-buffer e marked with the `CLN` state in the SQ, it logically extracts (fetches) e, by marking it as `FTCH`—we recall that for this to occur, the WT must have observed the presence of the event-buffer and must have successfully executed the try-lock on the target LP. In fact, two different WTs cannot take on the job of concurrently processing a same event destined to a given LP. Marking an event-buffer as `FTCH` leads to notify that the event is currently in charge to some WT.

It is possible that, when the WT successfully try-locks the LP associated with an event-buffer in the `CLN` state, this event has already become invalid, since the execution of its parent might have been undone or invalidated. When a WT observes this condition while traversing the SQ, it makes the event transit to the logical `ANTI` state, whose name evokes the anti-event concept—meaning that it simply does no longer belong to the simulation trajectory, along any timeline. All the state transitions are implemented atomically, given that they can be carried out by concurrent WTs. For potentially conflicting (mutually excluding) transitions—such as `CLN`→ `FTCH` and `CLN`→ `ANTI`—we rely on the `Compare&Swap (CAS)` machine instruction. For non-conflicting ones, we simply use atomic memory writes—such as `AtomicExchange`—abstracted by the `SET` statement in the pseudo-code. Therefore, the transition to `ANTI` excludes the possibility for a WT to successfully transit the node to `FTCH`. It is clear that an event still complying with the validity rules expressed by Equation 7.2, might really be no longer valid. It is only a matter of time for the WTs to detect that such information related to validity is reflected into the state of the parent event. On the other hand, speculative processing is already known to accept the risk of processing something that seems to be consistent, in terms of timestamp ordering, but which is actually no longer consistent given that something is happening concurrently along model execution.

An event successfully marked as `FTCH` is returned to the main loop of the WT, where it will be processed, as we shall describe. Here, the event transits to the

"execution" state, say `EXC`, meaning that the event-handler actually took it for performing the corresponding LP state manipulations.

Overall, the `ANTI` state, still reachable from the `EXC` state, is used to discriminate that the PDES system knows that the event does no longer belong to any valid LP timeline, either if it has been already fetched and executed by some thread—thus it passed through `EXC`—or if it is found to be invalid prior being fetched. The `ANTI` state is reached via a transition from `EXC` when a rollback occurs related to the passage of the event to the invalid state. Therefore, it will not need to be re-processed after the rollback. The PDES system can detect that a rollback needs to be executed when a WT traverses the SQ comparing the metadata of the LP and those kept by the event-buffer (such as the current LVT of the LP and the event timestamp, or its parent's state). These checks are anyhow executed without the need for locking the target LP. If the event marked as `ANTI` is found to stand in the future—or on a new timeline after a rollback of the target LP—the event is simply logically transitioned to the `ANN` state, an absorbing state leading to the unlink of the event from the SQ.

Similarly, when an event ends its life-cycle and appears along the correct LP timeline, it is logically marked as `COM` (commit state). Clearly, an event transits to `COM` after being processed if it is found to be a safe one. In this case it is also unlinked from the SQ. Although unlinked from the SQ, an event-buffer in the `COM` state will be garbage collected (reused) successively, as we shall discuss, since some child could still refer to it for validity assessment according to Equation 7.2. Another motivation for retaining the event is its usefulness for state reconstruction purposes in a rollback phase, as we will also discuss.

As a final note, an event can persist in the `EXC` state across multiple executions, caused by rollbacks, up to the point in time when its safety is assessed, or it becomes invalid.

We want to note that, unlike other implementations of simulation platforms that embrace the Time Warp protocol, we do not rely on the notion of an "anti-message" as a separate platform-side event entity which conveys data used to annihilate another event; rather we embed this type of information within the event itself which become an anti-event for itself, as soon as it becomes invalid. Boiling from this fact, we do not need to exchange extra messages nor process extra event-buffers to the only purpose of canceling the effects done by others.

7.1.1.3 Scheduling Queue

The Scheduling Queue (SQ) used in our share-everything PDES system is a conflict-resilient lock-free priority queue that sorts event-buffers (across all the LPs) on the basis of their timestamps. In particular, it is a calendar queue supporting non-blocking operations. We borrow its implementation from Chapter 5, reshuffling it in order to meet the needs of this further improved share-everything PDES system. At a logical level, such a queue can be abstracted as a generic non-blocking ordered linked-list like the one proposed by Harris [96]—although being much more efficient thanks to its multi-bucket organization leading to amortized constant-time access. Relying on this abstraction allows us to hide the complexity of non-blocking calendar queue operations, which are not the focus of this chapter—jointly enabling us to

focus on how we exploit non-blocking capabilities of such priority queue in our *ultimate* share-everything PDES system.

In a way similar to what we have introduced in Chapter 6, the reshuffled priority queue has an Enqueue API and two other primitives: GetMin, which retrieves a pointer to the event with the smallest timestamp which is still linked to the queue, and GetNext, which retrieves the pointer to the event which immediately follows—along virtual time—the one identified by its input argument. Thanks to this support we can build a cross-layer optimized Fetch operation that returns in a non-blocking mode a to-be-processed event associated with some LP not currently locked by any WT (see Section 7.1.2.2 for the details). Further, the SQ supports an Unlink API which is used to disconnect a generic event from the SQ—those that transit to the `ANN` or `COM` state—still in non-blocking fashion.

Before returning an event e, a WT executing a Fetch executes a try-lock operation on the target LP. If this operation fails, the WT slides to the subsequent event in the queue—the one successive to e. This is done thanks to the exploitation of the above mentioned "get" services in the queue API. This sliding scheme is iterated up to the point where a try-lock on the LP targeted by some event, encountered along the queue, executes successfully.

According to the event's state diagram in Figure 7.3, fetched (or already processed) events are not unlinked from the SQ. In fact, an event could be re-executed due to a rollback. Hence, it must be visible in future queue explorations by concurrent WTs in order to be properly handled. Moreover, our event validity definition (see Equation 7.2) implies that parent metadata could be accessed while assessing the state of its children. Therefore, the actual garbage collection of the event-buffer—leading to its reuse—is also determined by the relation between the GVT value and the timestamp of child events, as we shall discuss. On the other hand, the logical removal of the node in the `COM` state associated with the minimum timestamp value from the SQ via the Unlink API is enough to move forward—beyond that node—the pointer to the new minimum timestamp element into the queue. As said, this is because in our PDES system the removal of that `COM` event, which was previously processed, already led to incorporate into the SQ all its children, if any.

7.1.2 Worker thread algorithm

7.1.2.1 Main loop

The pseudocode of the main loop carried out by any WT is shown in Algorithm 7.1. Initially a call to the Fetch procedure is executed to retrieve from the SQ an event to be handled (processed or undone/retracted), which is destined to an LP not currently in charge of another WT. The Fetch procedure returns to the caller WT a pointer to the event to be handled, and the indication of whether the event is safe (according to Equation 7.1) or at least valid. The Fetch procedure also returns the minimum timestamp of the non-committed event standing into the SQ, namely the current GVT value. Further, according to state transitions, such procedure may lead events destined to whatever LP to transit from `CLN` to `ANTI` while traversing the SQ. This is based on validity checks as expressed by Equation 7.2.

Algorithm 7.1 Main Loop

```
M1:  procedure MainLoop()
M2:      < evt, safe, valid, gvt > ← Fetch()
M3:      curLP ← evt.receiver
M4:      if evt ≠ null then
M5:          if evt.ts < curLP.bound.ts then
M6:              Rollback(curLP, evt.ts)
M7:          end if
M8:          if ¬valid then
M9:              SET(evt.state← ANTI)
M10:         else
M11:             MakeUndoable(curLP,evt,safe)
M12:             LinkToLPQueue(curLP.bound, evt)
M13:             evt.epoch ← curLP.epoch
M14:             newEvts ← Execute(evt)
M15:             Flush()
M16:             curLP.bound ← evt
M17:             SET(evt.state, EXC)
M18:         end if
M19:         Unlock(curLP.lock)
M20:         if safe = TRUE then
M21:             Unlink(evt)
M22:         end if
M23:     end if
M24:     GVTOperations(gvt)
M25: end procedure
```

Regardless the retrieved event's current state—namely, if it has to be executed as an event or if it is an invalid one to be retracted—the thread checks if its timestamp is smaller than the LP's LVT, namely the time reached by the LP executing the event pointed by its `bound` variable in the LPCB. In the positive case a Rollback is triggered in order to bring back the LP's state to the timestamp of the last event preceding the retrieved straggler event and belonging to the current timeline. At the end of the Rollback procedure the simulation state is compliant with respect to the execution of the currently retrieved event. If this event was already marked as `EXC`, it simply persists into this state. Otherwise, it will transit from `FTCH` to `EXC` at the end of its execution.

As mentioned, the Fetch procedure returns the information about the validity of the event, namely if it has to be executed or not. In the second case, corresponding to an event to be retracted, it is atomically marked as `ANTI` by the in charge WT and the loop current iteration is completed by unlocking the current LP. Otherwise, it has to be actually executed. If it is a straggler, correct alignment to the LP timeline has already been performed by the aforementioned rollback operation, so it can be normally processed.

Before the execution of the event, a MakeUndoable operation is called, which implements whatever policy for making the current state transition undoable (i.e.,

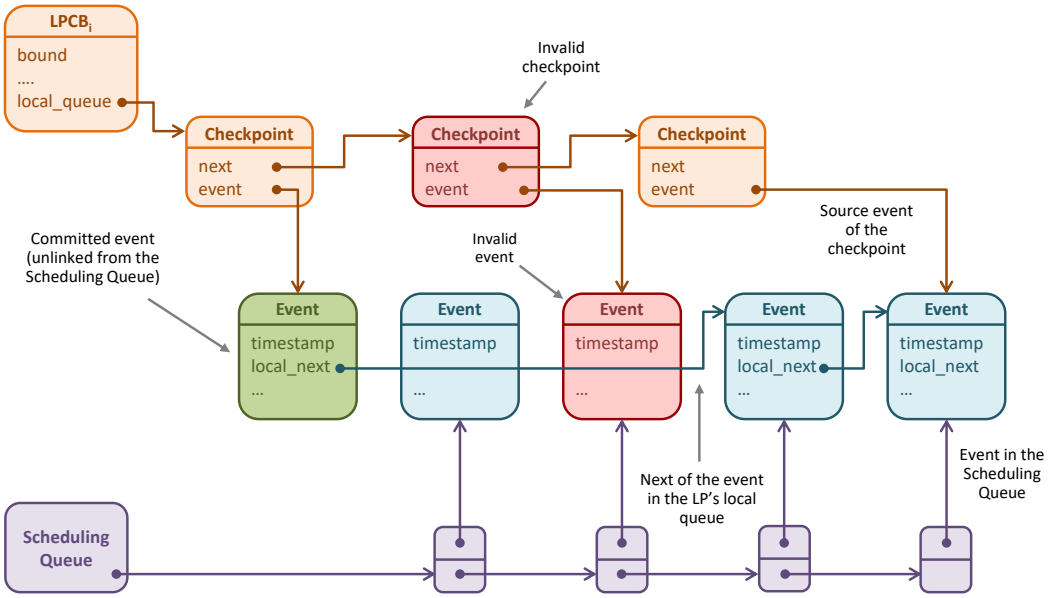

Figure 7.4. Data structures of LP state, SQ and events, and relative relationships

rollbackable). In our implementation we opted for a traditional periodic checkpointing approach. Based on the selected policy, if the log is taken, the log-node is linked to the corresponding event (the one pointed to by `bound`), for correct alignment of the data structures. We note that the safety information associated with the event retrieved via Fetch can be exploited for optimizing the management of the undo support. In our case, it could lead to take a checkpoint and to rejuvenate the checkpoint period if the event will never be undone—this will enable reducing coasting forward overhead by inserting a log exactly on the currently committed event of the LP. On the other hand, if reverse computing approaches were employed (see [42, 43]), the safety information would simply tell that there will be no need to generate data/metadata for reversing the event execution. The relation among the different data structures we used to manage each individual LP is schematized in Figure 7.4. By the scheme we see how the event queue of the LP—namely `local_queue`—is incorporated into the SQ—which is global to all the LPs. Thus, the LP local queue represents a sort of view on such global queue achieved via a parallel linked-list. The processed event is linked to the per-LP view of the SQ via LinkToLPQueue, so as to make it immediately available for any purpose, including coasting forward if requested.

