Sapienza University of Rome

Ph.D. program in Computer Engineering

XXVI Cycle

Techniques for Transparent Parallelization of Discrete Event Simulation Models

Alessandro Pellegrini

A.Y. 2013/2014

Sapienza University of Rome

Ph.D. program in Computer Engineering

XXVI Cycle

Alessandro Pellegrini

Techniques for Transparent Parallelization of Discrete Event Simulation Models

Thesis Committee	**Reviewers**
Prof. Francesco Quaglia (Advisor)	Prof. George F. Riley
Prof. Camil Demetrescu	Prof. Philip A. Wilsey

A.Y. 2013/2014

Author's address:
Alessandro Pellegrini
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Sapienza Università di Roma
Via Ariosto 25, 00185 Roma, Italy
e-mail: `pellegrini@dis.uniroma1.it`

www: `http://www.dis.uniroma1.it/~pellegrini/`

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

The work leading to this dissertation has been a long journey, which would not have been possible without the help of many. First and foremost, I would like to thank Prof. Francesco Quaglia, my advisor and my mentor. He has initiated me to the world of simulation, and has since then gone beyond the call of duty of his advising work. He has been a great fellow in thinking, discussing, realizing and assessing every proposal. The outcome of this work owes much to his enthusiastic encouragement, continuous guidance, and permanent availability, despite all. All the coherence that you will find in this work directly comes from his personal ability in climbing the trees and see the whole forest.

A special thanks goes to Roberto Vitali, an invaluable co-worker and a real friend. We have been working together since the time of my Bachelor's degree, we have spent many night asleep looking at cumbersome dumps to find subtle bugs in assembly code, and he has never held back to give unconditional help. All this would have never seen the light without him.

A word of gratitude goes to Prof. Bruno Ciciani. He has given me the opportunity to be involved in so many interesting projects, which were aside from the main track of my research, and which therefore gave me a more thorough view on what is going on in the world. He has always emphasized that if your work is amazing, but you are not a good person, then your work is not valuable. This is a teaching that I will never forget.

To all the other guys in Room B120: Diego, Pierangelo, Roberto, and Sebastiano. You have always been a source of inspiration, and you have never missed the opportunity to criticize any work to make it better, or to look for a solution altogether, whatever the problem was. It has been a great experience to work with a group of very prepared people who enjoyed what we were doing. These years will be stuck in my head forever.

I am very grateful to my whole family: Giorgio, Norma, Marta, Andrea, and Francesca. They have always believed in me, and they have always encouraged me to keep on working. The nights spent asleep have been many, but they have been less hard, because you all were there supporting me!

Thanks to my old friends Giulia, Gian Marco, and Simone, the only ones forcing me to shut my computer down to go out and see the light of the sun.

I make no pretence about the efforts made, the fear of the Lord is the beginning of knowledge. He created everything with such complexity that, fortunately, Ph.D.'s can still be earned for some time to come.

Thanks again to those who helped me, supported me, taught me a lot, laughed with me, and kept me going. And to Marta for all of the above.

Alessandro Pellegrini

Abstract

Simulation is a powerful technique to represent the evolution of real-world phenomena or systems over time. It has been extensively used in different research fields (from medicine to biology, to economy, and to disaster rescue) to study the behaviour of complex systems during their evolution (*symbiotic simulation*) or before their actual realization (*what-if analysis*).

A traditional way to achieve high performance simulations is the employment of Parallel Discrete Event Simulation (PDES) techniques, which are based on the partitioning of the simulation model into Logical Processes (LPs) that can execute events in parallel on different CPUs and/or different CPU cores, and rely on synchronization mechanisms to achieve causally consistent execution of simulation events. As it is well recognized, the optimistic synchronization approach, namely the Time Warp protocol, which is based on rollback for recovering possible timestamp-order violations due to the absence of block-until-safe policies for event processing, is likely to favour speedup in general application/ architectural contexts.

However, the optimistic PDES paradigm implicitly relies on a programming model that shifts from traditional sequential-style programming, given that there is no notion of global address space (fully accessible while processing events at any LP). Furthermore, there is the underlying assumption that the code associated with event handlers cannot execute unrecoverable operations given their speculative processing nature. Nevertheless, even though no unrecoverable action is ever executed by event handlers, a means to actually undo the action if requested needs to be devised and implemented within the software stack.

On the other hand, sequential-style programming is an easy paradigm for the development of simulation code, given that it does not require the programmer to reason about memory partitioning (and therefore message passing) and speculative (concurrent) processing of the application.

In this thesis, we present methodological and technical innovations which will show how it is possible, by developing innovative runtime mechanisms, to allow a programmer to implement its simulation model in a fully sequential way, and have the underlying simulation framework to execute it in parallel according to speculative processing techniques. Some of the approaches we provide show applicability in either shared- or distributed-memory systems, while others will be specifically tailored to multi/many-core architectures.

We will clearly show, during the development of these supports, what is the effect on performance of these solutions, which will nevertheless be negligible, allowing a fruitful exploitation of the available computing power. In the end, we will highlight which are the clear benefits on the programming model that the model developer will experience by relying on these innovative solutions.

Chapter Organization

In Chapter 1, we introduce the research problems which we will address throughout this thesis. In Chapter 2 we specifically frame the research context in which

this thesis is inserted. Chapter 3 discusses relevant research literature results which will be used or advanced during this work.

In Chapter 4 we present an innovative transparent incremental state saving subsystem, which allows the user to transparently scatter the simulation state on dynamic memory, while enabling the simulation kernel to take benefit of incremental state saving for performance enhancement. Consistent interactions with the outside world (namely, the generation of consistent output from a speculative execution of the simulation model) is discussed in Chapter 5. Chapters 6 and 7 deal with transparent access to shared portions of the simulation state, the former targeting global variables, the latter cross-LP state dependencies.

In Chapters 8 and 9 we highlight which are the benefits from the proposed supports. In Chapter 10 we draw conclusions and open up for future improvements in this, and other, research contexts.

Appendix A and Appendix B provide technical details on supporting software which has been developed to implement all the proposed advancements in this thesis.

CHAPTER 1

Introduction

"Where shall I begin, please your Majesty?" he asked.
"Begin at the beginning", the King said, gravely,
"and go on till you come to the end: then stop"
— Lewis Carroll, Alice's Adventures in Wonderland, 1865

Over the past 45 years, the total number of transistors available on a microchip has doubled every 18–24 months, a trend which is known as *Moore's law* [109]. This yielded a proportional increase in a single processor's clock speed which, in the past decades, was bringing an enhancement in the computing speed which researchers, developers and users were receiving for free, whenever they were upgrading their hardware. Improvements in algorithms and/or code optimizations were not strict requirements to be pursued, because—as the time was passing by—the software was just working more and more efficiently.

Nowadays, while we are transitioning from petascale to exascale systems, we are facing new challenging issues. In fact, even though the previous transition (from terascale to petascale) could directly benefit from the aforementioned implications of Moore's law, we are now hitting limits imposed by fundamental

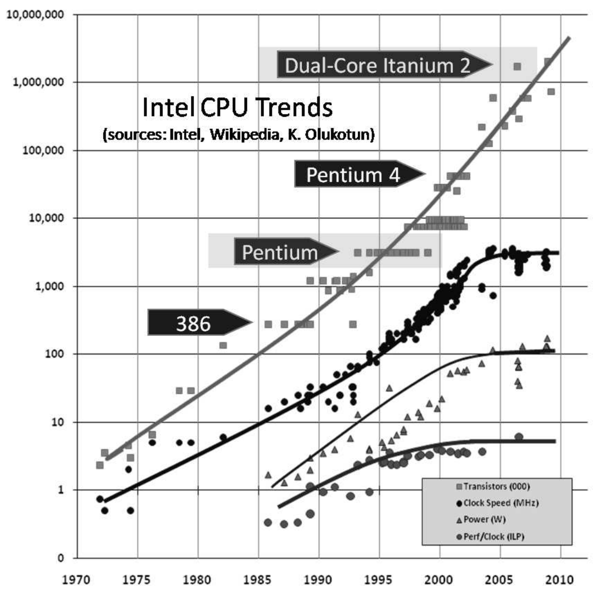


Figure 1.1: CPU specifications trend[1]

physics, as shown by the *dynamic power equation* [180]:

$$P = ACV^2 f \tag{1.1}$$

where P is the power consumption, A is the activity factor (i.e., the fraction of the circuit that is switching), C is the switched capacitance, V is the supply voltage, and f is the operating frequency.

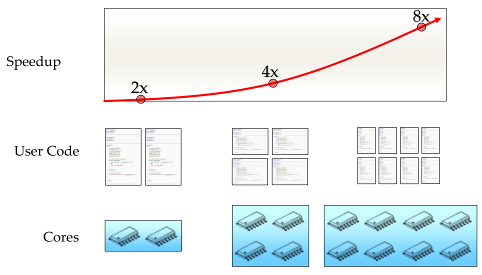
Historically, MOSFET technology was able to scale because, while increasing the total number of transistors per chip, it was decreasing their size and capacitance. At the same time, an increase in the overall frequency was counterbalanced by a decrease in the supply voltage. This process, known as *Dennard scaling* [33], was actually the electrical basis for Moore's law. In Figure 1.1 we see the trend of off-the-shelf processing units during the last 45 years. Moore's

[1]Figure taken from [160].

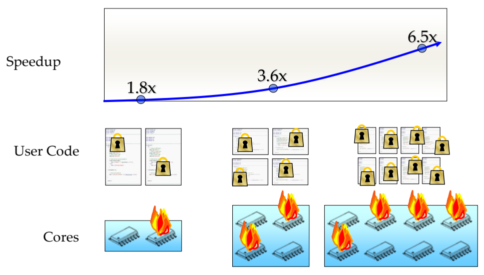
law's effect is clearly visible in the first part, as the number of transistors in a CPU keeps doubling, and the frequency follows the same trend. Yet, around year 2003 something happened: Although the number of transistors presents the same trend, clock speed increase has stalled. This is connected to the fact that switching noise in the circuits poses a limit to supply voltage decrease, and current leakage (due to the extremely small size of transistors) causes the chip to heat up, requiring a flattening in the clock frequency to keep Equation (1.1) in balance [112]. In fact, 130 W of power consumption in a processor is considered an upper bound, the so-called *clock-frequency wall* [160]: an increase in the clock frequency would create an unacceptable power consumption.

The industry has therefore approached physical limits in the computing power of a single processing unit, although the number of transistors per area unit is still increasing. Nevertheless, the demand for continued improvements in computing speed is still there, and this is driving hardware manufacturers to switch to the multicore technology, where chips with multiple processing units are being produced.

While it is interesting to note that Moore's law is still present (as it's now describing the number of cores in a multicore CPU [11]), it's important to emphasize that this fairly new technology paves the way to a massive usage of concurrent programming, and requires at the same time a global rethinking of the mechanisms and methodology used to build up parallel applications. The naïve approach to parallel programming sees the user simply partitioning her code and making it run on different processing unit instances (let them be cores or parallel CPUs). What the user may expect is that, like in the past years, doubling the computing power (which today means doubling the amount of CPUs/ CPU cores) consequently produces a proportional increase in the speed. This



(a) Expected Speedup



(b) Real Speedup

Figure 1.2: Parallel Speedup

scenario is shown in Figure 1.2(a).

Unfortunately, this is not the real scale up: To ensure correctness of the result, a program cannot freely execute any instruction in parallel. In particular, every concurrent access to objects not being serialized by any underlying layer (e.g., memory accesses) might produce unexpected (inconsistent) results. To avoid this, several techniques have been proposed, the simplest of which is the *locking* primitive, which enables one instance of the parallel program to read or modify data only if no-one else is performing the same action at the same time. As presented in Figure 1.2(b), this produces a reduced performance increase, because many computing resources are burnt in locking operations to produce consistent results.

There is a (theoretical) maximum speedup that a program being parallelized can reach. This is stated by *Amdahl's Law* [2]: Every program has some fraction of its code (namely P) which can be run in parallel, since in that portion there is no conflict on data accesses. In the optimal case, given that we have S processing units which can run the program in parallel, the maximum achievable speedup

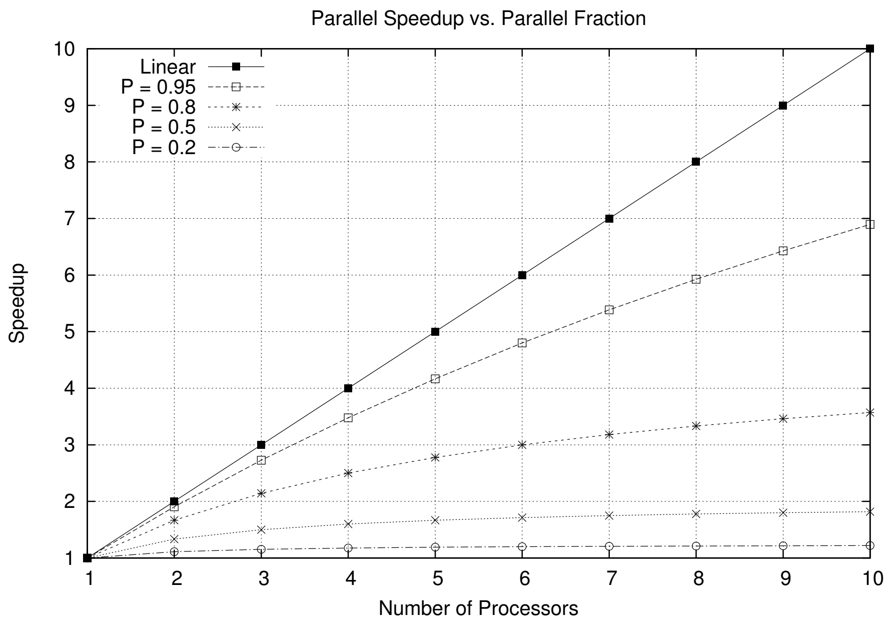


Figure 1.3: Amdahl's Law

for the entire program is:

$$S_{max} = \frac{1}{\left(1 - P\right) + \frac{P}{S}} \tag{1.2}$$

Usually, factor S is lower than the number of available processing units which can concurrently run the program, because of secondary effects and inter-process communication [15, 3]. As shown in Figure 1.3, if 80% of the program is parallelizable and S is equal to the number of processors, then the maximum achievable speedup is:

$$\lim_{S \to +\infty} S_{max} = \frac{1}{1 - 0.80} = 5 \tag{1.3}$$

Therefore, no matter how many processors are used, this program can never run more than five times faster.

The maximum theoretical speedup is anyway extremely difficult to reach. In fact, data accesses must be synchronized, and in order to efficiently do so, a common approach is to reduce the amount of synchronization points by decomposing the parallel tasks and by dividing the global set of data structures, so that the so-defined subtasks and subportions of data structures can be mapped onto the different processing units. This approach, often referred to as *data separation*, is recognized as an efficient way of designing parallel application, but it's not necessarily applicable to any sequential program. In any case, this methodology shows us that, in order to have an efficient parallel program, it is important to reduce synchronization points and therefore find the maximum value of P. In recent years, *non-blocking algorithms* [61, 62] have exacerbated this approach by creating extremely fine-grained synchronization points, by relying on atomic operations (like, e.g., compare-and-swap or load-link/store-conditional) provided by underlying hardware architectures. This way of designing parallel applications produces hardly understandable code, showing us that the most efficient serial algorithm is hardly the most efficient parallel algorithm, and yet the efficient parallel one is usually less comprehensible by humans.

Writing an efficient parallel application is time- and money-consuming, and requires skills which are prerogatives of a restricted set of experts. Yet, the need for parallel application is very compelling nowadays. It is therefore necessary to provide a methodology which is accessible to the masses. This is a very hot topic, and the academic community is pushing a lot towards the definition of new synchronization schemes, allowing the easy development of concurrent application. For example, a promising approach is Transactional Memory [63], which tries to alleviate the burden of implementing synchronization schemes by relying on the notion of *transaction*, and have both hardware [47] and software incarnations [149].

In this complex, unprecedented, quickly-evolving, and challenging scenario, a very interesting and important role is played by *simulation*. On the one hand, simulation is interesting because it offers a problem domain which often contains a high amount of parallelism to be exploited [97] (thus, giving us the possibility to find a value of P which is not minimal). On the other hand, the need for high-performance parallel simulation is extremely clear, as number-crunching applications require to process models which are continuously growing in both the number of parameters, and the size of the datasets. Additionally, there is a high demand for fast simulations, in order to make them applicable in context where timely production of simulation-outputs is critical for decision making, like e.g. *symbiotic systems* [5] and *what-if analysis* [142]. Furthermore, scientists approaching simulation are not necessarily computer scientists, as a large part of the models come from, e.g., the biology, medical, telecommunication, military, physics, economics, and business-oriented processes fields, just to mention a few. It is therefore a perfect context where to develop methodologies and frameworks to bridge the gap between the need for efficient parallelism exploitation, and the complexity of developing such an efficient application.

The goal of this thesis is to explore the simulation research field to provide methodological and technical solutions towards the generation of efficient parallel simulation models from sequential ones. In particular, a simple-yet-powerful programming paradigm oriented at the development of simulation models—namely *Discrete Event Simulation* (DES)—will be analysed: Some of its intrinsic properties will be selected to show how they can be directly exploited to support concurrent execution of simulation models. Others will be coupled with runtime supports in order to make them suitable for the automatic generation of parallel simulation models, from traditional (sequential) ones.

A central point of this thesis is to provide the reader with solutions which will enforce *transparency*, in the sense that (i) no modifications will be made by the application-model writer to the original (sequential) code to make it run in parallel, and (ii) common services for supporting efficient parallel simulation will be provided via means of specifically-targeted subsystems, allowing an experienced programmer to fully exploit the power of parallel simulations avoiding the burden to manually fine tune its code. In the end, this will increase the productivity of both categories of programmers, resulting (at the same time) in a faster development and a reduced execution time of simulation models, giving the opportunity to concentrate more on the actual model definition, and to cut the expenses related to model development and parallelization. Whether multi-core computing is a choice or a need, this thesis' results can provide the end user with supports that allow her to benefit from new-generation parallel computing architectures, with a much lower effort.

All the ideas which are gathered into this thesis, some of the preliminary results, or even some experiments which helped us shape in a better way the needs and the goals of this work, have been published in the following research works:

Book Chapters

[1] Francesco Quaglia, Alessandro Pellegrini and Roberto Vitali. Reshuffling PDES Platforms for Multi/Many-core Machines: a Perspective with focus on Load Sharing. In *Modeling and Simulation-based Systems Engineering Handbook*, Crc Pr I Llc, 2014. To appear.

Journal Articles

[2] Alessandro Pellegrini, Roberto Vitali and Francesco Quaglia. Autonomic State Management for Optimistic Simulation Platforms. In *IEEE Transactions on Parallel and Distributed Systems*, TPDS. To Appear.

[1] Roberto Vitali, Alessandro Pellegrini and Francesco Quaglia. Load sharing for optimistic parallel simulations on multi core machines. In *SIGMETRICS Performance Evaluation Review*, PER, vol. 40, issue 3, pp. 2–11, August 2012.

Conference Proceedings Papers

[17] Alessandro Pellegrini and Francesco 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, pages 105–116. ACM, May 2014.

[16] Alessandro Pellegrini and Francesco Quaglia. The ROme OpTimistic simulator: A tutorial (invited tutorial). In *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*, PADABS. LNCS, Springer-Verlag, August 2013.

[15] Alessandro Pellegrini and Francesco Quaglia. A study on the parallelization of terrain-covering ant robots simulations. In *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*, PADABS. LNCS, Springer-Verlag, August 2013.

[14] Alessandro Pellegrini. Hijacker: Efficient static software instrumentation with applications in high performance computing (poster paper). In *Proceedings of the 2013 International Conference on High Performance Computing & Simulation*, HPCS, pages 650–655. IEEE Computer Society, July 2013. Candidate for (but not winner of) the Outstanding Poster Paper Award.