The real event-processing phase starts by updating the event epoch with the LP's current one, in order to represent its current incarnation within the LP's current timeline. Then, the event is executed by invoking the application-level event-handler provided by the simulation model. New events possibly produced along the event-handler execution are stored in a local buffer, in order to Flush them to the SQ at the end of the current event execution. At this point, to complete event processing, the state of the event is atomically set to `EXC` and the LP bound is updated in order to point to the current event.

The order of those operations is fundamental to guarantee correctness. In fact, newly produced events are flushed into the SQ before their parent's processing is finalized, otherwise we would violate the definition of GVT we rely on. So, it is important to postpone the update of the current `bound` and of the state of the event after the flush operation.

Clearly, once updated the event state, the WT releases the lock on the target LP, thus enabling concurrent WTs to eventually take care of whatever event destined to the same LP, as in the spirit of the share-everything paradigm. Finally, if the Fetch procedure indicates that the event is a safe one, the final logical transition to the `COM` state is directly executed by the in-charge WT, which unlinks the event from the SQ.

At the end of the main loop, despite what happened before, some housekeeping tasks are performed, which take place via the GVTOperations procedure. Specifically, this procedure is used to reclaim memory associated with no longer useful event-buffers, as well as checkpoints. As we will clarify while explaining the Fetch procedure, event-buffers are initially unlinked from the SQ when they will no longer need to be handled, which means they are committed or have been annihilated, transiting respectively in `COM` and `ANN` states. On the other hand, the event-buffers in the `COM` state still remain into the per-LP `local_queue`, while the event-buffers in the `ANN` state are inserted (upon their unlink from the SQ) into per-WT retirement queues. All these event-buffers can be actually reused (reclaimed by the memory system) only after the GVT value oversteps the maximum timestamp of any child possibly generated by their execution. This is because validity of an event-buffer in our system is based on referring the state of a parent event-buffer according to Equation 7.2. To easily keep track of such condition at runtime, each event-buffer is filled with a timestamp field which exactly keeps such a maximum child-timestamp. While executing the GVTOperations procedure, the WT deallocates all the event-buffers from its list of `ANN` nodes which satisfy such condition. The same occurs for the nodes in the `local_queue` of the LP that the WT is currently in charge of.

7.1.2.2 Fetch procedure

The Fetch procedure, whose pseudocode is shown in Algorithm 7.2, has two objectives: (i) returning an event to be processed by the WT main loop; (ii) unlinking no-longer needed events from the SQ and hence from the per-LP views of this queue. This procedure uses two local variables gvt and $jmpLP$. The former keeps the timestamp of the minimum event still linked to the SQ, the latter keeps the set of LPs targeted by "skipped" events—the events that have been traversed by the WT without actually locking the target LP.

First, the Fetch procedure retrieves the current minimum from the SQ by invoking GetMin and storing its timestamp into gvt. Also, it initializes $jmpLP$ as an empty set. Then, for each traversed event evt a safety check is performed—so we check if $evt.\texttt{ts} \in [gvt, gvt + LA)$ and $LP \notin jmpLP$, where LP is the receiver of evt (this is the implementation of the check on the safety condition expresses in Equation 7.1, a scheme of which is depicted in Figure 7.5). Moreover, we check if evt is in the past of the LP timeline by comparing the timestamps associated with evt and the LP `bound`.

Algorithm 7.2 Fetch procedure

```
F1:  procedure Fetch()
F2:      evt ← GetMin()
F3:      gvt ← evt.ts
F4:      jmpLPs ← {}
F5:      while evt ≠ null do
F6:          LP ← evt.receiver
F7:          evtState ← evt.state
F8:          safe ← is_safe(evt,gvt,jmpLP)
F9:          in_past ← evt.ts ≤ LP.bound.ts
F10:         valid ← is_valid(evt)
F11:         if TryCleanAndSkip(evt,jmpLPs,LP,safe,in_past, valid) then
F12:             if TryLock(LP.lock) then
F13:                 curr ← GetLocalNextAndValid(evt)
F14:                 if curr ≠ evt then
F15:                     valid ← true
F16:                     safe ← is_safe(curr,gvt,jmpLP)
F17:                 end if
F18:                 in_past ← evt.ts ≤ LP.bound.ts
F19:                 if ¬valid then
F20:                     if in_past then
F21:                         return ⟨evt, safe, gvt⟩
F22:                     end if
F23:                     SET(evt.state← ANTI)
F24:                 end if
F25:                 evtState ← CAS(evt.state, CLN, FTCH)
F26:                 switch(evtState)
F27:                     case CLN:
F28:                         return ⟨evt, safe, valid, gvt⟩
F29:                     case EXC:
F30:                         if ¬in_past
F31:                             return ⟨evt, safe, valid, gvt⟩
F32:                         else if ¬safe
F33:                             jmpLPs ← jmpLPs ∪ {LP}
F34:                         break
F35:                 ReleaseLock(LP.lock)
F36:             else
F37:                 jmpLP ← jmpLP ∪ {LP}
F38:             end if
F39:         end if
F40:         evt ← GetNext(evt)
F41:     end while
F42: end procedure
```

Algorithm 7.3 TryCleanAndSkip procedure

```
T1:  procedure TryCleanAndSkip(evt,jmpLPs,LP,safe,in_past, valid)
T2:      tryLock ← true
T3:      if valid then
T4:          if in_past ∧ evtState = EXC then
T5:              if safe then
T6:                  Unlink(evt)
T7:              else
T8:                  jmpLPs ← jmpLPs ∪ {LP}
T9:              end if
T10:             tryLock ← false
T11:         end if
T12:     else
T13:         if evtState = CLN then
T14:             evtState ← CAS(evt.state, CLN, ANTI)
T15:         end if
T16:         if evtState = ANTI then
T17:             Unlink(evt)
T18:             tryLock ← false
T19:         end if
T20:     end if
T21:     return tryLock
T22: end procedure
```

Algorithm 7.4 GetLocalNextAndValid procedure

```
V1:  procedure GetLocalNextAndValid(curr)
V2:      LP ← curr.receiver
V3:      lNext ← GetLocalNext(LP.bound)
V4:      while lNext ≠ null ∧ ¬is_valid(lNext) do
V5:          SET(evt.state← ANTI)
V6:          Unlink(evt)
V7:          lNext ← GetLocalNext(LP.bound)
V8:      end while
V9:      if lNext ≠ null ∧ lNext.ts < evt.ts then
V10:         return lNext
V11:     end if
V12:     return curr
V13: end procedure
```

Then, TryCleanAndSkip (Algorithm 7.3) is executed. This is a non-blocking procedure aimed at unlinking from the SQ events that have expired their lifetime (they are into an absorbing state) or telling whether the current event has to be processed, by returning a boolean value set to true. The former case corresponds to valid and executed events that can be considered committed since they are found to be *safe*, or to not-valid events that can be safely deleted since they not appear

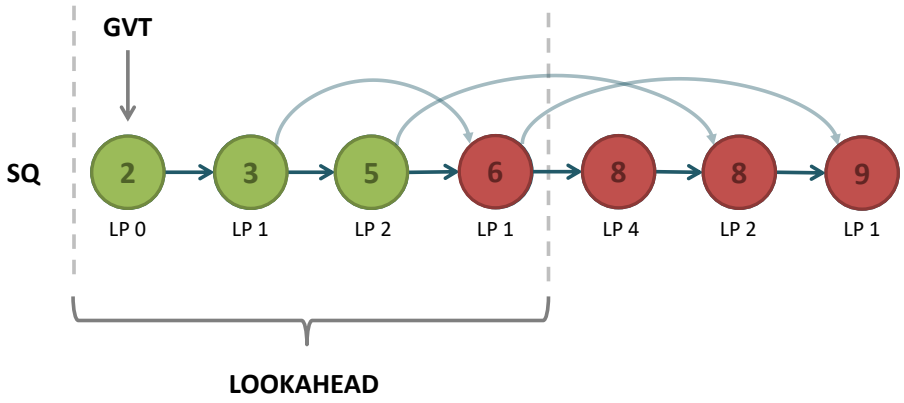

Figure 7.5. Safely-executable/to-be-committed events

in the current simulation trajectory (not jet executed or in the future respect the LVT). Instead, the latter case is associated with any event, which is not in absorbing states, and requires the target LP to execute it. In this category are classified all the valid events still to be executed—both ones in the future or in the past but in a `CLN` state—and the not-valid ones that are still to be annihilated—namely, the not-valid event placed in the past respect the LVT and in an `EXC` state. Finally, for all the execyted events that are waiting to be committed, the corresponding LP identifier is placed inside the $jmpLP$ set used to check the safety of farther events. All these checks are carried out by the TryCleanAndSkip procedure relying on the metadata we included in our event-buffers and in the LPCBs.

If TryCleanAndSkip returns true, then the WT try-locks the target LP. If this fails, then the WT skips to the subsequent event into the SQ, by relying on the GetNext API. This pattern is iterated up to the point where the WT successfully locks a target LP, or the tail of the SQ is reached—GetNext returns `null`. While traversing the SQ, the $jmpLP$ variable is populated, keeping track of all the LPs for which the WT observed something to be present into the SQ, until some target LP is locked. This set is used to compute the safety of an event.

When some target LP is successfully locked, evt is checked again to determine whether it is in the past of the LP. This is because in the wall-clock-time interval between the first check on evt performed by TryCleanAndSkip and the current processing phase, some other WT may have changed the `bound` of the target LP.

Then WT invokes GetLocalNextAndValid (Algorithm 7.4), that returns an event $lNext$ which is either the first valid event following the `bound` of LP into its local view of the SQ—the `local_queue`—or the event just fetched from the SQ. We need this procedure to check whether some event that is subsequent to the `bound` into `local_queue` has a timestamp lower than the one just fetched from the SQ, which represents a critical scenario to cope with. Such a scenario is illustrated via an example shown in Figure 7.6. Suppose that WT A holds a lock on an LP X because it is processing an event e. If another thread B tries to acquire the lock on X for processing an event f such that e precedes f ($e \rightarrow f$), it will fail and continue to analyze the next event g. This event is such that $e \rightarrow f \rightarrow g$ and targets X, so B will try to acquire again the lock on that LP. If in the meanwhile A has released

Wall-clock time	Thread A	Thread B
1:	`Check Event A on LP 1`	
2:	`Trylock on LP 1`	
3:	`Success`	`Check Event A on LP 1`
4:	`Process Event A`	`Trylock on LP 1`
5:	`.`	`Fail`
6:	`.`	`Check Event B on LP 1`
7:	`.`	`Trylock on LP 1`
8:	`.`	`Fail`
9:	`Release lock on LP 1`	`Check Event C on LP 1`
10:		`Trylock on LP 1`
11:		`Success`
12:		`Process Event C`

Figure 7.6. The scenario tackled by GetLocalNextAndValid.

the lock on X, B takes the lock and starts processing g. Supposing that e, f and g are all valid events, B executes g moving forward the bound of LP X. If f has a `CLN` state, executing g is not problematic since some WT eventually retrieves f and executes it after triggering a rollback as if it were a straggler event. Conversely, if f has an `EXC` state, no WT will ever execute such event again since it appears to be in the past of the current trajectory. This situation is prevented to occur exactly by the presence of GetLocalNextAndValid, that searches for an event (f) with higher priority than the currently fetched one (g) from the SQ. This is possible because, since the event f has been executed at least once, it is linked in the `local_queue` of the corresponding events and it is placed in some point after the actual bound, thus it can be found without traversing the SQ.

Then, if the GetLocalNextAndValid procedure returns a different element with respect to the one originally identified, the relative validity information is updated. Thus, the first performed check is about the event validity—note that this check is carried out outside the locking region of the target LP. Since in the TryCleanAndSkip procedure invalid and `CLN` events are correctly marked (as `ANTI`) and unlinked, here we can assume that if we met an invalid event-buffer, it is not a newly inserted one. In this case we have to perform a rollback if and only if it is in the past of the current incarnation (epoch) of LP, otherwise we atomically set its state to `ANTI`, in order to notify that the event no more needs to be processed—we remember that, since we have booked the corresponding LP, its `bound` is temporally fixed, thus event time position respect the LVT cannot change during these controls.

Otherwise, if the event was observed to be valid, WT tries to atomically apply the state transition `CLN`→`FTCH` with the `CAS` instruction. Of course, if it has been concurrently marked as `ANTI` by another WT that has seen the event as invalid, the `CAS` will fail.