[13] Francesco Antonacci, Alessandro Pellegrini, and Francesco Quaglia. Consistent and efficient output-stream management in optimistic simulation platform. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pages 315–326. ACM, May 2013.

[12] Alessandro Pellegrini and Giuseppe Piro. Multi-threaded simulation of 4G cellular systems within the LTE-Sim framework. In *Proceedings of the 8th IEEE International Workshop on the Performance Analysis and Enhancement of Wireless Networks*, PAEWN. IEEE Computer Society, March 2013.

[11] Pierangelo Di Sanzo, Francesco Antonacci, Bruno Ciciani, Roberto Palmieri, Alessandro Pellegrini, Sebastiano Peluso, Francesco Quaglia, Diego Rughetti, and Roberto Vitali. A framework for high performance simulation of transactional data grid platforms. In *Proceedings of the 6th ICST Conference of Simulation Tools and Techniques*, SIMUTools. ICST, March 2013.

[10] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. A load sharing architecture for optimistic simulations on multi-core machines. In *Proceedings of the 19th International Conference on High Performance Computing*, HiPC. IEEE Computer Society, December 2012.

[9] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Assessing load sharing within optimistic simulation platforms (invited paper). In *Proceedings of the 2012 Winter Simulation Conference*, WSC. Society for Computer Simulation, December 2012.

[8] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Transparent and efficient shared-state management for optimistic simulations on multi-core machines. In *Proceedings 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 134–141. IEEE Computer Society, August 2012.

[7] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th International Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 211–220. IEEE Computer Society, August 2012.

[6] Roberto Vitali, Alessandro Pellegrini, and Gionata Cerasuolo. Cache-aware memory manager for optimistic simulations. In *Proceedings of the 5th ICST Conference of Simulation Tools and Techniques*, SIMUTools. ICST, March 2012. Winner of the Best Paper Award.

[5] Alessandro Pellegrini, Roberto Vitali, and Francesco 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.

[4] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. An evolutionary algorithm to optimize log/restore operations within optimistic simulation platforms. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools. ICST, 2011.

[3] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Autonomic log/restore for advanced optimistic simulation systems. In *Proceedings of the Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 319–327. IEEE Computer Society, 2010.

[2] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Benchmarking memory management capabilities within ROOT-Sim. In *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, DS-RT. IEEE Computer Society, 2009.

[1] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 45–53. IEEE Computer Society, 2009. Candidate for (but not winner of) the Best Paper Award.

Other Publications During my Ph.D. course, I have published together with colleagues the following articles, which are not strictly related to the topic of this thesis, but which helped me to put some lessons learned into practice in different research fields.

[2] Diego Rughetti, Pierangelo Di Sanzo and Alessandro Pellegrini. Adaptive transactional memories: Performance and energy consumption tradeoffs. In *Proceedings of the Third IEEE Symposium on Network Cloud Computing and Applications*, NCCA. IEEE Computer Society, February 2014.

[1] Alice Porfirio, Alessandro Pellegrini, Pierangelo Di Sanzo, and Francesco Quaglia. Transparent support for partial rollback in software transactional memories. In *Proceedings of the International Euro-Par 2013 Conference*, Euro-Par. LNCS, Springer-Verlag, August 2013.

Chapter 2

Research Context and Achieved Results

Οὐδ' ὅλως εὐχερὲς οὔτε τῶν παρόντων ἐξηγήσασθαι
διὰ τὴν ποικιλίαν τῆς πολιτείας, οὔτε περὶ τοῦ μέλλοντος προειπεῖν
(It is neither an easy matter to describe their present state, due to the complexity of their constitution, nor to speak with confidence of their future)
— Polybius, Histories

The research context into which this thesis is plunged is *simulation*, which (from the Latin *simulare*, to "fake" or to "replicate") is the imitation of a real-world process' or system's operation over time. In particular, we focus on Discrete Event Simulation (DES), which is a simulation methodology expressing the behaviour of a given system using a sequence of *discrete* events during time. Events can only occur at a particular time instant, and they produce a change in the *simulation state* of the system [139].

An event is said to be discrete because its duration is *impulsive* (i.e., the time associated with its beginning corresponds to the time associated with its ending). Therefore, between two consecutive events the simulation state does not change, and the *simulation time* “hops” from one time instant to another. In particular, the simulation time advances according to the *timestamp* of each event that “happens” during the simulation.

This approach to simulation is therefore clearly different from *continuous simulation*, where the evolution of the simulation state (and therefore of the system) continuously tracks the system’s response over time according to a set of equations, typically involving differential equations.

There are two main approaches which can be used to shape DES, one consisting in a *formalism* for modelling and analysis of discrete-event systems, and one consisting in a set of systemic tools and methodologies to support the actual execution of a simulation model. In this chapter, both ways of addressing DES will be presented, showing the respective strengths and weakness. The original DES approach will later be transferred to parallel architectures, introducing the Parallel Discrete Event Simulation (PDES) methodology, which will be used throughout this thesis to build the automatic parallelization of simulation models technique. Finally, the results achieved within this thesis will be presented, pointing to the specific chapters which will address them.

2.1 Formal Definition of DES Models

A formal definition of simulation models can efficiently rely on *Discrete Event Systems Specification* (DEVS), which is a set-theoretic formalism introduced in the early 70s [184], and can be seen as an extension of the Moore machine formalism [183]. While the Moore machine formalism is essentially a finite state automaton whose output is determined by its state only, DEVS (i) makes the output directly depend on the input, (ii) associates a lifespan with each state, and (iii) associates a hierarchical concept with an operation, called *coupling*.

Basically, DEVS[1] provides a formalism used both to design hierarchically decomposable discrete-event models and to have a general understanding of discrete event systems, decoupling them from the computer-generated models. At the same time it provides a framework for model generation and execution via its abstract simulator concepts. The formalism defines a basic DEVS model (namely the *atomic model*) to be a structure:

$$M = \langle X, S, s_0, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle \tag{2.1}$$

where we have:

X: a set of external events;

S: a set of sequential states. A state's definition can be extended with the σ state variable, which tells the maximum time spent in a state when no external events are received, before triggering an internal transition.

s_0: the initial simulation state;

Y: a set of output events;

$\delta_{int} : S \to S$: the internal transition function, which specifies to which next state the system will transit after the steady time specified for a given state without the arrival of an external event has elapsed;

$\delta_{ext} : Q \times X \to S$: the external transition function, where $Q = \{(s, e) | s \in S, 0 \le e \le ta(s)\}$, and e is the elapsed time since the last state transition, which specifies how the system changes state when an input is received;

$\lambda : S \to Y$: the output function, which generates an external output just before an internal transition takes place;

$ta : S \to R_{0\to\infty}$: time advance function, which determines the permanence time in the state. Once the time assigned to the state is consumed, an internal

[1]A thorough description of the formalism and its semantics can be found in [185, 186].

transition is triggered. When the σ variable is present in the state, the amount of time specified by σ is returned.

A model designed according to the DEVS formalism expressed by Equation (2.1) transits along the states in S via its transition functions. If no events occur, the time advances according to the ta function applied to the current state. A new state is determined by δ_{int} applied to the old state. Output events are generated by the model right before an internal transition takes place. The function δ_{ext} produces a state transition if an external event occurs, applied on the old state, the time spent in the old state, and the external event itself.

As hinted, DEVS defines the concept of *coupled model* as well:

$$DN = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select\rangle \tag{2.2}$$

where

X_{self}: is set of external events handled by the coupled model;

Y_{self}: is set of output events handled by the coupled model;

D: a set of component references, i.e. the *name set* of sub-components of the model;

$\{M_i\}$: a set such that $\forall i \in D$: M_i is a component structure, i.e. a DEVS model definition;

$\{I_i\}$: a set such that $\forall i \in D \cup \{self\}$: I_i are the *influencees* of i, i.e. the set of external input couplings;

$\{Z_{i,j}\}$: a set such that $\forall j \in I_j$: $Z_{i,j}$ is the i-to-j output translation function, i.e. a function which maps events generated by one of the models in D to any other model in D;

select is the *tie-breaker function*, which defines how to select the event from the set of simultaneous events.

The structure is subject to the constraints that for $\forall i \in D$, the i-th model is defined according to Equation (2.1), i.e. like:

$$M_i = \langle X^i, S^i, s_0^i, Y^i, \delta^i, \lambda^i, ta^i \rangle$$

and that:

$I_i \subseteq D \cup \{self\}, i \notin I_i$ i.e. influencees of i must be taken only among the available model definitions, and i cannot influence itself;

$Z_{self,j} : X_{self} \rightarrow X_j$ i.e. external events handled by the coupled model can be translated to external events of model $j \in D$;

$Z_{i,self} : Y_i \rightarrow Y_{self}$ i.e. output events generated by any component structure $i \in D$ can be handled by the coupled model;

$Z_{i,j} : Y_i \rightarrow X_j$ i.e. output events generated by any component structure $i \in D$ can be translated to input events of any component model $j \in D$;

A coupled model, therefore, tells how to connect several component models together to form a new model. The latter model is a DEVS model itself (thanks to a closure property under coupling [185]), and can therefore be employed as a component in a larger coupled model. This means that component structured present in $\{M_i\}$ can be defined either according to Equation (2.1) or Equation (2.2). This is where the *hierarchical notion* of composable models comes into effect.

To give an example of application of the DEVS formalism, let's build the model description of a producer-consumer system, according to the diagram in Figure 2.1. The behaviour of the system is described by input and output events. In the example, input events are `?consumed` and `?produced`, while out-

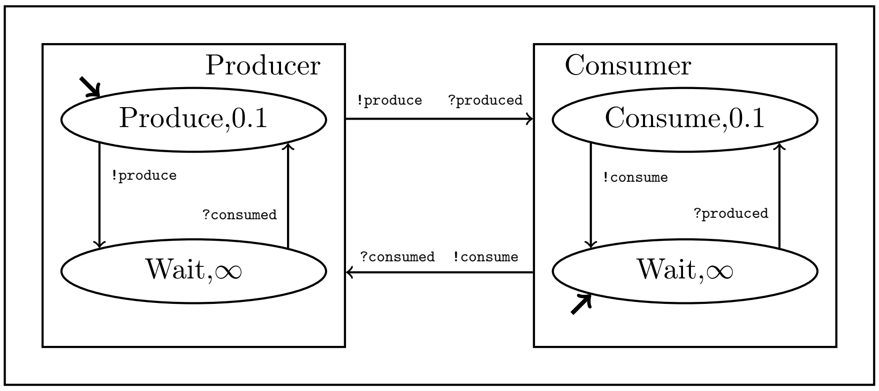


Figure 2.1: Sample DEVS model: Producer/Consumer

put events are `!produce` and `!consume`. Both the producer and the consumer have their states, namely *Produce*/*Wait* and *Consume*/*Wait*. Producer starts the simulation in the *Produce* state, while Consumer in the *Wait* state. *Produce* takes 0.1 seconds to produce an item, and similarly *Consume* takes 0.1 to consume it. When an item is produced, the output event `!produce` is sent out; when the item is consumed, the output event `!consume` is similarly sent out.

The goal of this simulation model is to interconnect two sub-models (namely, the producer and the consumer) into a larger simulation model. In order to formalize the producer and the consumer, two *atomic* DEVS models can be specified, by relying on the formalism expressed in Equation (2.1):

$$Producer = \langle X^P, S^P, s_0^P, Y^P, \delta_{int}^P, \delta_{ext}^P, \lambda^P, ta^P \rangle \tag{2.3}$$

where:

$$X^P = \{?\texttt{consumed}\}$$
$$Y^P = \{!\texttt{produce}\}$$

$$
\begin{aligned}
S^P &= \{(d,\sigma)|d \in \{Produce, Wait\}, \sigma \in [0,\infty]\} \\
s_0^P &= (Produce, 0.1) \\
ta^P(s) &= \sigma, \forall s \in S \\
\delta_{ext}^P(((Wait,\sigma),t_e), ?\texttt{consumed}) &= (Produce, 0.1) \\
\delta_{int}^P(Produce,\sigma) &= (Wait,\infty) \\
\delta_{int}^P(Wait,\sigma) &= (Produce, 0.1) \\
\lambda^P(Produce,\sigma) &= !\texttt{produce} \\
\lambda^P(Wait,\sigma) &= \emptyset
\end{aligned}
$$

and similarly:

$$Consumer = \langle X^C, S^C, s_0^C, Y^C, \delta_{int}^C, \delta_{ext}^C, \lambda^C, ta^C \rangle \tag{2.4}$$

where:

$$
\begin{aligned}
X^C &= \{?\texttt{produced}\} \\
Y^C &= \{!\texttt{consume}\} \\
S^C &= \{(d,\sigma)|d \in \{Consume, Wait\}, \sigma \in [0,\infty]\} \\
s_0^C &= (Consume, 0.1) \\
ta^C(s) &= \sigma, \forall s \in S \\
\delta_{ext}^C(((Wait,\sigma),t_e), ?\texttt{produced}) &= (Consume, 0.1) \\
\delta_{int}^C(Consume,\sigma) &= (Wait,\infty) \\
\delta_{int}^C(Wait,\sigma) &= (Consume, 0.1) \\
\lambda^C(Consume,\sigma) &= !\texttt{consume}
\end{aligned}
$$

$$\lambda^C(Wait, \sigma) = \emptyset$$

The final (larger) simulation model can be therefore expressed by merging the atomic models expressed by equations (2.4) and (2.3) as a coupled DEVS model according to Equation (2.2):

$$DN = \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select\rangle$$

where:

$$\begin{aligned}
X_{self} =& \{\}\\
Y_{self} =& \{\}\\
D =& Producer, Consumer\\
M_{Producer} \text{ and } M_{Consumer} & \text{ are defined according to Equations (2.4) and (2.3)}\\
I_i =& \{\}\\
Z_{i,j} =& \{(Producer.!\texttt{produce}, Consumer.?\texttt{produced}),\\
& (Consumer.!\texttt{consume}, Producer.?\texttt{consumed})\}
\end{aligned}$$

Although it might look cumbersome to rely on a formal definition to describe such an easy model, there are two main advantages:

1) The model definition can undergo a *verification* and *validation* process, to check if it is accurate and credible [148, 87]. This is a very important factor, considering that simulation models are just an imitation of the real world phenomenon, and cannot exactly reproduce it. Therefore, depending on the actual purpose of the application, it should be verified and validated to the needed degree;

2) The model definition can be transformed into an actual computer model, which can be run by a simulation algorithm. This can be a fully-automated or a user-aided process, depending on the typology of the model.

As for point 1, when a given DEVS model falls in DEVS extensions like Schedule-Preserving DEVS (SP-DEVS) [70], Finite & Deterministic DEVS (FD-DEVS) [71], or Finite & Real-time DEVS (FRT-DEVS) [69], it is formally proven that a behaviourally isomorphic finite structure can be derived from the infinite state structure of the original model. This implies that a reachability graph can be derived from the structure, allowing, e.g., to decide whether the model suffers from deadlock and/or livelock [68, 71, 69]. In the case of SP-DEVS, it is possible as well to define minimum and maximum execution time bounds [70].

Concerning point 2, in order to transform the formal definition of a model into an actual computer simulation, DEVS handles atomic and coupled models in a different way. In particular, the simulation of an atomic model is carried out by a *simulator*, while the simulation of a coupled model is carried out by a *coordinator*. The coordinator's main role is to enforce *time synchronization* and support *message propagation*. The former controls the advancement of the simulation time in all the atomic models, in order to always have them aligned. The latter transmits a triggering message (either input or output) along the associated couplings, which are defined in the coupled DEVS model. As an example, the CD++ simulation toolkit [175] allows many simulation models to be automatically defined. In this way, the construction of new models is simplified, along with their validation and verification.

Overall, this formal approach to discrete-event simulation has the benefit that the simulation model can "exist" even though there is no actual implementation of it. This allows the simulation model writer to perform behavioural analysis of the model itself, so that its properties can be studied/modified before the actual implementation is realized. Nevertheless, such analysis is difficult, and therefore «*direct computer simulation will remain a primary means of generating, and studying, model behaviour*»[187].

2.2 Systemic Approach to DES

The systemic approach to DES tackles the discrete-event simulation field with a more pragmatic approach, concentrating much more on model development supports and on the internal implementation of the software taking care of the simulation, usually referred to as *simulation kernel* or *simulation core*. In this context, although many aspects are directly borrowed from the formal DEVS definition of the model, much more effort has been put in the development of simulation techniques aiming at maximizing the overall execution throughput. From this point of view a model is seen as:

- the joint union of a *simulation states*, where the variables that keep track of the studied system's evolution are stored;
- a set E of events, which cause modifications to the state and describe parts of the real-world phenomenon being studied;
- a transition function $\sigma(s, e) : S \times E \to S$ which determines the actual transition from a simulation state s to a simulation state s' whenever an event $e \in E$ is executed.

Each (impulsive) event e is associated with a timestamp T_e which allows the global simulation time to advance. It is important, in this context, to make a strong distinction between the concepts of *simulation time* (ST) and *wall-clock time* (WCT). The former describes the logical time associated with the simulation model and can be therefore expressed in any time unit (e.g., hours, day, years), depending on the phenomenon being described by the model. On the other hand, the latter is the actual notion of time that we, human beings, are familiar with. Simulation time is then used to describe the evolution of the under-study phenomenon, while wall-clock time allows to determine the actual speed of the simulation execution.

Traditional DES implementations directly borrow from the more broader *event-driven programming* paradigm, where the flow of the program and the actions taken are determined by events captured by the application, resembling what happens with *interrupts* in computing architectures and operating systems. This paradigm has seen many applications, from sensors networks to (more modern) graphical user interfaces. Similarly, a DES model can be seen as a set of *event handlers*[2], which capture the events generated by the same application (i.e., the simulation model) and, depending on the nature of the event, produce a state variation. In fact, the nature of the aforementioned transition function $\sigma(s, e)$ is completely described by the set of event handlers and the operations performed on the state by each of them.

During the execution of an event e, new events $e', e'', \dots, e^n$ can be generated. This is related to the fact that the simulation model captures the *causality* between operations. In particular, if an event e' is generated during the execution of event e, it means that event e' *causally depends* from e, as it describes a portion of the system's functioning which is strictly related to the operations described by event e. It is important to emphasize that due to the temporal nature of simulation models, and due to the causal correlation of events, any event e' generated during the execution of e must be associated with a times-

[2]An event handler is essentially an *asynchronous callback*, which is used by the simulation kernel to pass control to the model's code, delivering a piece of application-level information (i.e., an event).

tamp $T_{e'} \geq T_e$. From a model point of view, this means that an action in the present cannot affect the past, which is clearly an acceptable assumption.

It is interesting to note that, similarly to the formal approach, this systemic vision of DES can perfectly split a simulation into two components: the simulation model and the simulation kernel. Although there is no standard definition of the components of a DES kernel, practice tells us that there is a minimum set of basic components which are essential. Additionally, due to the event-based nature of the simulation models, the totality of the simulation kernels rely on a *simulation loop*[3] in order to make the simulation progress, although different implementation might use different incarnations of it.

Since one of the simulation kernel's duties is to manage events injected into the system, whenever a simulation model wants to schedule a new event, this has been traditionally achieved by using a provided ad-hoc API. Therefore, whenever during the execution of event e two new events e' and e'' are generated, associated with timestamps $T_{e'} = T_{e''}$ (which is an acceptable causal dependency), the simulation kernel must decide the order according to which these two different events must be delivered to the proper event handlers. The problem of simultaneous events can be tackled by relying on a *tie breaking* function, exactly as the DEVS formalism does, but is non-trivial, as it can affect the behaviour of some simulation models [176], as it might introduce a bias towards some (logical) entity of the simulation model.

2.2.1 Basic Components of DES

Simulation State As shown before, the evolution of the simulation model depends on the state transitions produced by a transition function in the form

[3]This is an element that is directly borrowed from event-driven programming.

$\sigma(s,e) : S \times E \rightarrow S$. Therefore, a DES model cannot prescind from the presence of a simulation state. Historically, simulations states where regarded as a set of global variables which where explicitly defined within the model code and altered by the execution of events.