After reading the actual state value of the event as a result of the `CAS` instruction[2], a switch case on it is carried out, implementing the events' finite-state machine discussed in Section 7.1.1.2. If the state is `CLN`, we know that the event has never been fetched and executed, thus, it is directly returned to the main loop. As discussed, it is up to the main loop to check if it is a straggler or not and trigger a rollback if required. The second case is the one where we have an `EXC` state, meaning that the event has been executed at least once. It follows that, it can be re-executed if and only if it is beyond the LP bound, meaning that it is in the future of the actual incarnation of the LP trajectory, namely the LP timeline. This is an event that has been rolled back and is still valid in the current (refreshed after the rollback) timeline. If the event is committable, namely if it is safe, we can unlink it from the global queue, otherwise we can skip the event by adding the LP to the $jmpLP$ set, releasing the lock and retrieving another event.

Once the switch case has been executed, we can release the lock on the target LP, since here the event state can only be set to `ANTI` or `EXC`. In any case, the event unlink operation, that transits the event to the `COM` or `ANN` state, will be performed by any WT in TryCleanAndSkip, thus completing the event life-cycle.

7.1.2.3 A further optimization

Let us consider a scenario where a worker thread WT_a has scheduled LP_x in order to process one of its events, while a worker thread WT_b is starting a fetching phase in order to retrieve an event to be processed. What can happen is that, during its traversal of the SQ WT_b tries multiple times to book LP_x—on different not-yet-processed events—failing each time due to the short-term binding of LP_x to WT_a. Now, let us consider that WT_a finishes its current processing loop releasing the lock on LP_x, which is now available to be booked again. At this point, according to the solution exposed in Algorithm 7.2, the next time that WT_b, which may still be performing the previous Fetch operation, finds an event destined to LP_x, it will try again to reserve the corresponding LP succeeding in its purpose. Differently by the critical run exposed in Table 7.6, here the skipped events destined to LP_x have not been processed, thus, the causality inversion will be noted by the next worker thread that, after WT_b, will extract an event—hopefully the one with the smallest timestamp—destined to LP_x. However, given that multiple to-be-processed events destined to LP_x have been traversed, the execution of the currently-picked one will lead of course to a rollback in the next scheduling of LP_x. In order to overcome such problem, we have introduced an additional mechanism in the Fetch procedure to never attempt again, during the same fetch operation, to retrieve an event destined to an LP found to be booked by another worker thread. An additional contribution provided by such mechanism is related to data locality. In fact, skipping events destined to LPs previously seen as booked increases the probability that events destined to the same LP are eventually executed by the same WT.

As a final note, this optimization could be exploited also within the simulation engine presented in Chapter 6, although we believe it can be more useful in an engine with higher speculation capabilities, just like the one we present in this chapter.

[2]As in most common implementations, we assume that `CAS` returns the original value of the targeted memory location independently of whether its update fails.

7.2 Experimental assessment

In this section we present performance results for a comparison of our Ultimate Share-Everything PDES system (USE) and the Share-Everything solution presented in the previous chapter (SE). We remember that the latter does not entail Time-Warp style processing of the events since, for each LP, at most one event is executed speculatively, and is eventually committed or aborted before any other event for the same LP can be CPU-dispatched. Its unique event pool—fully shared across WTs—only keeps so called *schedule-committed* events, which all need to appear along the LP timeline, thus not requiring the non-blocking management of any state machine for determining their actual role (across multiple ones) along model execution (e.g., if they need to be retracted because of the rollback of the parent). In other words, SE is blocking in virtual-time synchronization, while USE is fully non-blocking in both wall-clock-time thread coordination, and virtual-time LP synchronization, also thanks to its more sophisticated—and still non-blocking—logic for the management of the fully-shared event-pool data structure. For completeness of the analysis, we also include another competitor, which is the ROOT-Sim last generation traditional-PDES environment [59] not based on the share-everything paradigm—it adopts partitioning of the LPs across threads, with dynamic rebind for load-balancing purposes. It anyhow entails optimizations in its internal organization suited for shared-memory machines [107].

All the tests have been run on the same 32-core HP ProLiant machine exploited for previously presented experimental studies. We recall that it is equipped with four 2GHz AMD Opteron 6128 processors. Each processor has 8 physical cores that share a 12MB L3 cache (6 MB per each 4-cores set), and each CPU-core has a 512KB private L2 cache. The machine is equipped with 64 GB of RAM—organized in 8 NUMA nodes—and we used Linux (kernel 3.2) as operating system.

The number of WTs running within all the used PDES systems we are comparing has been varied from 1 to 32 in order to perform a scalability study. All the reported data points have been computed as the average over 10 runs, executed with different seeds for the pseudo-random generation of event timestamps.

7.2.1 Results with PHOLD

As first test-bed application we used the classical PHOLD benchmark [14] configured with 1024 LPs. Each LP schedules events for any other LP in the system, with an exponential timestamp increment. As already pointed out, for PHOLD, event processing leads to spending some CPU-time via a busy loop emulating a given event granularity. In our experiments we set the loop to give rise to events with granularity of the order of 60 or 120 microseconds, thus spanning between fine-to-mid values leading to a representative setting for testing parallel processing platforms—larger grain events might mask platform level costs for parallelization/coordination/rollback-support independently of the used PDES paradigm, share-everything vs traditional.

In one PHOLD configuration we included 10 hot-spot LPs, towards which 50% of the events injected by the other LPs are routed. As already hinted, PDES workloads with hot spots are difficult to manage since they might provide unbalance in case of traditional PDES platforms relying in the binding between LPs and WTs—load

balancing, as in ROOT-Sim, can anyhow mitigate this problem. We decided to experiment with this kind of complex workload just to study how our new approach could overcome such known limitations. In any case, for fairness, we also report data with the PHOLD model configured with no hot-spots, thus naturally leading to a more balanced advancement of virtual time (per wall-clock-time unit) across the LPs, independently of the underlying execution platform among the ones we compare. Finally, in this study we focus on a zero-lookahead scenario.

In Figures 7.7-7.8 we report data related to the configuration with no hot spot. By the plots we see that SE suffers from performance degradation with respect to USE. In fact, USE allows achieving a maximum speedup—over sequential execution—which is about 34% (resp 18%) better than the one provided by SE for event granularity set to 60 (resp. 120) microseconds. Such a maximum value is achieved for 32 WTs, where USE allows for better exploitation of parallelism, via speculative processing, with respect to SE.

As a note, approaches simpler that USE, which support a limited level of speculation (e.g., one speculated event per LP as in SE) can be more efficient in particular scenarios with a relatively reduced level of parallelism (up to 16 cores) and fine-grain events (60 microseconds); this is why in Figures 7.7-7.8 we note a slightly overcome of SE's performance with respect to USE. In any case, both SE and USE perform better than traditional speculative PDES, namely ROOT-Sim, since they avoid a lot of operations that the traditional engine needs to carry out. As an example, in our USE proposal, the cancellation of events that are no longer valid does not require any anti-event—since it is embedded within the non-blocking event state-machine management in the form of a simple event-buffer state transition. Also, no output queues are generated and traversed for managing rollbacks, since all the work is supported at the level of the SQ where the "positive" copy of the events is posted—still thanks to event-buffers state machines.

When moving to the hot-spot case—whose speedup data are provided in Figures 7.9-7.10—the potential of USE becomes definitely more evident. SE shows performance worse than USE, just because of the impossibility to provide scalable virtual-time synchronization (with no speculation) when hot-spot LPs tend to slow down the advancement of the commit horizon of the simulation. For this workload, USE can provide performance up to 64% (resp. 105%) better than SE for event granularity set to 60 (resp. 120) microseconds. This is a hard-workload scenario which ROOT-Sim cannot cope with in effective manner, even though it implements load balancing. In fact, with very few spots—10 in our case—long term planning in the distribution of the workload does not capture sudden unbalance, which becomes extremely adverse to performance especially when the number of WTs oversteps the number of hot-spot LPs. On the contrary, USE allows to concentrate the overall computing power towards all the events that are close to the current commit horizon, regardless of their distribution with respect to the hot spots (or other LPs). As a result, the system gains much more effective parallel execution, with much less likelihood of rollback operations. This phenomenon is evidenced by the efficiency data[3] in Figures 7.13-7.14, where we show that even for this hard workload, USE

[3]We recall that the efficiency of a speculative PDES run is the ratio between the number of committed events, and the total number of processed events, namely committed plus rolled back.

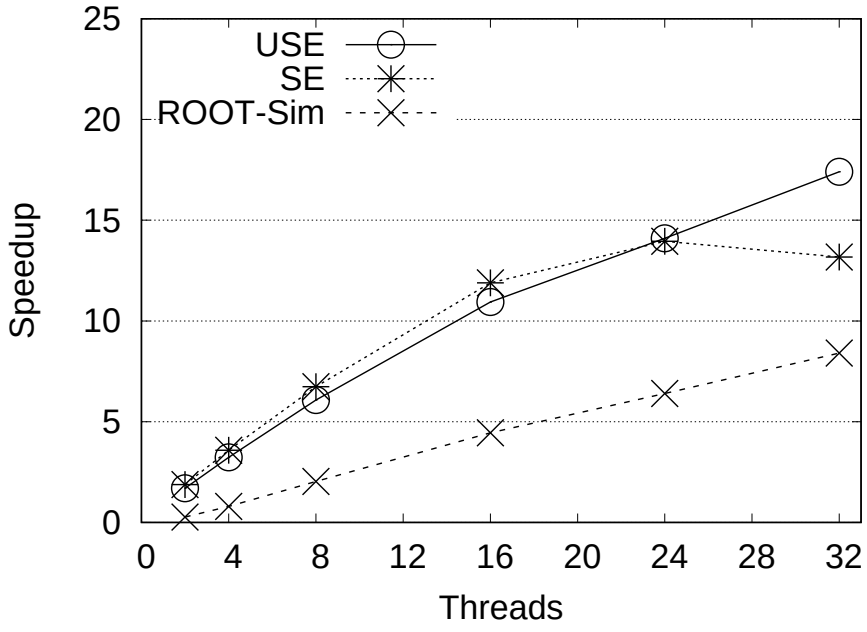

Figure 7.7. PHOLD (no hot spot) speedup results, event granularity equal to 60 μs

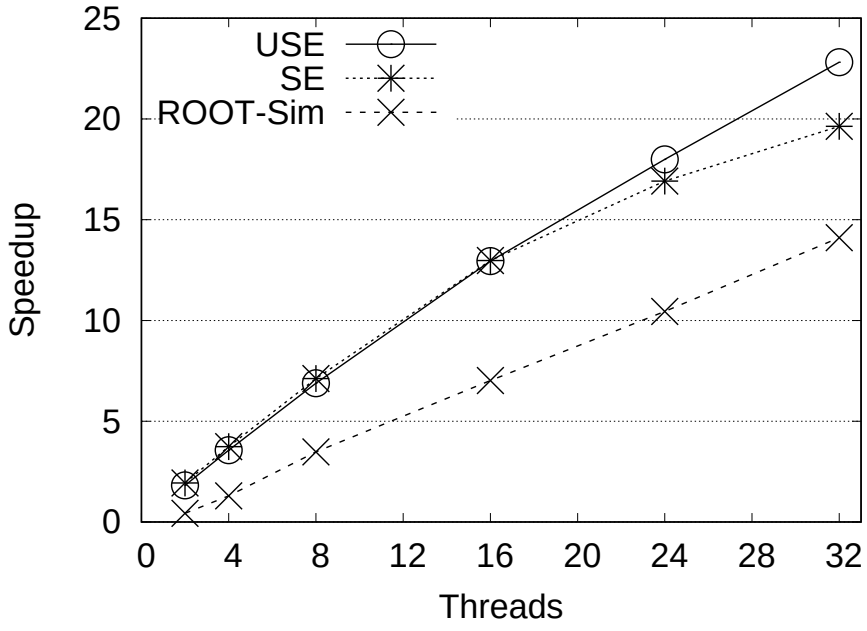

Figure 7.8. PHOLD (no hot spot) speedup results, event granularity equal to 120 μs