Events The evolution of a simulation model proceeds thanks to the execution of events. As stated before, an event is impulsive, i.e. it has no duration in time. A simulation model must define the logic associated with each event a-priori (i.e., no "unknown events" can be delivered to a simulation model), thus a closed set E of events that can be executed during the simulation run must be programmatically defined. Each event $e \in E$ is therefore handled by one of the event handlers registered in the system. Due to the *sequential* nature of DES, during the execution of an event e any variable belonging to the simulation state S can be freely accessed.

In order to give start to the simulation, an initial event (often referred to as `INIT`) must be scheduled. The typical behaviour of this event entails the set up of the simulation model, i.e. the definition of the simulation state's *initial conditions* and the *scheduling* of next event(s) to be processed. The `INIT` event is usually automatically generated by the simulation kernel, but due to its model-dependent nature, it must be explicitly managed by an event handler specifically implemented in the simulation model.

Clock Since a simulation model describes the evolution of a system during time, the model must keep track of this advancement. The simulation clock is usually a global variable (which is often, but not necessarily, updated by the simulation kernel) which tracks this temporal evolution. It is important to emphasize that the simulation clock tracks the evolution of the ST, not the WCT.

Therefore, given the nature of DES, its value does not continuously change, instead it jumps from the timestamp T_e of one event e to the timestamp $T_{e'}$ of the next event e'.

Event Queue During the execution of event e, the simulation model can generate any number of new events, depending on the actual model's logic. For example, given that the events are impulsive, activities that extend over time are often modelled as a sequence of differentiated events, which might be generated, e.g., during the first event of the sequence. This behaviour requires a data structure which takes care of managing generated (but still to be executed) events.

Since all the events are associated with their respective timestamps, the list of pending events must be ordered according to this value [59]. In this way, determining which is the next event to be processed is as easy as taking the event found on the head of the event queue. If the simulation kernel fails to do so, it means that the simulation might execute event e_x associated with timestamp $T_x > T_{min}$ and modify state variables which were needed by event e_{min}. This is like having the future which affects the past, and is clearly unacceptable: this kind of error is called *causality* error.

The actual implementation of the event queue is not necessarily a linked list, as due to performance requirements we want to reduce the time needed for insertion of events and selection of the next event to be processed [80]. Several proposals entail the adoption of skip lists [131], calendar queues [17], splay trees [153], or ladder queues [35].

Ending Condition A large number of simulation models describe phenomena which do not automatically halt (e.g., network traffic, chemical reactions, ...).

Furthermore, many models involve stochastic processes to describe the evolution of the system, making it impossible to predict before-hand how the simulation will evolve (isn't this the goal of simulation, anyway?!). In order to collect statistics which are meaningful for a specific experiment, it is important to configure the simulation so that when a particular ending condition is met, the simulation halts. This can be done by specifying a time range of interest for the simulation (this kind of ending condition can be handled by the simulation kernel), or by inspecting particular values of the simulation state along the simulation trajectory (this must necessarily be done by the simulation model, using specific API provided by the simulation kernel).

Simulation Object Although a simulation object is not mandatory for a DES model, it is an interesting extension which gives more semantic power to the model writer, and is therefore supported by a large number of simulation kernels. A simulation object describes a portion of the whole model, let it be a *spatial portion* (e.g., a "cell" in grid simulations, like military simulations [119]) or an *agent* (e.g., in agent-based social simulations [93], where simulation objects can model the behaviour of agents acting within the system). This allows the model writer to concentrate on the description of subportions of the whole model, and to link them together in the end, via the usage of special "interconnection events", much like what the $\{Z_{i,j}\}$ set of Equation (2.2) was thought for.

Additionally, the simulation state can be reorganized, so that each simulation object has its own set of state variables. As it will be shown in Section 2.3, this has been a fundamental step in literature to move DES to parallel/distributed architectures.

Algorithm 2.1 DES Skeleton

```
procedure INIT
    $End \leftarrow$ false
    initialize $State$, $Clock$
    schedule $INIT$
end procedure

procedure SIMULATION-LOOP
    while $End ==$ false do
        $Clock \leftarrow$ next event's time
        process next event
        Update Statistics
    end while
end procedure
```

2.2.2 Simulation Kernel's Basic Logic

As mentioned before, simulation kernels must allow the simulation model to set up the initial conditions of their state (i.e., they must generate an `INIT` event) and must handle the generation and execution of following events. The basic structure of a simulation kernel's implementation, cleaned from all possible optimizations and additional facilities, is shown in Algorithm 2.1.

Two main parts are essential. First, the Init procedure:

1) sets the ending flag to false, saying that the simulation will halt only when the ending condition is met;

2) initializes the simulation state, by allocating its memory, if needed by the simulation kernel implementation;

3) initializes the simulation clock, by setting its value to the initial simulation time, which is usually 0;

4) schedules the `INIT` event. This can be done by either calling the proper

event handler directly, or by placing the `INIT` event into the event queue, before entering the main loop.

Second, the main loop (which is entered right after having initialized the simulation) is essentially divided into two main phases:

1) Event selection: the smallest timestamped event e_{min} is selected from the event queue, in order to avoid causality violations;

2) Event handling: the clock is updated to the value T_{min} associated with event e_{min}, to reflect the advancement in ST; then the event is passed to the model, by calling the proper event handler, so that the logic associated with it is actually executed.

An additional phase, which is not mandatory, is *statistics update*, which involves logging, e.g., performance information about the simulation run, and collecting model-related data from the simulation state. The latter point can be done, of course, by triggering a specific function implemented within the simulation model.

2.3 Parallel Discrete Event Simulation (PDES)

The research in parallel and distributed simulation began in 1979 with a work by Chandy and Misra [25], and quickly gave birth to Parallel Discrete Event Simulation (PDES) [48], which refers to the execution of a single discrete event simulation program on a parallel/distributed system. This entails the "transformation" of a DES program into a PDES program. Due to the separate nature of a DES program into *model* and *kernel*, as described in Section 2.2, this can be done on the one hand by imposing a few restrictions on the model's definition, and on the other by significantly altering the simulation kernel. Of course, the advantage of this approach is that the largest part of DES models can be extremely easily ported to PDES systems, provided that they respect the restrictions.

PDES strongly relies on the notion of Simulation Objects, as described in Section 2.2.1, which here get the name of Logical Processes (LP), to emphasize the similarity with operating systems' processes, which (if they do not explicitly request the contrary) execute independently of each other and without any shared portion of memory. We can generally say that a simulation is composed by N LPs, each one uniquely identified by an integer number[4] in the range $[0, N-1]$. We will call them $LP_0, LP_1, \ldots, LP_{N-1}$. Each LP is associated with a *private clock*, which expresses the simulation time up to which it has progressed. The value of this private clock will be referred to as Local Virtual Time (LVT), emphasizing the fact that LP_i and LP_j have reached the different (local) ST values $LVT_i \neq LVT_j$.

The most important restriction to the implementation of a simulation model regards the usage of *shared variables*. In particular, the model programmer is requested to partition the whole simulation state S into per-LP subportions of the state S_i, which must guarantee the following property:

$$S = \bigcup_{i=0}^{N-1} S_i \qquad \wedge \qquad S_i \cap S_j = \emptyset, \forall i \neq j \tag{2.5}$$

Equation (2.5) tells us that the partitioning into sub-states S_i must not leave out any additional portion of the global simulation state, i.e. no global shared variables are allowed. Additionally, each LP cannot access any portion of another LP's state during the execution of one event. More specifically, inter-LP

[4]In case there is the need for a different representation of the LPs (e.g., symbolic names), a mapping function can be used to transform it into integers.

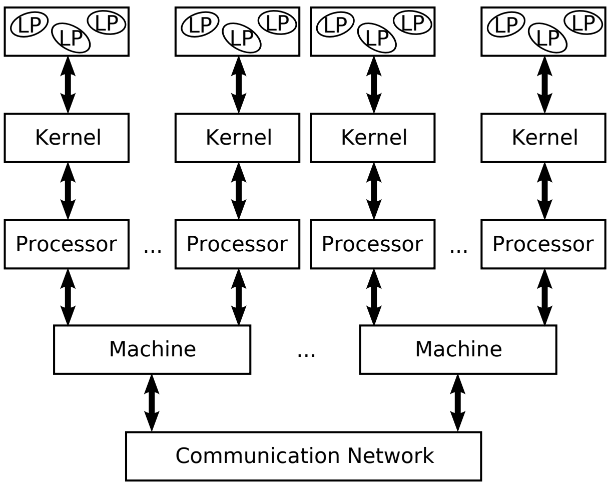


Figure 2.2: Parallel Discrete Event Simulator Classical Architecture

communication is only allowed via *event exchange*.

The exclusion of global variables is straightforward in some applications (like, e.g., queueing network simulations [151]), but can be burdensome in others. For example, in several *war game* simulations, where units move across a terrain and interact with each other only seldom (see, e.g., [182, 53]), the most immediate way to store the simulation state is to have a global shared matrix representing the state of each terrain grid. Although the grid might be replaced with a set of messages being exchanged to retrieve/update the state of a cell, doubts should arise about the performance and the scalability of this approach.

Provided that LPs do not share any portion of the simulation state, a classical (distributed) PDES simulation kernel's architecture is shown in Figure 2.2. Basically, the LPs are mapped to different *simulation kernel instances*, which are different user-space processes being run on top of a processing unit. Different instances located onto different machines are interconnected via a network. Therefore, simulation kernel instances hosted by the same machine can commu-

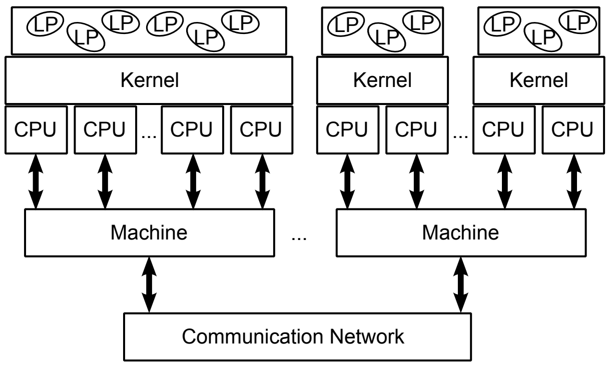


Figure 2.3: Parallel Discrete Event Simulator Multithread Architecture

nicate relying, e.g., on shared memory or inter-process communication facilities provided by the underlying operating system, while remote instances can rely on the *distributed memory* paradigm, which is turn is built on top of *message passing primitives*, like the message Passing Interface (MPI) protocol [110]. Each message (let it be passed via shared memory or the network) envelopes one event, so that in literature event exchange is often referred to as *message exchange* as well.