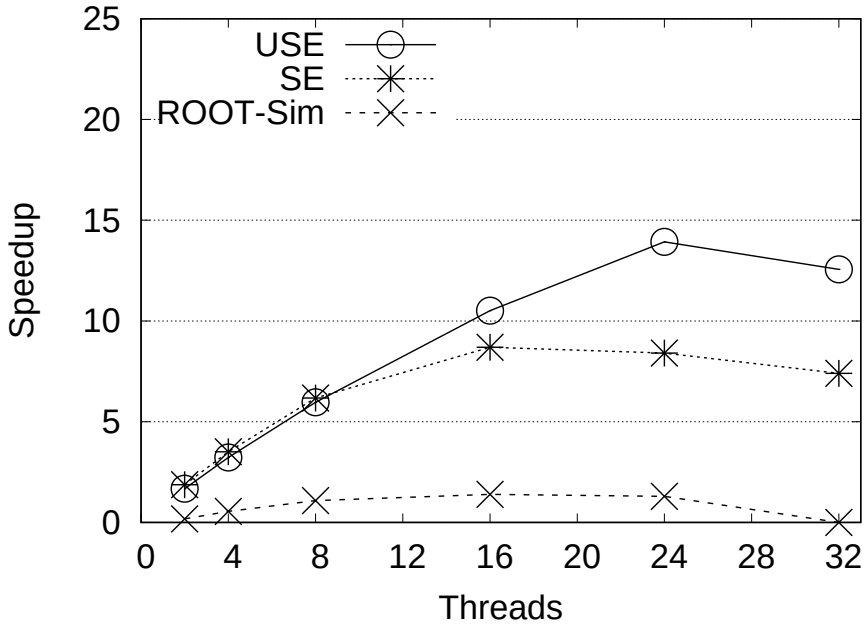

Figure 7.9. PHOLD (hot spot) speedup results, number of spots equal to 10, probability of hitting the spot equal to 0.5 and event granularity equal to 60 μs

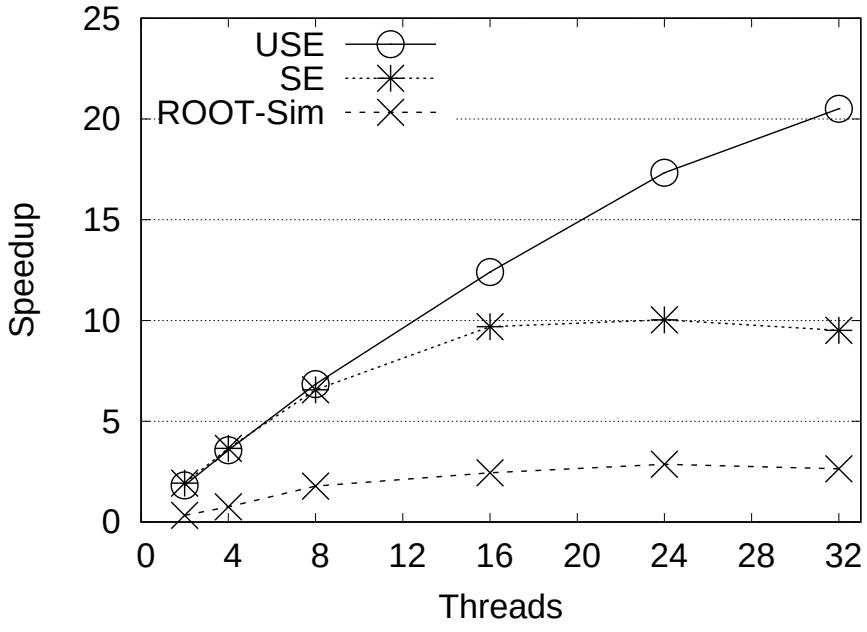

Figure 7.10. PHOLD (hot spot) speedup results, number of spots equal to 10, probability of hitting the spot equal to 0.5 and event granularity equal to 120 μs

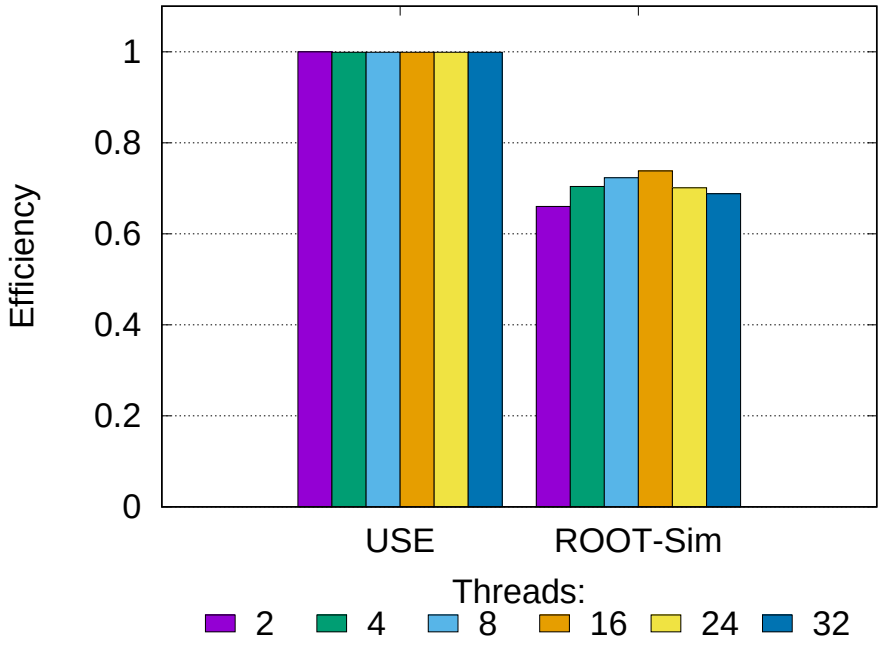

Figure 7.11. PHOLD (no hot spot) efficiency, event granularity equal to 60 μs

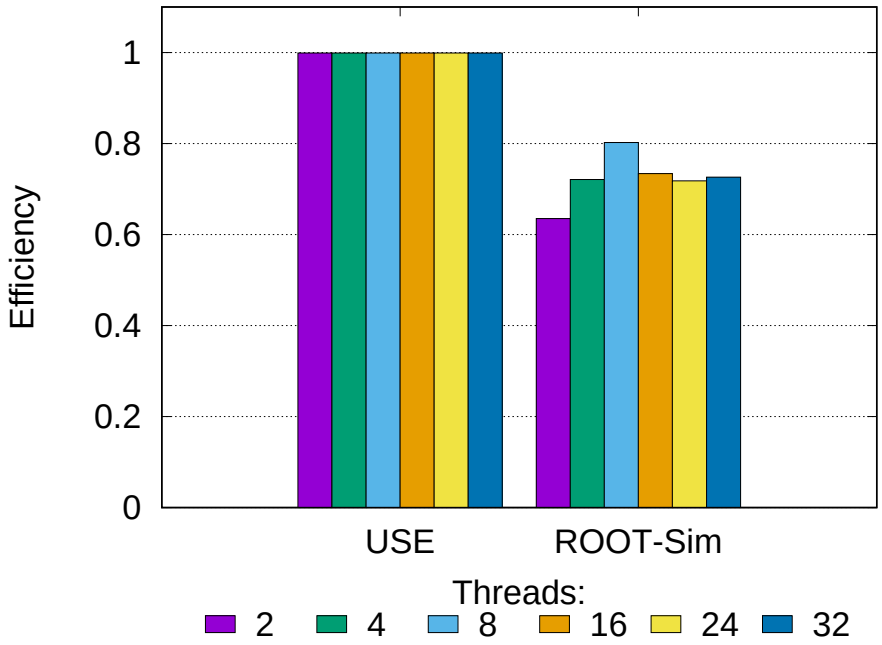

Figure 7.12. PHOLD (no hot spot) efficiency, event granularity equal to 120 μs

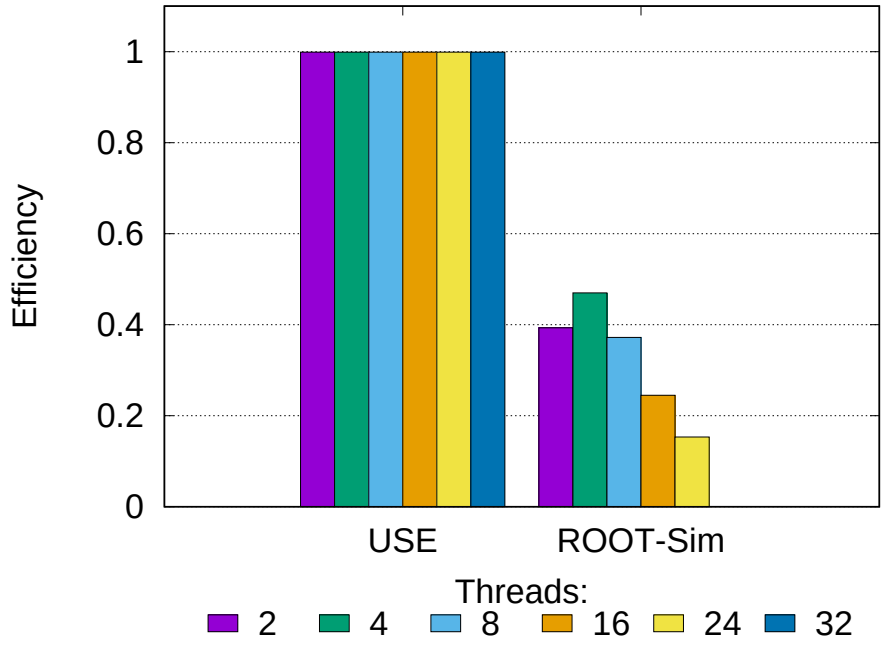

Figure 7.13. PHOLD (hot spot) efficiency, number of spots equal to 10, probability of hitting the spot equal to 0.5 and event granularity equal to 60 μs

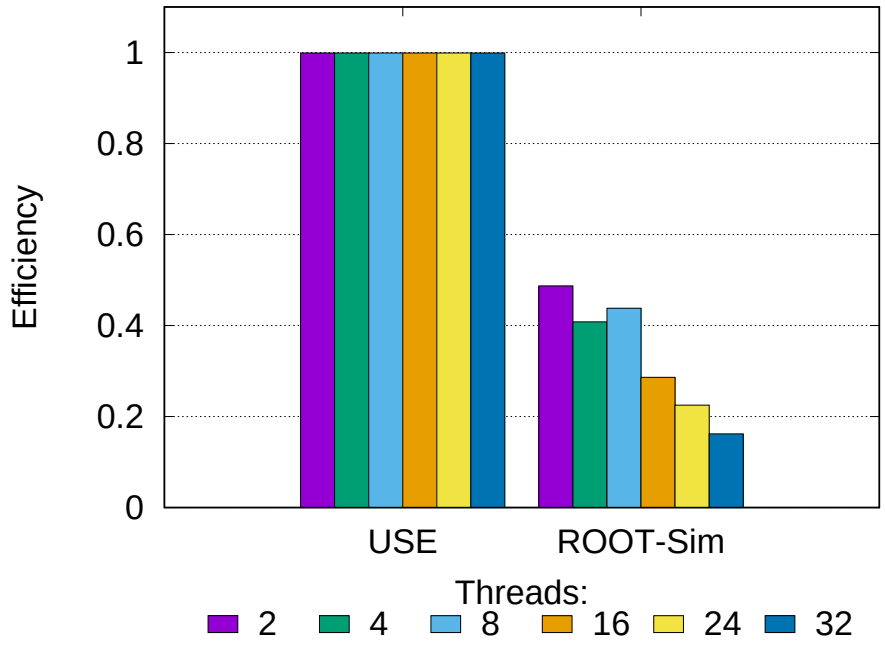

Figure 7.14. PHOLD (hot spot) efficiency, number of spots equal to 10, probability of hitting the spot equal to 0.5 and event granularity equal to 120 μs

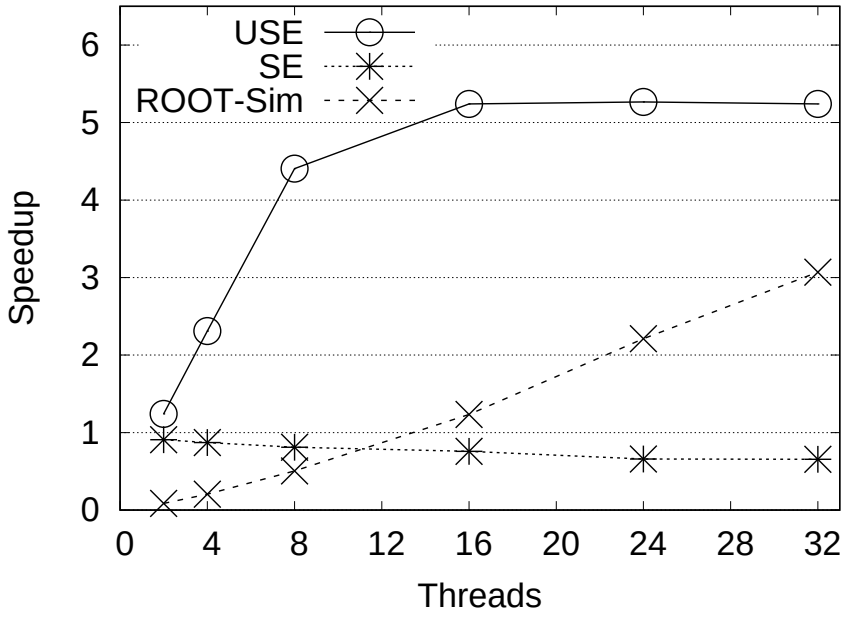

Figure 7.15. Speedup results for TCAR

achieves almost 100% of efficiency, as opposed to ROOT-Sim, which only achieves the order of 40% or less. In Figures 7.11-7.12 ROOT-Sim performs better in the configuration with no hot-spots where it reaches much higher values of efficiency around 70%, though still significantly lower than USE.