A more recent research trend, which we consider the standard reference architecture in this dissertation, is addressing the topic of reshuffling the traditional PDES architecture, to realize multithreaded simulation kernels [172, 30, 74]. This is a clear response to the architectural trend discussed in Section 1: If there current machines are mostly multi-core or SMP, then in order to exploit the available computing power a new technological paradigm should be enforced. Independently of the actual implementation, a general description of the new paradigm is shown in Figure 2.3.

By the picture, we can see that this new modern architecture has less kernel instances deployed on a single machine (that can be, nevertheless, more than one, as in the classical case) end each kernel instance is in charge of managing multiple processing units. To this end, kernel instances relies on the *worker thread* paradigm, where each thread implements a main simulation loop (as in the traditional DES case) but message (i.e., event) passing between locally hosted LPs can be implemented completely in user space, without requiring external libraries's support. This approach, although more complex in the implementation, has been proven to be more efficient and scalable with respect to the classical PDES implementation [169].

In this scenario, it is important to define which worker thread is in charge of dispatching a specific LP. In fact, having a completely symmetric approach, in which every worker thread can execute any LP, might affect performance due to locality effects. To this end, the concept of *LP binding* [170, 171] is essential. This concept defines virtual temporal windows which define a static coupling between the (locally) available LPs and the worker threads. This temporal window can be set to the whole simulation (i.e., a *static binding* is implemented), or can be periodically recomputed, in order to, e.g., level the workload on all the available worker threads and therefore reduce the rollback probability, which in turn enhances the overall performance. This is a technique called *load sharing* (see, again, [169, 170, 171]), which is different to the more traditional *load balancing*. The latter was specifically targeting the classical PDES architecture, and in order to reduce the rollback probability (due to imbalances in the workload) it supported the *migration* of LPs from one kernel instance to another (see, e.g., [128]). Migrating a LP is a costly operation, as it entails transferring (even on the network in case of remote kernel instances) the simulation state S_i as well.

Let us now discuss an aspect related to events exchange patterns in the multithreaded architecture depicted in Figure 2.3. As mentioned, each worker thread implements, de facto, the functionalities of a DES kernel, thus allowing a parallel execution of the simulation models. It therefore has its own basic components according to the description in Section 2.2.1. Let us assume that on a single (local) simulation kernel instance at a given instant of WCT LP_i (bound to worker thread k_0) and LP_j (bound to worker thread k_1) have reached, respectively, simulation time $LVT_i = 5$ and $LVT_j = 15$. Since worker thread k_0 and k_1 have different event queues[5], and since they both apply Algorithm 2.1 to carry on the simulation, they will select respectively events e^0_{min} and e^1_{min} as next events for LP_i and LP_j. Let us assume that the timestamps associated to the next events e^0_{min} and e^1_{min} are, respectively, $T_{e^0_{min}} = 10$ and $T_{e^1_{min}} = 20$. The execution of these events brings LP_i's and LP_j's clocks to the values 10 and 20, respectively. If, during the execution of e^0_{min} a new event e^0_{new} associated with timestamp $T_{e^0_{new}} = 12$ is generated, destined for LP_j, we incur in the situation where LP_j's clock value is 20, but an event at time 12 has not been executed yet. This "late" event is called a *straggler message*, and provides us with an example of *causal violation* generated by the parallel nature of PDES, which is depicted in Figure 2.4. This problem is not new, as the same situation might arise in the classical PDES implementation, where the same behaviour is exhibited by different (local) simulation kernel instances, rather than by different worker threads within a same simulation kernel instance.

2.3.1 The Synchronization Problem

The problem which we have spotted in Figure 2.4 is related to the parallel/ distributed nature of PDES. This problem, which might generate causality errors depending on the asynchronous (model and simulation-run dependent) pattern

[5]This is a general case. Common implementation of simulation kernels rely on multiple queues, e.g. one per LP, for performance reasons.

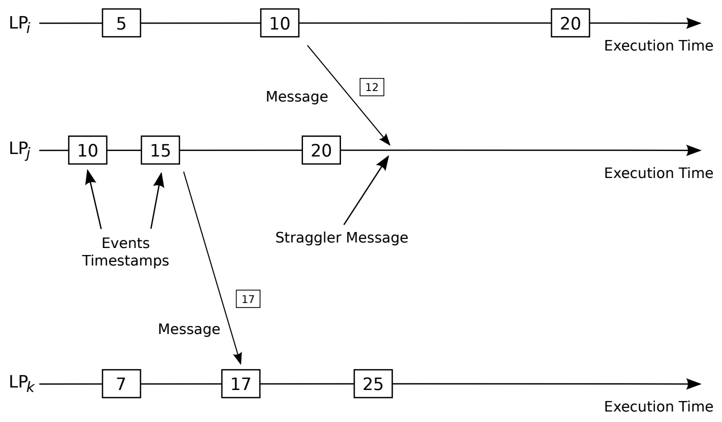


Figure 2.4: Event Causality Violation

of events execution, generation and delivery, is known as the *synchronization problem*.

To overcome this problem, and therefore enforce a correct simulation run which is independent of the asynchronous events-exchange pattern, different strategies have been discussed in literature [138, 48], which can be categorized into *conservative* [19, 25, 26], *optimistic* [76], and *hybrid* [155, 157] approaches. The first category tackles the synchronization problem by essentially avoiding the possible occurrence of causality errors a-priori (i.e., an event is executed only when it's considered to be *safe*), while the second category executes pending events independently of their *safety* and adopts a-posteriori strategies to detect causality errors and correct them. Since both approaches can show performance drawbacks, the third category essentially tries to determine which is, for a given simulation model, the best-suited approach and selects it.

Conservative Synchronization

Historically, conservative synchronization has been the first mechanism developed to enforce consistency of PDES systems. As mentioned, the goal is to

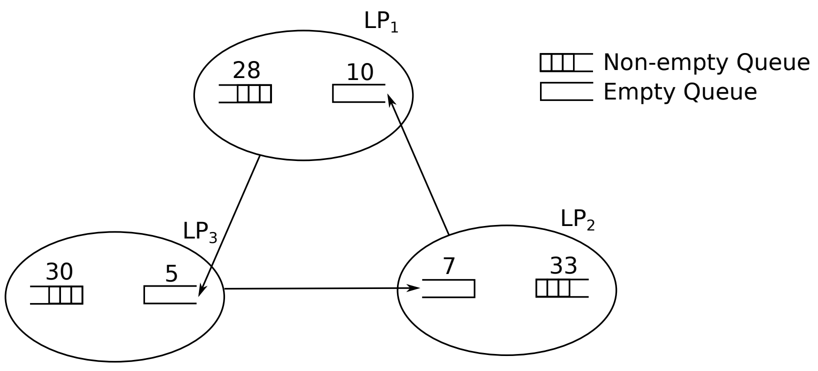


Figure 2.5: Deadlock in Conservative Synchronization

determine, among the events which are present in the event queue of one LP, which are *safe* to be executed. Then, simulation goes on by executing (among the safe ones) the event e_{next} which has the smallest timestamp [51].

The first approach, as shown in [19, 25], forced the model writer to explicitly indicate where each LP was communicating with other ones, by *statically* defining communication channels. Therefore, each LP can have more than one communication channel, and each one (which can be logically seen as a FIFO queue) is associated with a timestamp. The timestamp of a queue is either the timestamp of the message at the beginning of the queue, or the timestamp of the last executed event if the queue is empty.

Each LP selects the queue with the smallest timestamp to get an event to process. If the queue with the smallest timestamp is empty, then the execution blocks until an event is received into that queue. If queues Q_1 and Q_2 are at timestamp $T_{Q_1} < T_{Q_2}$ and only Q_1 is empty, then executing events at timestamp T_{Q_2} might produce causality errors because an event e at timestamp $T_{Q_1} \leq T_e < T_{Q_2}$ might be received, as this does not violate the FIFO ordering of Q_1. Then, blocking is a necessary (and sufficient) condition to avoid causality errors.

Nevertheless, this approach is prone to deadlocks. Let us assume that a simulation model is realized according to the scheme in Figure 2.5. LP_1 has a communication channel towards LP_3, which in turn has a communication channel towards LP_2. Each of the three LPs have one additional communication channel. Since all three queues associated with the three channels are empty, we have a deadlock situation. It does not help the fact that each LP has one additional queue which stores messages incoming from other LPs in the simulation, given that the timestamps of the empty queues are lower than the others. The selection algorithm, therefore, selects the empty queues as the ones to be checked for incoming events.

This deadlock situation can be avoided by relying on *null messages*. They are messages which are exchanged between LPs but do not carry any event to be processed. Rather, they are the "promise" that LP_i will not send any event associated with a timestamp $T_{null} > T_{Q_i}$, where Q_i is the queue where events sent by LP_i are stored. This means that by using null messages, the system is capable of determining that it is impossible for LP_i to receive any event e_j with a timestamp $LVT_i \leq T_j < T_{null}$. Any event e_k belonging to a different queue, with a timestamp $LVT_i \leq T_k < T_{null}$ is therefore safe, and can be executed.

More sophisticated techniques to avoid deadlock are presented in [4, 21, 27, 55, 56, 98, 117, 122, 137]. Nevertheless, a central aspect necessary for these approaches to work is that the system must know what is the minimum amount of ST that can be between two consecutive events in an immediate future. This is known as *lookahead* L, and if it can be correctly identified, then the LP_i which has reached the (system-wide) minimum ST LVT_i (i.e., the LP which has the smallest clock value among all the LPs in the system) can execute all the events associated with a timestamp in the interval $[LVT_i, LVT_i + L]$. Any other LP_j which has reached ST LVT_j such that $LVT_i < LVT_j < LVT_i + L$ can execute all the events associated with timestamps in the interval $[LVT_j, LVT_i + L]$ as well. The rule of thumb is that the larger the value of L, the higher the probability that all the LPs have some safe-event pool from which to select the next event to be executed. Nevertheless, the correct lookahead value is necessarily coupled with the simulation model, and must be therefore explicitly specified by the simulation-model writer, a process which can be extremely burdensome.