7.2.2 TCAR speedup results

As the second test-bed application, we used a variant of the Terrain-Covering Ant Robots (TCAR) previously introduced in Section 5.2.2.2 in which the activities performed by each robot within a cell are extremely lightweight. We configured TCAR with 1024 cells, with 15% of the cells initially set to be occupied by a robot.

In Figure 7.15 we show speedup results with the TCAR model. An important aspect for this application is that it shows significantly finer grain events—of the order of a few microseconds—compared to the used configurations of PHOLD. This stresses the execution of the different tested engines along another dimension, with respect to what done with PHOLD. In such a scenario, the overhead for parallelization that is paid by a traditional engine like ROOT-Sim, including its lower efficiency compared to USE, leads to very limited speedup. The same is true for SE, given its impossibility to carry out Time-Warp style speculative processing and the consequent active waiting, namely spinning behind a lock, for causality re-alignment, that increases contention on the underlying memory subsystem. Instead, USE allows achieving speedup that increases up to 16 threads. It then flattens, indicating somehow that more threads do not longer pay-off for performance reduction. We note that USE reaches its maximum speedup beyond a number of threads much less than ROOT-Sim indicating its superior ability to exploit actual model parallelism even when scaling up the amount of resources to more limited extents.

Chapter 8

Related literature results and discussion

While PDES has been thoroughly studied in distributed environments—like clusters of machines possibly relying on high speed interconnection [108]—the trend of devising and optimizing it in multi/many-core platforms purely based on shared memory is a relatively recent one. In this chapter we focus on surveying the latter proposals, discussing the relation and the differences with the solutions presented in this thesis.

8.1 Optimized data exchange

An important benefit offered by a multi-core machine is the ability to exchange information without the need to rely on explicit message passing primitives based on memory copies of the exchanged data.

Relying on this capability, in [109] the authors drop-down the classical multi-process PDES paradigm shifting to the multi-thread one to deeply exploit shared memory benefits offered by a common virtual address space. In this work such capability is exploited in order to put in place a zero-copy message passing mechanism, relying on the sharing of messages' pointers, integrated with GVT calculation in order to determine when buffers could be reused.

Similarly, in [68] the authors suggest a series of optimizations to enhance a multi-threaded PDES platform. In particular, these optimizations are tested on the ROSS-MT platform [64], a multi-threaded instance of ROSS [58]. The major contribution by this work is related to the placement of memory buffers used to exchange messages. Similarly to [109], message copies are avoided by enqueueing the pointer to the original message directly into the input queue of the receiver. However, in [64] buffer management is enhanced by splitting the free memory pool in order to track the allocation source. Thus, when a memory buffer is freed, this is not stored by the receiver in its pool, rather it is returned to the one of the sender, in order to reduce memory access latencies: in this way, in case the PDES system is executed on NUMA machines, at least one of the two parties involved in a message exchange is near to the memory buffer location. Moreover, a more

complex mechanism is proposed, in order to guarantee optimal memory access for couples of WTs, rather than only one, by introducing intermediate memory pools in memory regions characterized by reduced memory access latencies for both the parties. Along this line, a more advanced proposal is found in [110], where NUMA optimization is achieved in a combined manner for event-buffers and for buffers belonging to the state of the LPs.

A zero-copy approach is also used in our solutions considering that each event is directly filled into the fully shared event pool and can be picked by any WT with no need for additional copies. However, while the works in [109, 68, 111] directly tackle the problem of safely re-allocating shared memory portions, we adopt ad-hoc non-blocking mechanisms for memory reuse in combination with non-blocking mechanisms for memory buffers access/update.

8.2 Shared data access

A recent bunch of literature on PDES on shared memory multi-core platforms has focused on removing the limitations associated with state partitioning across LPs. In [112] a solution is provided enabling WTs to access global variables of the simulation model on multi-core machines. This is achieved by relying on code instrumentation, leading to the runtime ability to identify shared variable accesses in order to manage these in the proper way. In particular, each time the application code tries to read/ write a global variable, these operations are redirected to a list-based multi-version implementation. When a write operation is performed, a new version of the variable is generated, marked with the LVT of the LP that has been CPU-scheduled along the writing WT. On the other hand, each time a WT performs a read, the version associated with a timestamp strictly smaller than the LVT of the "reader LP" is returned, and its identifier (linked to its current LVT) is appended to a list of readers. In order to manage straggler writes, if a WT updates a variable in the past, all LPs that have accessed such variables in the future (i.e., they accessed a version with a larger timestamp) are rolled back, thus ensuring consistency. Finally, the authors provide a non-blocking implementation to guarantee scalability in the access to versioning lists.

In [113] the information-sharing problem is tackled by allowing each event handler activated at an LP to access directly any memory area that has been dynamically allocated by any other LP (cross-state access). This is achieved by encapsulating the access to multiple LP states by an event handler within an atomic action that is, in its turn, based on an ad-hoc synchronization protocol triggered on demand, if and only if a cross-state access materializes. Such a materialization is detected by relying on an advanced Linux oriented virtual memory management architecture able to track at runtime the access to the state of whichever LP.

In [114], the work in [113] is improved by introducing granular simulation objects. Specifically, given that in general contexts, a cross-state access by an event handler could be the evidence that two or more LPs are executing in a synergistic way, in terms of overall simulation model execution trajectory, the approach presented in [114] aims at reducing the number of times the cross-state synchronization protocol needs to be activated. The basic idea exploited in this work is that the

aforementioned LPs can be clustered in islands depending on their mutual cross-state accesses, in order to sequentially execute groups of strictly correlated LPs as a unique object, named granular LP (GLP).

These proposals are still bound to the traditional PDES architectural paradigm. In fact, they have been integrated into environments that still rely on LP/GLP (namely workload) partitioning across threads, rather than fine-grain sharing of individual work units (single events) like in the solutions provided in this thesis.

The work in [63] divides the LP state attributes in three different categories: per-LP private attributes, not accessible by other LPs; public attributes, still belonging to an LP but accessible by others; common attributes, not belonging to an LP and accessible by anyone. In order to access public and common attributes, the authors use a mechanism based on Software Transactional Memory, thus essentially relying on instrumenting read/write operations to memory locations. This is quite different from the proposals in this thesis since we do not rely on any form of hardware/software instrumentation for driving memory operations by event handlers to guarantee isolation. In fact, in the share-everything PDES systems we presented, we adhere to the traditional paradigm based on data separated accesses to the LP states. Also, the proposal in [63] allows WTs to share the workload of individual events, but moves these events across the WTs by relying on spinlock protected separate queues, an approach that has been shown to be effective only when there are very low CPU-core counts. Differently, we have pursued a fully non-blocking approach in the management of any data structure in our share-everything PDES proposals.

8.3 Uniform simulation advancement

One of the most significant aspects that has conducted our work in the direction of fully-shared PDES platforms is to provide a system able to advance the simulation time along the different LPs fringe evenly, with very fine control of the delivery of computing power to the simulation model. As for this aspect, the literature offers variations of load-balancing approaches, a few of which have characteristics strictly related to the shared-memory multi-core architecture.

The solutions in [107, 72] take advantage of the possibility for any thread to promptly access the state of any LP and of its event pool when a re-bind between LPs and threads is needed—for load sharing—depending on the LPs' current weight in the computation. Actually, this is the approach that we have used as competitor of our share-everything PDES solution in the experimentation—it is in fact implemented in the ROOT-Sim speculative PDES system that we used as a reference literature platform for the comparative study.

Although in [107] the distributed paradigm, characterized by multiple simulation kernel instances, is preserved, each one of these is deployed following the multi-threaded approach with the goal to execute a sort of load-balancing operations exploiting load-sharing mechanisms. Considering a multi-core machine characterized by C_{tot} CPU-cores on which K_{tot} simulation kernel instances are concurrently executed, the goal of such approach is to determine periodically the amount of CPU-cores C_i (corresponding to the same number of WTs) to assign to each kernel

instance K_i for a given time frame in order to have a balanced virtual-time advancement across LPs belonging to different kernel instances. So, the idea at the base of this work is to move CPU-cores across kernels, rather than moving LPs between kernel instances running on top of a fixed number of CPU-cores. Once a resizing phase of the kernel computation is carried out, each simulation kernel has to re-bind LPs to WTs depending on the amount of work requested by each LP, in order to allow a balanced inter-kernel advancement. An additional interesting contribution offered by such work is about the internal architectural organization of each kernel instance. While the event pool is still decoupled in multiple input queues (one for each LP), the authors are interested in reducing the synchronization pressure on each one of these while exchanging messages across threads. In particular, [107] introduces a mechanism similar to the one adopted by modern Operating System drivers, where, each cross-LP interaction is materialized in a couple of top/bottom-half operations. Thus, when a WT wants to deliver a message to an LP input queue, it does not lock the corresponding queue, instead it executes a light top-half function that registers the bottom-half one, passing the relative parameters, within a per-LP bottom-half queue, which is asynchronously processed by the destination WT in constant time by flushing these messages in the input queue of the target LP.

In [72] a load-sharing mechanism is used to reduce the negative effects produced by interferences caused by "simulation-external" workload. When the Operating System schedules a thread not running within the PDES system, a context switch is performed scheduling off the CPU a WT running within the PDES system—this is definitely true when the number of used WTs in the PDES system is equal to the number of available CPU-cores—which will be delayed. While in such situation the ideal slow-down is proportional to the number of cores stolen by the external workload, in practice the impact is significantly worse. In fact, the result of such scenario is that, while the rest of the simulation advances, the LPs bound to the descheduled WT are stalled, increasing the probability of virtual-time disalignment and rollback generation when they will be resumed. In this article authors face such problem by reducing the number of active threads when an interference is detected. Thus, since the adopted simulation platform is based on LP partitioning among WTs, when one or more of these are inactive, each active WT periodically helps the inactive one to carry on its simulation workload, in order the keep the advancement of the simulation balanced also in the presence of interferences. In the share-everything solutions provided in this thesis the ability to react to external workloads is endemic since we adopt non-blocking coordination (hence no descheduled thread will ever delay the others in their steps, especially in the "ultimate" configuration") and we keep the computing power of non-descheduled threads close to the commit horizon.

In [111] the problem of an unbalanced distribution of workload between WTs is solved according to a different perspective, exploiting mechanisms offered by modern processors to control at runtime the frequency and voltage of a chip. This approach exploits Dynamic Voltage and Frequency Scaling (DVFS) [115], commonly used to cap the power consumption of a system while maximizing the throughput. In this proposal, a set of LPs is bound to a given WT and are executed speculatively. Then, the thread that experiences an overoptimistic execution (with higher incidence of rollbacks) slows down the operating frequency (or power state) of the CPU-core so

as to rebalance the advancement of the LPs along virtual time. The proposals in this thesis are fully orthogonal to this approach, since we allow the CPU-cores to still work at maximum power usage while concentrating such power on the events close to the commit horizon, in order to advance all the LPs along virtual time in a balanced manner.

More generally, all the above solutions are based on traditional partitioning of the workload (the LPs) across the threads, and on periodic re-evaluation of partitions. Rather, we consider any individual event as a work unit that can be dispatched along any thread in the PDES system—ideally at any time instant—leading to implicit balancing capabilities when WTs pick at any time the higher priority events across the overall set of LPs.