Additionally, if the lookahead value is too small (or even zero), the simulation might proceed too slowly, frustrating the goal of providing high-performance simulations by the usage of many computing resources. To this end, a proposal aiming at increasing the lookahead can be found in [115], where a part of the pending events chain is pre-computed, and at the end of the pre-computation the resulting state is checked to see whether it is still consistent or not. This is a technique that, although different in practice, is similar in spirit with optimistic synchronization, which will be discussed later.

The conservative synchronization protocol provides several advantages:

- it is *aggressiveless*: the simulation is carried on in such a way that an inconsistent simulation state is never reached. This is done exactly by ensuring that each simulation step does not cause any error condition;
- it is *riskless*: results injected into the system always relate to a consistent portion of the simulation trajectory, therefore no "uncertain" data is externalized to other components of the model;
- it requires *minimal synchronization* among various LPs: differently from what we will see in optimistic synchronization, ST always advances (i.e., it is never rewound). This ensures that detecting the global point in simulation at which all the LPs have progressed is a simple and non-costly operation.

On the other hand, conservative synchronization is not able in the majority of contexts to fully exploit the parallelism provided by the underlying architecture. In fact, if we take any two events e_1 and e_2 in the simulation, the conservative synchronization scheme might force their serial execution independently of whether they really have a direct (or indirect) dependency.

Optimistic Synchronization

Optimistic synchronization is based on the notion of *speculative processing*. This technique, which has been effectively employed in a large variety of areas, like branch prediction in pipelined processing units [154], in file prefetching [29], and in transactional systems' concurrency control [86], is based on the idea that whenever a computing resource is available, it should be used to perform a task which is *likely* correct and/or useful. Doing this work *before* it's actually known if it were to be done can provide benefits when it is later discovered that it was actually needed. In case that part of work is a-posteriori known to be incorrect or not useful, the result is simply discarded. The advantage comes from the fact that waiting to know if a task is necessary might produce a delay in the actual execution of the task itself. If the speculative processing is correct, i.e. the *guess* on the work to perform was correct, this delay can be even reduced to zero. In case of a wrong guess, the system pays the same delay as if it were not speculative, except for the cost of discarding the wrong piece of work carried out.

In PDES, this approach was presented (under the name of *optimistic synchronization*) in the seminal paper [76], where the *Time Warp* mechanism was introduced. Differently from the conservative approach, optimistic synchronization selects, at each simulation kernel instance and at each worker thread, the

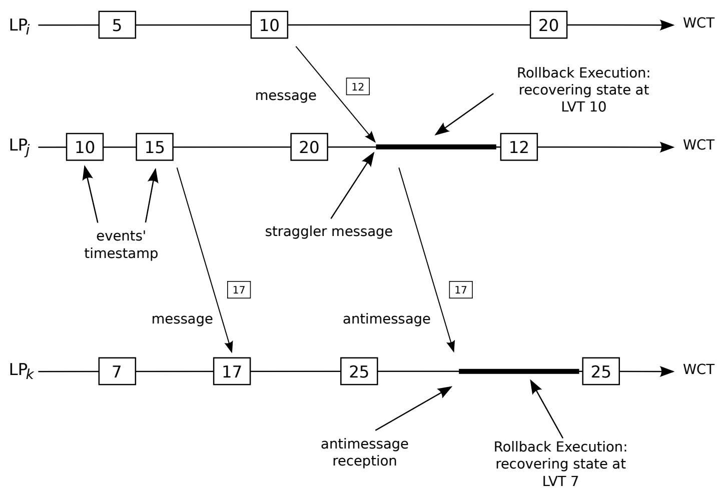


Figure 2.6: Rollback Operation

next event to be processed among the local ones independently of their safety. This clearly negates the guarantees provided by the conservative synchronization scheme, so that the system can fall into an inconsistent simulation state by following a simulation trajectory which is affected by a causal violation, as previously depicted in Figure 2.4.

Rollback Operation It is therefore necessary to support a mechanism which is able to a-posteriori detect the arrival of a straggler message and the occurrence of a causality violation, to temporarily halt the execution of the simulation model, to restore the simulation state to a previous (consistent) point in ST, and then to restart the *forward execution* of the model by taking into account the straggler message. This operation, known as the *rollback operation*, is depicted in Figure 2.6.

In the example provided by the picture we see that the rollback operation involves an additional problem. In particular, LP_j is at $LTV_j = 20$ when it receives a straggler message $e_{straggler}$ associated with timestamp $T_{straggler} = 12$.

Therefore, it rolls the execution back to ST $LVT_j = 10$, i.e. it discards the state changes which were operated by events at timestamps $T = 15$ and $T = 20$. This is sufficient for restoring a consistent state for LP_j, i.e. the subportion S_j of the global state is correctly restored to time $T = 10$. Nevertheless, we notice that during the execution of the event at timestamp $T = 15$, LP_j has sent a message to LP_k scheduled for ST $T = 17$. This message was the result of a wrong simulation trajectory, as LP_j reached $LVT_j = 15$ without taking into account the piece of information stored into event $e_{straggler}$. As LP_j has now rolled back to ST $LVT_j = 10$, it is possible that, upon the execution of $e_{straggler}$, the simulation will follow a different trajectory, involving the fact that either that message scheduled at ST $T = 17$ should have not been sent, or that the information associated with that message might change. Therefore, we have that:

- LP_i has sent a straggler message $e_{straggler}$ to LP_j;
- LP_j rolls back to the ST LVT_j such that $T_{straggler} > LVT_j \geq T_{e_{max}}$ where $T_{e_{max}} = \max\{T_e | e \text{ is already processed}\}$;
- This involves undoing an event processed by LP_j which caused the generation of a new event scheduled for LP_k.

Due to the rollback operation which was involving LP_j, LP_k has reached as well an inconsistent simulation state. Therefore, we must support a mechanism which notifies LP_k that an event which it has received belongs to an inconsistent portion of the simulation trajectory. This is done by relying on *antimessages*. An antimessage $\bar{e}$ is a negative copy of an already sent (positive) message e, scheduled by some LP to anyone else.

During the rollback operation, LP_j checks whether some positive messages where sent during the execution of the events the timestamps of which fall in the interval $[LVT_{rollback}, LVT_j]$, where LVT_j is the ST stored into the clock of LP_j before the rollback operation takes place and $LVT_{rollback}$ is the value that will be stored in LP_j's clock after the rollback operation is completed. A negative copy of all the messages falling into this category is sent to the destination LPs.

When LP_k receives an antimessage, two situations might arise:

1) the antimessage $\bar{e}$ is associated with a timestamp $T_{\bar{e}} > LVT_k$. This means that the positive message has not been processed yet. Then, the effect of the antimessage $\bar{e}$ is to simply annihilate the positive message e, i.e. e is removed from the event queue of LP_k;

2) the antimessage $\bar{e}$ is associated with a timestamp $T_{\bar{e}} \leq LVT_k$. In this case, LP_k has already processed the event e which belongs to an inconsistent portion of the simulation trajectory, and it has reached an inconsistent simulation state as well.

In the second case, LP_k must rollback to a consistent simulation state as well (as in the example provided in Figure 2.6). This phenomenon is know as *cascading rollback*, and can in turn affect more than one LP, if the same situation (i.e., during the rollback some events which generated new events destined to different LP are undone) arises.

It is important to mention that the rollback operation can be handled completely by the simulation kernel, if the implementation is able to know where the simulation states S_i of the various LPs are located in memory. If the kernel is able to do so, then no modification to the model's implementation must be done. On the other hand, the programmer must explicitly tell where the memory buffers storing the state are. This will be discussed more thoroughly in Chapter 4, where we will approach transparency towards scattered-memory states and incremental state saving.

In order to support rollback operations, two main approaches have been proposed in literature. One, relies on the notion of *state save & restore*, and was the original proposal in [76]. The other relies on *reverse computation* of simulation events, which assumes that events can be coupled with *negative events*, which are events that execute the same operations of the corresponding events, but in reverse order. It is important to make a clear distinction between *negative events* and *antimessages*, as they refer to a completely different logical operation. The former *undoes* the effect of processing one event on the simulation state, while the latter completely *annihilates* one event, which might or not be already processed. The two strategies will be discussed later in this section.

Global Virtual Time (GVT) If the rollback operation is easy to be proven correct [92], it is harder to reason about the progress condition of the whole system among all the rollback operations being performed. Additionally, since events being processed might belong to a portion of the simulation trajectory that will be later discovered as inconsistent, how is it possible to check whether the ending condition has been reached? And how is it possible to handle error conditions or I/O operations? Although the problem of I/O operations will be thoroughly discussed in Chapter 5, the original proposal in [76] provided a global control mechanism which is based on the concept of *Global Virtual Time* (GVT). GVT is a property of an instantaneous global snapshot of the system at WCT t, and is defined as follows:

Definition 2.1 (Global Virtual Time)**.** GVT(t) is defined as the minimum timestamp of any unprocessed message or anti-message flowing in the system at WCT t

Definition 2.1 tells us that the value of the GVT at WCT t can be computed by inspecting the timestamps associated with all messages in the system that are not yet processed, but it does not say where the messages are likely to be found. In particular, while it is easy to inspect the timestamps stored into the event queues of the various simulation kernels, if a message has been sent from LP_i to LP_j, but it has not been received yet by LP_j (due, e.g., to network latency in the case of remote simulation kernels), then the value of that message (which is referred to as *in-transit message*) must be taken into account as well.