8.4 Approaches exploiting HTM

As discussed in Chapter 3, an important aspect to keep in mind while devising a concurrent application to be run on top of multi-core machines, is to guarantee data consistency while still targeting performance. One of the most diffused alternatives to mutual exclusion is represented by Transactional Memory (TM), a concurrency control mechanism which enables concurrent accesses to critical sections by dynamically managing variable updates in order to ensure serializability. In particular, a recent solution offered by processor manufacturers is Hardware Transactional Memory (HTM), where accesses to shared resources are managed via a proper hardware implementation exploiting cache line invalidation. A thread working in hardware-transactional mode tries to execute the critical section atomically without limiting the behavior of other threads: if a conflict is detected on a written/read cache line, the transaction is aborted, discarding all memory operations (by simply squashing the cache content). However, the HTM support is constrained both in time (each time that a context switch or mode switch is performed the cache is invalidated aborting the transaction) and space (if the data touched by the transaction exceed the cache size or two unrelated accesses are mapped on the same cache line, the transaction is aborted), thus limiting its employment in a restricted set of scenarios.

As for the PDES field, HTM has been used in a few solutions. In [116] it has been exploited to make LP state updates automatically recoverable via hardware support in speculative executions, a topic that stands as orthogonal to the one we tackle in this thesis.

In [98], HTM facilities are exploited in WARPED, a nowadays multi-threaded speculative PDES platform, with the goal to improve scalability of the accesses to shared event queues, compared to lock-based approaches. WARPED [57] follows an asymmetric organization, with a manager thread employed to perform housekeeping tasks (e.g., message exchange) and a set of worker threads allowed to process events destined to any LP. The scheduling mechanism is based on a two-level pending event queue: each LP keeps an ordered to-be-processed pool (coupled with a processed event queue used for rollback purposes) used to populate a global priority queue storing, for each LP, the next event to be processed (see Figure 8.1). At each simulation loop iteration, a WT tries to lock the global priority queue to fetch the next event to be processed (namely the one with the smallest timestamp in the

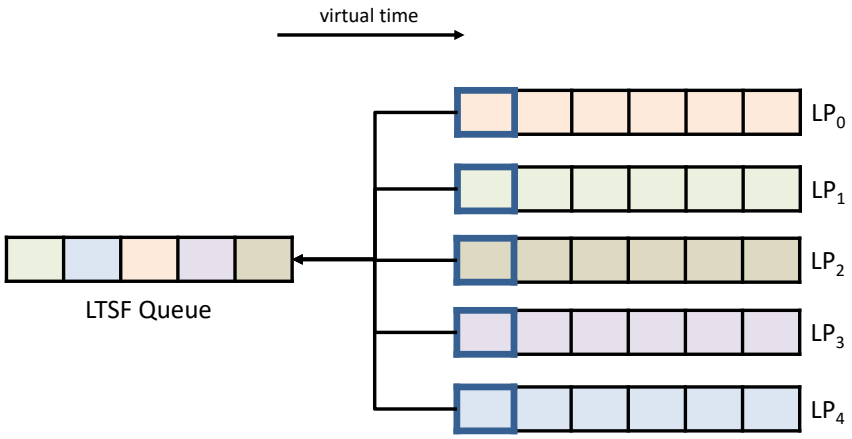

Figure 8.1. Two-level priority queue data structure

system). Once the WT succeeds in such extraction, it executes such event on the associated LP—it is interesting to note that, even if any WT is able to process any event, it is not required to reserve (book) the corresponding LP since there are no additional events destined to it that can be retrieved from the global priority queue. Once the event processing is completed, the WT attempts to lock the to-be-processed event pool associated with the LP for which it has just executed the event in order to extract the next event to be processed on this same LP: this event is then enqueued in the global priority queue, still exclusively reserving such resource. With this organization, only a WT at a time is enabled to work on each queue, leaving everyone else that is trying to access the same resource in a pending state—these queues are used also to store newly generated events. A first attempt to reduce the contention has been to split the two-level structure in multiple instances, each one related to a subset of LPs, and managed by a subset of WTs, thus creating a (temporary) binding between groups of LPs and groups of WTs. Such solution is more prone to an unbalanced advancement, thus load-balancing techniques are required. Moreover, while in this way the number of WTs trying to fetch events is reduced, the amount of enqueuers is still proportional to the number of WT employed in the simulation. In this scenario, the HTM support has been studied to enhance the LTSF queue implementation in order to allow concurrent accesses, providing two alternatives solutions based on the different interfaces offered by the HTM support.

Similarly to the solutions provided in this thesis, in this work any WT is allowed to execute any LP delivering the computing power near to the commit horizon (in the multi-list version this consideration is relaxed). On the other hand, in order to reduce the pressure on the shared pool, we rely on non-blocking software-based techniques. Moreover, it is important to note that HTM facilities are implemented only on processors characterized by a small amount of cores, reducing in turn the intrinsic scalability of such support. Overall, differently from the solution provided in [98], the share-everything PDES architectures presented in this thesis can scale still relying on a single fully shared event pool, thus avoiding at all the need to include load-balancing mechanisms, hence automatically coping with their limitations.

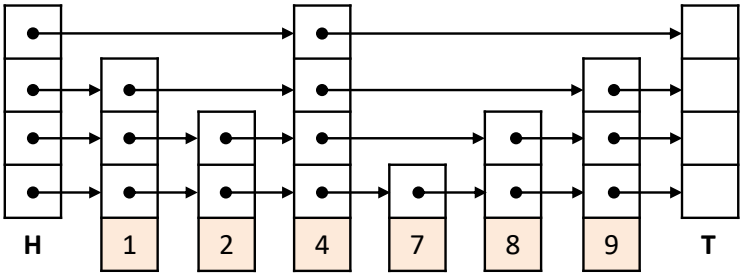

Figure 8.2. Skip-list data structure

8.5 Scalable access to shared event pools

The topic of providing event-pool data structures enabling concurrent accesses has been addressed in [117], in which an approach based on fine-grain locking of a subportion of the data structure upon performing an operation is proposed. However, the intrinsic scalability limitations of locking still lead this proposal to be not suited for large levels of parallelism, as also shown in [118]. Rather, in the solutions provided in this thesis, we base concurrent accesses to shared data structures on non-blocking algorithms, which have been shown to be much more prone to scalability.

As for non-blocking management of sets by concurrent threads, various proposals exist, such as lock-free linked-lists [96], skip-lists [93, 119] and calendar queues [91].

In [119] the authors devise an interesting solution providing an improved version of the original non-blocking skip-list presented in [93]. A skip-list [10] is a priority queue, similar in spirit to a search tree, able to guarantee fast extractions without performing periodic housekeeping tasks—a balanced node structure is guaranteed by a probabilistic approach. It is organized in a linked hierarchy of subsequences, with each successive subsequence skipping over fewer elements than the previous one. Practically, it is composed of a multilevel ordered linked-list, where each level contains a subset of the nodes in the list below, such that the lower layer contains all the nodes belonging to the current state of the structure (Figure 8.2). Thus, a search procedure to insert a new node starts by the top level list, finding the interval of the node to be inserted, and then gradually descending levels, until the lowest one where the final node position is found. While inserting a node, a maximal height is randomly defined (following a geometrical distribution) in order to decide the level, starting from the lowest one, at which the new node will participate. On the other hand, to extract a node with the highest priority it is enough to traverse the head to find the first element and then extract it by removing all its references (at the different levels) from the structure. The solution proposed in [93] uses a non-blocking linked-list for each level. Differently from its classical implementation, in order to guarantee correctness with respect to the multilevel organization, each node—the same node is used on multiple levels relying on an array of pointers to the next one—is provided with an additional value used to notify its current state (e.g., insertion or deletion). However, the marking bit in the next pointer is still used to prevent an erroneous removal of newly inserted nodes, as seen in Section 3.1.1. Thus, in order to extract a node, it is first marked as to be removed (relying on the status variable), and only later it is logically deleted by setting the next-field

marking-bit, for each level in which it is linked, in ascending order. On the other hand, while a node is inserted, it is continuously checked to be still valid, in order to suddenly stop the insertion of a concurrently removed node. In [119] this solution is enhanced to reduce the pressure on the highest priority node, by reducing the contention on the dequeue operation. Indeed, while in the previous solution at least two RMW instructions are required to remove a node (logically and physically), here it is required just a `CAS`—batch of nodes are removed only when a minimal amount of logically deleted nodes is reached. In order to enforce correctness while relying on this conflict resilient organization, the non-blocking linked-list has been reshuffled moving the marking bit of a deleted node in the next pointer of its predecessor.

The solution proposed in [119] appears to be similar in spirit to the one introduced in this thesis, however, while the skip-list requires logarithmic time complexity both for insertion and deletion, we are able to provide amortized constant time complexity for both these operations. Indeed, to the best of our knowledge, the non-blocking calendar queue is the only priority queue able to provide such property.

Overall, as compared to the proposals in this thesis, the above mentioned solutions have three main core limitations: 1) they (mostly) do no offer conflict resilient extractions of the highest priority event; 3) they exhibit worse time-complexity.

Non-blocking operations on event pools have also been studied in [89]. The simulation platform on which such improvement is devised is the one in [98], introduced in the previous section. However, differently from [98], where multiple pending event sets are proposed to reduce contention on the shared pool, in [89] authors rely on a single pending event set (still organized with a two-level structure) in order to avoid problems linked to an unbalanced advancement of the simulation. This is essentially a scheduling queue keeping the minimum timestamp event for any LP. While for enqueue operations the contention problem is automatically mitigated by the per-LP linked-list structure, the scheduling queue represents a unique synchronization point for dequeue operations, thus requiring ad-hoc mechanisms to reduce contention. A variant of the Ladder Queue (LQ), identified in [92] as the reference data structure to provide performance improvement, is employed. The classical LQ is organized in a multi-layer structure. Incoming events are inserted into an unordered set, named `top`. On receiving a dequeue request, such events are delivered in the first rung of the LQ. Each rung is an array of buckets, namely unordered sets of events partitioned in time spans. When the number of events contained in a bucket exceeds a defined threshold, a new rung, where such events are distributed, is defined. When a dequeue request is performed, the content of first non-empty bucket is inserted in a sorted list, named `bottom`, used to process subsequent requests. However, in optimistic PDES is non strictly required to process events in timestamp order, since rollback mechanisms are used in case a causal violation occurs. Moreover, depending on the size of a bucket, there is a high probability that events belonging to the same bucket are causally independent of each other. Thus, in this work the guarantee to retrieve each time the event with the smallest timestamp is relaxed in order to remove sorting costs while allowing a high performance non-blocking implementation. To this end the solution proposed in [120], suited for unordered list of elements, is extended. In such organization, a state machine for each element is defined, composed of two intermediate states

(inserting and removing) and two stable states to identify inserted (valid) and removed (invalid) elements. Thus, before inserting or removing a node, a new node is inserted to the head of the list by notifying the operation to be performed, in order to complete it once its validity is verified.

This proposal is intrinsically tailored to PDES systems relying on speculative processing, where unordered extractions leading to causal inconsistency within the simulation model trajectory are reversed. In the proposals in this thesis we guarantee the ordering of the events in the shared pool, which allows us to enforce the smart combination of conservative (i.e., safe) and speculative processing at the level of each individual event—also thanks to the explicit exploitation of the lookahead in the simulation model—thus enabling the optimization of the rollback support, an aspect that is not considered in [89]. Also, in this work the non-blocking data structure is essentially used as a CPU-dispatching support allowing threads to pick the next event to be processed. Differently, in our solution the event pool is used as a more complex central synchronization point in care of multiple tasks like, e.g., GVT computation.

Chapter 9

Conclusions

It is undeniable that the current hardware trends towards ever more parallel machines based on the multi-core paradigm is providing fully new opportunities. In fact, high computing power is no longer achievable by only clustering machines through some (high speed) interconnection network; rather, sufficiently large computing power, and main-memory storing capabilities, can be offered by an individual medium-to-high end multi-core machine. At the same time, software applications need to be rethought in terms of their structure in order to actually exploit these opportunities.

Focusing on the scientific computing context, or more generally on HPC applications, the multi-core era is opening the possibility to devise a new way of managing the workload, based on concepts that shift to the opposite side with respect to classical ones. In a cluster, a core issue to tackle is to determine how to partition the workload across the computing units in order to optimize runtime parameters based on the exploitation of locality, under the selected thread coordination scheme. On a multi-core machine the new dimension of sharing the workload at very fine granularity for further optimizing aspects related to runtime efficiency comes out.

Starting from this preamble, in this thesis we have proposed new solutions for the architectural organization of classical speculative Parallel Discrete Event Simulation (PDES) platforms, by moving towards a so called *share-everything* paradigm. This is based on the simple idea that all the threads (so all the CPU-cores) running the PDES application can fully share the accesses to whatever data representing the state of the simulation system—thus including both kernel level and application level data. Events—and their representation in memory via buffers indicating that they are there and need to be taken care of—become therefore fully shared data units leading to share across threads the fine grain tasks associated with them. This, in combination with adequate algorithmic solutions, allowed us to devise PDES systems able to concentrate the computing power towards the more relevant part of the overall simulation workload at any time instant.

We note that in the solutions we have proposed, the term "sharing data" is no way synonymous of "critical section". In fact, our algorithmic solutions pursue the objective of making coordination across threads in the access to shared data fully non-blocking. Actually, this is a prerequisite of any modern software platform

enabling scalability of thread operations on shared-memory multi-core machines—clearly in scenarios where we do not impose data (workload) partitioning schemes that would otherwise lead to limitations on punctual decisions on how to exploit the computing power offered by each single CPU-core.

Our non-blocking algorithms for PDES on multi-core machines mix together scalable synchronization aspects along two dimensions, namely wall-clock time and virtual (simulation) time—the latter via speculative processing techniques of simulation events—thus enabling unleashed exploitation of the hardware computational power. At the same time, unleashed actual parallelism while executing simulation models comes together with an improvement in the likelihood of correct speculation, a result that appears somehow counterintuitive when keeping our minds still bound to traditional coarser grain association of the workload to threads. Overall, we feel that our proposals stand as the result of looking at the virtual-time and wall-clock-time coordination problem in PDES systems from an alternative perspective, which we think can be helpful for researchers independently of the real potential or value demonstrated by our solutions in the experimental studies we presented.

At the same time, although one of the versions of share-everything PDES systems provided in this thesis has been named as *ultimate*, we firmly believe that the presented outcomes still stand a starting point for further research achievements. As for the latter assertion, we note that our solutions have been based on baseline hardware synchronization facilities, like classical RMW instructions offered by off-the-shelf processors' ISA. However, the presented algorithms do not embed solutions based on pervasive awareness of hardware heterogeneity. We intend therefore to further step ahead along this research path towards the inclusion of additional solutions coping with heterogeneity in both the timeliness of firmware operations in modern hardware—such as NUMA systems—or even ISA diversity—such as when combining multi/many-core processing capabilities based on joint usage of CPU/GPGPU technologies.

Chapter 10

Bibliography

[1] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, pp. 114–117, Apr. 1965. 1

[2] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005. 1

[3] E. Intel, "Speedstep® technology for the intel® pentium® m processor," 2004. 1

[4] S. Robinson, *Simulation: The Practice of Model Development and Use*. John Wiley & Sons, 2004. 7

[5] B. P. Zeigler, *Multifacetted modelling and discrete event simulation*. Academic Press Professional, Inc., 1984. 8

[6] B. P. Zeigler, "Hierarchical, modular discrete-event modelling in an object-oriented environment," *Simulation*, vol. 49, no. 5, pp. 219–230, 1987. 8

[7] A. Pellegrini, *Techniques for Transparent Parallelization of Discrete Event Simulation Models*. PhD thesis, Sapienza, University of Rome, 2014. 8

[8] G. A. Wainer, *Discrete-event modeling and simulation: a practitioner's approach*. CRC Press, 2009. 10

[9] J. O. Henriksen, R. M. O'Keefe, C. D. Pegden, R. G. Sargent, and B. W. Unger, "Implementations of Time (Panel)," in *Proceedings of the 18th Conference on Winter Simulation*, WSC, pp. 409–416, ACM, 1986. 10

[10] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, pp. 668–676, jun 1990. 11, 117

[11] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985. 11

[12] R. Brown, "Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, no. 10, pp. 1220–1227, 1988. 11, 46, 53, 55, 57

[13] T. Dickman, S. Gupta, and P. A. Wilsey, "Event pool structures for pdes on many-core beowulf clusters," in *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, pp. 103–114, ACM, 2013. 11, 45

[14] R. M. Fujimoto, "Parallel Discrete Event Simulation," in *Communications of the ACM*, vol. 33 of *WSC*, pp. 19–28, ACM Press, 1989. 13, 16, 64, 80, 104

[15] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE–S5, no. 5, pp. 440–452, 1979. 13, 16, 17

[16] P. F. Reynolds Jr., "A Spectrum of Options for Parallel Simulation," in *Proceedings of 1988 Winter Simulation Conference*, pp. 325–332, Society for Computer Simulation, 1988. 16

[17] R. E. Bryant, "Simulation of Packet Communication Architecture Computer Systems," Tech. Rep. MIT/LCS/TR-188, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977. 16

[18] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Communications of the ACM*, vol. 24, no. 4, pp. 198–206, 1981. 16

[19] D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and System*, vol. 7, no. 3, pp. 404–425, 1985. 16, 18, 20, 43, 88

[20] H. M. Soliman and A. S. Elmaghraby, "An Analytical Model for Hybrid Checkpointing in Time Warp Distributed Simulation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 947–951, 1998. 16

[21] S. Srinivasan and P. F. Reynolds Jr., "Elastic Time," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, pp. 103–139, Apr. 1998. 16, 42

[22] R. N. Zobel and D. P. F. Möller, eds., *12^{th} European Simulation Multiconference - Simulation - Past, Present and Future, June 16-19, 1998, Machester, United Kingdom*, SCS Europe, 1998. 17

[23] C. D. Carothers and K. S. Perumalla, "On deciding between conservative and optimistic approaches on massively parallel platforms," in *Proceedings of the 2010 Winter Simulation Conference*, pp. 678–687, 2010. 17

[24] V. Jha and R. Bagrodia, "Simultaneous Events and Lookahead in Simulation Protocols," *ACM Transactions on Modeling and Computer Simulation*, vol. 10, no. 3, pp. 241–267, 2000. 17

[25] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*, pp. 135–148, IEEE Computer Society Press, 1981. 18

[26] F. Chang and G. A. Gibson, "Automatic i/o hint generation through speculative execution," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 1999. 18

[27] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, pp. 213–226, June 1981. 18

[28] Y.-B. Lin and E. D. Lazowska, *Reducing the saving overhead for Time Warp parallel simulation.* University of Washington Department of Computer Science and Engineering, 1990. 21, 88

[29] F. Quaglia, "A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 4, pp. 346–362, 2001. 21

[30] F. Quaglia, "Combining periodic and probabilistic checkpointing in optimistic simulation," in *Proceedings of the 13th workshop on Parallel and distributed simulation*, pp. 109–116, IEEE Computer Society, 1999. 21

[31] B. R. Preiss, W. M. Loucks, and D. MacIntyre, "Effects of the Checkpoint Interval on Time and Space in Time Warp," *ACM Transactions on Modeling and Computer Simulation*, vol. 4, no. 3, pp. 223–253, 1994. 21

[32] A. Pellegrini, R. Vitali, and F. Quaglia, "Autonomic state management for optimistic simulation platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 1560–1569, June 2015. 21, 22

[33] A. C. Palaniswamy and P. A. Wilsey, "Adaptive checkpoint intervals in an optimistically synchronised parallel digital system simulator," in *Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration*, pp. 353–362, North-Holland Publishing Co., 1993. 21

[34] F. Quaglia, B. Ciciani, and R. Baldoni, "Checkpointing protocols in distributed systems with mobile hosts: A performance analysis," *Parallel and Distributed Processing*, pp. 742–755, 1998. 21

[35] H. Bauer and C. Sporrer, "Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving," in *Simulation Symposium, 1993. Proceedings., 26th Annual*, pp. 12–20, IEEE, 1993. 22

[36] D. West and K. Panesar, "Automatic Incremental State Saving," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, PADS, pp. 78–85, IEEE Computer Society, 1996. 22, 23

[37] D. Cingolani, A. Pellegrini, M. Schordan, F. Quaglia, and D. R. Jefferson, "Dealing with reversibility of shared libraries in pdes," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, ACM, May 2017. 22

[38] M. Schordan, T. Oppelstrup, D. R. Jefferson, P. D. Barnes, and D. Quinlan, "Automatic Generation of Reversible C++ Code and Its Performance in a Scalable Kinetic Monte-Carlo Application," in *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, ACM Press, 2016. 22

[39] J. M. LaPre, E. J. Gonsiorowski, and C. D. Carothers, "Lorain: A step closer to the pdes 'holy grail'," in *Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, (New York, NY, USA), pp. 3–14, ACM, 2014. 22

[40] R. Rönngren, M. Liljenstam, R. Ayani, and J. Montagnat, "Transparent Incremental State Saving in Time Warp Parallel Discrete Event Simulation," in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pp. 70–77, IEEE Computer Society, 1996. 22

[41] A. Pellegrini, R. Vitali, and F. Quaglia, "Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects," in *Proceedings - Workshop on Principles of Advanced and Distributed Simulation, PADS*, pp. 45–53, IEEE, 2009. 22

[42] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto, "Efficient optimistic parallel simulations using reverse computation," *ACM Transactions on Modeling and Computer Simulation*, vol. 9, no. 3, pp. 224–253, 1999. 22, 24, 97

[43] D. Cingolani, A. Pellegrini, and F. Quaglia, "Transparently mixing undo logs and software reversibility for state recovery in optimistic pdes," in *Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pp. 211–222, ACM, June 2015. 23, 24, 62, 64, 97

[44] H. Rajaei, R. Ayani, and L.-E. Thorelli, "The local time warp approach to parallel simulation," in *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, PADS, (New York, NY, USA), pp. 119–126, ACM, 1993. 23

[45] A. C. Palaniswamy and P. A. Wilsey, "Adaptive bounded time windows in an optimistically synchronized simulator," in *Third Great Lakes Symposium on Design Automation of High Performance Systems*, pp. 114–118, 1993. 23

[46] J. Wang and C. Tropper, "Selecting {GVT} interval for time-warp-based distributed simulation using reinforcement learning technique," in *SpringSim '09: Proceedings of the 2009 Spring Simulation Multiconference*, pp. 1–7, Society for Computer Simulation International, 2009. 24

[47] M. Lees, B. Logan, C. Dan, T. Oguara, and G. Theodoropoulos, "Decision-theoretic throttling for optimistic simulations of multi-agent systems," in *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT, (Washington, DC, USA), pp. 171–178, IEEE Computer Society, 2005. 24

[48] F. Quaglia, "A restriction of the elastic time algorithm," *Information processing letters*, vol. 83, no. 5, pp. 243–249, 2002. 24

[49] A. Ferscha, "Probabilistic Adaptive Direct Optimism Control in Time Warp," in *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pp. 120–129, IEEE Computer Society, 1995. 24, 39

[50] M. Lees, B. S. Logan, and G. K. Theodoropoulos, "Analysing probabilistically constrained optimism," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 11, pp. 1467–1482, 2009. 24, 42

[51] D. R. Jefferson and P. D. Barnes, "Virtual time III: Unification of conservative and optimistic synchronization in parallel discrete event simulation," in *2017 Winter Simulation Conference*, WSC, pp. 786–797, IEEE, dec 2017. 24

[52] D. R. Jefferson, "Virtual time II: storage management in conservative and optimistic systems," in *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, (New York, NY, USA), pp. 75–89, ACM, 1990. 24

[53] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 423–434, aug 1993. 24

[54] R. M. Fujimoto and M. Hybinette, "Computing Global Virtual Time in Shared-Memory Multiprocessors," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 4, pp. 425–446, 1997. 24, 27

[55] A. Pellegrini and F. Quaglia, "Wait-Free Global Virtual Time Computation in Shared Memory Time-Warp Systems," in *Proceedings of the 26th International Conference on Computer Architecture and High Performance Computing*, SBAC-PAD, IEEE Computer Society, 2014. 24, 27

[56] M. Ianni, R. Marotta, A. Pellegrini, and F. Quaglia, "A non-blocking global virtual time algorithm with logarithmic number of memory operations," in *Proceedings of the 21st IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, DS-RT, IEEE Computer Society, Oct. 2017. Candidate for (but not winner of) the Best Paper Award. 24

[57] D. E. Martin, T. J. McBrayer, and P. A. Wilsey, "WARPED: A Time Warp simulation kernel for analysis and application development," in *Proceedings of the 29th Hawaii International Conference on System Sciences. Volume 1: Software Technology and Architecture*, HICSS, pp. 383–386, IEEE Computer Society, 1996. 26, 27, 115

[58] C. D. Carothers, D. W. Bauer, and S. Pearce, "ROSS: a High Performance Modular Time Warp System," in *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pp. 53–60, IEEE Computer Society, 2000. 26, 111

[59] A. Pellegrini, R. Vitali, and F. Quaglia, "The ROme OpTimistic Simulator: Core internals and programming model," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools, ICST, 2011. 26, 104

[60] S. R. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette, "GTW: a time warp system for shared memory multiprocessors," in *Proceedings of the 26th conference on Winter simulation*, WSC, pp. 1332–1339, Society for Computer Simulation International, 1994. 26

[61] B. Wang, J. Himmelspach, R. Ewald, Y. Yao, and A. M. Uhrmacher, "Experimental analysis of logical process simulation algorithms in JAMES II," in *Proceedings of the 2009 Winter Simulation Conference*, WSC, pp. 1167–1179, 2009. 26

[62] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. W. LIFT, "A low-overhead practical information flow tracking system for detecting general security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 135–148, 2006. 26

[63] L. Chen, L. Wu, S. Peng, Y. Lu, and Y. Yao, "A well-balanced time warp system on multi-core environments," in *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, vol. 00 of *PADS*, pp. 1–9, 06 2011. 26, 113

[64] D. Jagtap, N. B. Abu-Ghazaleh, and D. Ponomarev, "Optimization of parallel discrete event simulator for multi-core systems," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 520–531, May 2012. 26, 111

[65] R. Vitali, A. Pellegrini, and F. Quaglia, "Assessing load sharing within optimistic simulation platforms," in *Proceedings of the 2012 Winter Simulation Conference*, WSC, Society for Computer Simulation, Dec. 2012. 26

[66] S. Conoci, D. Cingolani, P. Di Sanzo, A. Pellegrini, B. Ciciani, and F. Quaglia, "A Power Cap Oriented Time Warp Architecture," in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, ACM, 2018. 27

[67] A. Park and R. M. Fujimoto, "Efficient master/worker parallel discrete event simulation," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS, (Washington, DC, USA), pp. 145–152, IEEE Computer Society, 2009. 27

[68] J. Wang, D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, "Parallel discrete event simulation for multi-core systems: Analysis and optimization," *IEEE*

Transactions on Parallel and Distributed Systems, vol. 25, no. 6, pp. 1574–1584, 2014. 27, 111, 112

[69] F. Quaglia, "A low-overhead constant-time lowest-timestamp-first CPU scheduler for high-performance optimistic simulation platforms," *Simulation Modelling Practice and Theory*, vol. 53, pp. 103–122, 2015. 27

[70] Y. Lin and E. D. Lazowska, "Effect of process scheduling in parallel simulation," *Int. Journal in Computer Simulation*, vol. 2, no. 1, 1992. 27

[71] F. Quaglia, A. Pellegrini, and R. Vitali, "Reshuffling PDES platforms for multi/many-core machines: a perspective with focus on load sharing," in *Modeling and Simulation-based Systems Engineering Handbook* (D. Gianni, A. D'Ambrogio, and A. Tolk, eds.), Crc Pr I Llc, Dec. 2014. 27

[72] J. Wang, N. Abu-Ghazaleh, and D. Ponomarev, "Air: Application-level interference resilience for pdes on multicore systems," *ACM Transactions on Modeling and Computer Simulation*, vol. 25, pp. 19:1–19:25, Apr. 2015. 27, 113, 114

[73] T. Oguara, D. Chen, G. K. Theodoropoulos, B. S. Logan, and M. Lees, "An adaptive load management mechanism for distributed simulation of multi-agent systems," in *9th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT, pp. 179–186, 2005. 27

[74] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990. 29, 30

[75] R. P. Case and A. Padegs, "Architecture of the ibm system/370," *Commun. ACM*, vol. 21, pp. 73–96, Jan. 1978. 30

[76] E. H. Jensen, G. W. Hagensen, and J. M. Broughton, "A new approach to exclusive data access in shared memory multiprocessors," Nov 1987. 30

[77] M. P. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, 1991. 30, 31, 43

[78] M. Herlihy and N. Shavit, "On the nature of progress," in *Principles of Distributed Systems* (A. Fernàndez Anta, G. Lipari, and M. Roy, eds.), (Berlin, Heidelberg), pp. 313–328, Springer Berlin Heidelberg, 2011. 30

[79] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-Free Synchronization: Double-Ended Queues as an Example," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS, pp. 522—-, IEEE Computer Society, 2003. 31

[80] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition*. Morgan Kaufmann, 2012. 32

[81] D. J. Sorin, M. D. Hill, and D. A. Wood, "A Primer on Memory Consistency and Cache Coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, 2011. 32, 33

[82] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. 28, pp. 690–691, Sept. 1979. 32

[83] G. L. Peterson, "Concurrent Reading While Writing," *ACM Transactions on Programming Languages and Systems*, vol. 5, pp. 46–55, jan 1983. 33, 34

[84] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, (New York, NY, USA), pp. 73–82, ACM, 2002. 34

[85] J. P. Nielsen and S. Karlsson, "A scalable lock-free hash table with open addressing," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, (New York, NY, USA), pp. 33:1–33:2, ACM, 2016. 34

[86] A. Natarajan, L. H. Savoie, and N. Mittal, "Concurrent wait-free red black trees," in *Proceeding of the 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems - Volume 8255*, SSS 2013, (Berlin, Heidelberg), pp. 45–60, Springer-Verlag, 2013. 34

[87] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, (New York, NY, USA), pp. 131–140, ACM, 2010. 34

[88] J.-J. Tsay and H.-C. Li, "Lock-free concurrent tree structures for multiprocessor systems," in *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, (Washington, DC, USA), pp. 544–549, IEEE Computer Society, 1994. 34

[89] S. Gupta and P. A. Wilsey, "Lock-free pending event set management in Time Warp," in *Proceedings of the 2014 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pp. 15–26, ACM Press, 2014. 34, 45, 118, 119

[90] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A Non-Blocking Priority Queue for the Pending Event Set," in *Proceedings of the 9th ICST Conference of Simulation Tools and Techniques*, SIMUTools, ICST, 2016. 34, 56, 62

[91] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A lock-free o(1) event pool and its application to share-everything pdes platforms," in *Proceedings of the 20th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, DS-RT, IEEE Computer Society, Sept. 2016. Winner of the Best Paper Award. 34, 45, 46, 57, 117

[92] W. T. Tang, R. S. M. Goh, and I. L. Thng, "Ladder queue: An O(1) priority queue structure for large-scale discrete event simulation," *Transactions on Modeling and Computer Simulation*, vol. 15, no. 3, pp. 175–204, 2005. 34, 56, 118

[93] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, pp. 609–627, may 2005. 34, 117

[94] M. Ianni, A. Pellegrini, and F. Quaglia, "Anonymous readers counting: A wait-free multi-word atomic register algorithm for scalable data sharing on multi-core machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, pp. 286–299, 2018. 34

[95] A. Larsson, A. Gidenstam, P. H. Ha, M. Papatriantafilou, and P. Tsigas, "Multiword atomic read/write registers on multiprocessor systems," *Journal of Experimental Algorithmics*, vol. 13, no. 1, p. 1.7, 2009. 34

[96] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing* (J. Welch, ed.), vol. 2180 of *DISC*, pp. 300–314, Springer Berlin/Heidelberg, 2001. 35, 47, 49, 52, 94, 117

[97] R. M. Fujimoto, "Performance of Time Warp Under Synthetic Workloads," in *Proceedings of the Multiconference on Distributed Simulation*, pp. 23–28, Society for Computer Simulation, 1990. 43

[98] J. Hay and P. A. Wilsey, "Experiments with hardware-based transactional memory in parallel simulation," in *Proceedings of the 2015 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pp. 75–86, ACM Press, 2015. 45, 115, 116, 118

[99] S. Oh and J. Ahn, "Dynamic calendar queue," in *Proceedings 32nd Annual Simulation Symposium*, pp. 20–25, 1999. 55

[100] K. L. Tan and L.-J. Thng, "Snoopy calendar queue," in *Proceedings of the 32Nd Conference on Winter Simulation*, WSC '00, (San Diego, CA, USA), pp. 487–495, Society for Computer Simulation International, 2000. 55

[101] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 19:1–19:12, IEEE Press, 2008. 63

[102] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller, "Scheduling support for transactional memory contention management," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, (New York, NY, USA), pp. 79–90, ACM, 2010. 63

[103] S. Seelam, L. L. Fong, A. N. Tantawi, J. Lewars, J. Divirgilio, and K. Gildea, "Extreme scale computing: Modeling the impact of system noise in multicore clustered systems," in *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, pp. 1–12, 2010. 63

[104] S. Koenig and Y. Liu, "Terrain coverage with ant robots: a simulation study," in *Proceedings of the fifth international conference on Autonomous agents*, AGENTS, pp. 600–607, ACM, 2001. 69

[105] J. Svennebring and S. Koenig, "Building Terrain-Covering Ant Robots: A Feasibility Study," *Autonomous Robots*, vol. 16, no. 3, pp. 313–332, 2004. 69

[106] S. Bellenot, "State skipping performance with the Time Warp operating system.," in *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, PADS, pp. 53–64, 1992. 88

[107] R. Vitali, A. Pellegrini, and F. Quaglia, "Towards Symmetric Multi-threaded Optimistic Simulation Kernels," in *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, PADS, pp. 211–220, IEEE, 2012. 104, 113, 114

[108] D. W. Bauer Jr, C. D. Carothers, and A. Holder, "Scalable Time Warp on Blue Gene Supercomputers," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS, pp. 35–44, IEEE Computer Society, 2009. 111

[109] B. P. Swenson and G. F. Riley, "A New Approach to Zero-Copy Message Passing with Reversible Memory Allocation in Multi-core Architectures.," in *PADS*, pp. 44–52, 2012. 111, 112

[110] A. Pellegrini and F. Quaglia, "Numa time warp," in *Proceedings of the 2015 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pp. 59–70, ACM, June 2015. 112

[111] P. Putnam, P. A. Wilsey, and K. V. Manian, "Core frequency adjustment to optimize Time Warp on many-core processors," *Simulation Modelling Practice and Theory*, vol. 28, pp. 55–64, 2012. 112, 114

[112] A. Pellegrini, S. Peluso, F. Quaglia, and R. Vitali, "Transparent speculative parallelization of discrete event simulation applications using global variables," *International Journal of Parallel Programming*, apr 2016. 112

[113] A. Pellegrini and F. Quaglia, "Transparent Multi-Core Speculative Parallelization of DES Models with Event and Cross-State Dependencies," in *Proceedings of the 2014 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pp. 105–116, ACM, 2014. 112

[114] N. Marziale, F. Nobilia, A. Pellegrini, and F. Quaglia, "Granular Time Warp objects," in *Proceedings of the 2016 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, (New York, New York, USA), pp. 57–68, ACM Press, 2016. 112

[115] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, vol. 53, pp. 108–109, 2010. 114

[116] E. Santini, M. Ianni, A. Pellegrini, and F. Quaglia, "Hardware-transactional-memory based speculative parallel discrete event simulation of very fine grain models," in *Proceedings of the 22nd International Conference on High Performance Computing*, HiPC, pp. 145–154, IEEE, dec 2015. 115

[117] R. Ayani, "LR-Algorithm: concurrent operations on priority queues," in *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, SPDP, (Dallas, TX, USA), pp. 22–25, IEEE Computer Society, 1990. 117

[118] R. Rönngren and R. Ayani, "A Comparative Study of Parallel and Sequential Priority Queue Algorithms.," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, no. 2, pp. 157–209, 1997. 117

[119] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*, OPODIS 2013, (Berlin, Heidelberg), pp. 206–220, Springer-Verlag, 2013. 117, 118

[120] K. Zhang, Y. Zhao, Y. Yang, Y. Liu, and M. Spear, "Practical non-blocking unordered lists," in *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205*, DISC 2013, (New York, NY, USA), pp. 239–253, Springer-Verlag New York, Inc., 2013. 118