There are several other definitions of GVT, and differentiated implementations [100, 99, 177, 8, 50, 159, 104, 12, 94] which address different simulation scenarios and different computing architectures. Nevertheless, computing the GVT value is an essential component of (distributed) PDES, as it allows to define the *commitment horizon* of the simulation. In fact, we recall that the original DES model's definition states that during the execution of an event e a new event e' can be generated only if associated with a timestamp $T_{e'} \geq T_e$. Then, at any instant t of WCT, since the value of $GVT(t)$ is the minimum timestamp among all the not-yet-processed events in the whole system, it is clear that no event executed by whichever LP in the system can generate a new event e' associated with a timestamp $T_{e'} < GVT(t)$.

Thus, this property states that at WCT t, no rollback operation can bring any LP i to a ST $LVT_i < GVT(t)$. All the executed events associated with a timestamp $T < GVT(t)$ are therefore *committed*, and can be used to verify, e.g., the simulation's ending condition.

Rollback Supports: State Save & Restore As mentioned before, to support the rollback operation, one of the two main approaches is *state save & restore*. The first proposal of this technique appears in conjunction with Time

Warp, in [76]. The technique is based on the idea that a *snapshot* of the simulation state of a LP can be taken (i.e., the memory region(s) containing the private state variables of S_i are copied into a separate buffer) and is associated with a timestamp (namely, the timestamp of the last executed event[6]). Again, depending on the programming model supported by the simulation kernel, this operation can either be done transparently, or shall require the user to explicitly specify which are the memory regions containing the LP's simulation state S_i.

Whenever a rollback operation occurs, the ST $T_{rollback}$ is defined, and the state restore operation can be simply executed by retrieving the state snapshot s associated with timestamp $T_s = T_{rollback}$ and restoring back in place the values of the state variables. In this way, if a straggler message is received and a rollback operation shall be executed, the system can select the simulation snapshot associated with the highest timestamp among the ones which are lower than the straggler's[7].

It is interesting to note that taking state snapshot is a costly operation, both in terms of execution time and memory usage. To cope with memory usage, we can rely on the notion of GVT as presented in Definition 2.1. In fact, since state snapshots are only required to support rollback operations, and given that at any WCT instant t it is not possible to execute a rollback operation to ST $T < GVT(t)$, all the state snapshots s associated with a timestamp $T_s < GVT(t)$ can be discarded, and the memory buffers can be retrieved for future usage. This operation, known as *fossil collection* [77], can be performed periodically. The frequency according to which the value $GVT(t)$ is computed (and therefore the

[6]In fact, given the property that between two consecutive simulation events the value of the clock does not increase, the timestamp of the last executed event can be safely associated with the ST span which is described by this single state snapshot.

[7]Again, in case contemporaneous events are processed, a tie-breaking function can help in the selection of the correct state to be restored among the ones associated with a same timestamp.

fossil collection operation is performed) can be either manually specified at simulation start-up, or might depend on the underlying hardware architecture [8]. Of course, executing the fossil collection often might affect the overall simulation throughput, as more computing power is used to compute the GVT reduction and to release old (committed) buffers.

The issue of execution time for taking snapshots is, on the other hand, tackled by modifying the frequency of the state saving operation (which indirectly tackles memory usage as well), and by tuning the amount of information that is stored into one of them (namely, a snapshot can be either *full* or *incremental*).

Rollback Supports: Reverse Computation A completely different approach to support the rollback operation is *reverse computation* [23], which essentially relies on the notion of *reverse events* to restore a previous simulation state. By relying on compiler techniques, the work proposes to automatically generate reverse events which are able, by executing the same operations of regular events but in reverse order, to undo the effects of one event on the simulation state. A rollback operation is therefore supported by executing at the rolling back LP i all the reverse events in backwards order, starting from ST LVT_i up to $T_{rollback}$.

A reverse event is essentially a copy of a regular event which executes the same operations but in reverse order. If, for example, we consider this sample regular event shown in [23] which models a cell transition of an ATM multiplexor model:

```
if(qlen > 0) {
    qlen--;
    sent++;
}
```

The reverse event would be:

```
if(qlen "was" > 0) {
    sent−−;
    qlen++;
}
```

We note that while reversing arithmetic operations can be straightforward, the branching condition is not, as the reverse event must check an "old" state variables' value, which is not available when processing it.

Thus, reverse computation relies on the modification of regular events by adding *bit variables*, which are transparently-added state variables which tell whether a particular branch was taken or not during the forward execution phase. The regular event presented in the example above would therefore become:

```
if(qlen > 0) {
    b = 1;
    qlen−−;
    sent++;
}
```

and the reverse event can now rely on a (bit) state variable which was set *after* the branch was taken:

```
if(b == 1) {
    sent−−;
    qlen++;
}
```

This approach increases the size of the simulation state, but relying on bit variables allows to do so in a very negligible way, as the state increase is $\log(\#branches)$. This approach can be used to keep track of the execution of n-way if statements and `switch`/`case` constructs as well.

In reverse computation, nevertheless, we note that not all operations are reversible. For example, *disruptive operations* like `=` or `%=` must be handled relying on state saving techniques. Nevertheless, since these operations have a very fine granularity, and considering that the model's executable is instrumented to alter the original logic to support the execution/generation of reversible events, this operation can be done using a technique similar to the one in [179] described above, where memory updates are tracked and a fine-grain word-based log is generated for each particular memory update.

Loops can be handled by simply taking note of the number of iterations n—which is extremely important in case of variable number of iterations like with the `while` statement. The reverse loop will be then executed n times.

On the other hand, handling jump instructions (i.e., `goto`, `break`, and `continue` instructions) or function calls requires that some bit variables are used to store the actual flow of the forward events. The reverse events can then rely on a set of (automatically-generated) `switch`/`case`s to reproduce the actual (reverse) execution flow. This problem can generate a non-negligible state increase, depending on the complexity of the code and on the actual runtime execution flow of the model.

An important aspect in reverse computation is that of (pseudo) random number generators. This problem, which is similar to the one described in the case of State Saving, requires that repeated calls to random generators belonging to the same (logical) invocation return the same result, thus allowing a piecewise deterministic execution of the model. This can be done by applying the same rules used to generate reverse events to the random number generator code, provided that they do not rely on lossy floating point operations, which are anyway hard to support in the generation of reverse events.

Overall, reverse computation adds a limited overhead to the forward execution of the simulation model, and can provide a significant reduction in the delay of rollback operations, provided that the rollback length is not very high. On the other hand, if the simulation model has a large number of disruptive operations, it can substantially fall back to ISS, but with an overall performance which is lower than ISS'.

Hybrid Synchronization

As shown by the discussion, both conservative and optimistic synchronization schemes can provide the simulation model writer with several advantages or disadvantages, depending on the actual model's logic. Although Time Warp has been proven very efficient in a large range of applications, it can suffer from thrashing phenomena in case of, e.g., the presence of many zero-lookahead events.

To this end, several approaches have been proposed in literature, a couple of the most representative ones are here discussed. The proposal in [136], discusses an approach aiming at combining both synchronization techniques, called *Local Time Warp* (LTW). This approach is specifically targeted at simulation models entailing a large number (i.e., thousands) of LPs, and is realized by introducing the optimistic protocol within the conservative one.

Whenever an unsafe event is detected, rather than stopping the simulation's execution, it is executed optimistically. Two different optimistic executions can be supported:

1) *Limited aggressiveness*: events are executed optimistically only locally, i.e. if a portion of the simulation trajectory is found to be inconsistent, it is sufficient to restore the (local) simulation state, without the need for any

antimessage to be sent;

2) *Unlimited aggressiveness*: execution is more biased to the traditional optimistic synchronization protocol, so that cascading rollbacks might arise, but the optimism is limited so that they only involve a limited number of LPs.

The latter strategy is realized by defining a *temporal window* which identifies an upper bound in the ST of events that can be optimistically executed. In this way, the simulation trajectory which might be detected as inconsistent does not evolves indefinitely, and therefore undoing it will not result in an unbounded execution cost.

Of course, having a too small temporal window might result in a pointless usage of optimistic synchronization, as the performance gain might not justify the more complex synchronization protocol. To this end, probabilistic estimates or machine-learning approaches can be used, in order to fine tune the length of the window, as it has been proposed, e.g., in [177].

On the other hand, the work in [157] proposes a technique called *Elastic Time* which tries to limit the *aggressivity* property of optimistic synchronization. In particular, the proposal notes that whenever a LP has reached a LVT value which is "too far" from the minimum LVT in the system, then the probability of rollback of that LP is likely going to increase.

Therefore, that particular LP is "pulled backwards", as constrained by an elastic, towards the minimum LVT value. This technique, called *throttling*, essentially relies on giving more importance (i.e., a higher scheduling probability) to those LPs which are "closer" to the minimum LVT in the system. This simple approach implements a *Near Perfect State Information* system, i.e. a situation where all the LPs are not "too far" from the minimum, although there is not the

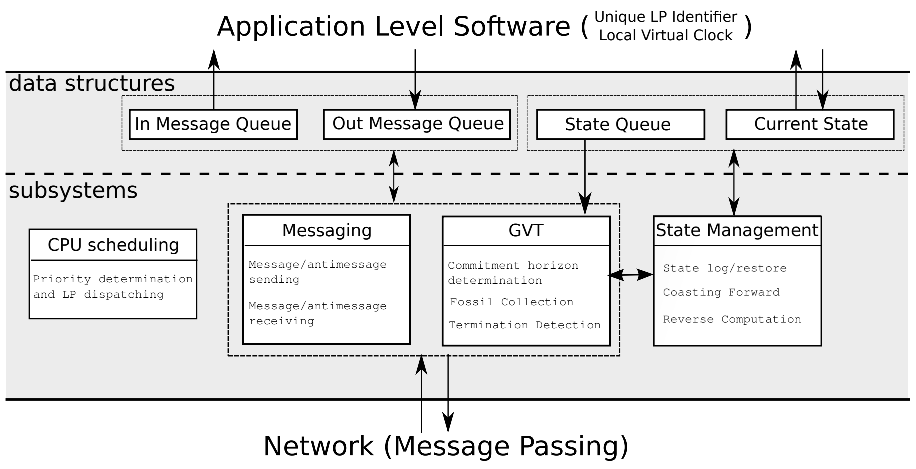

Application Level Software (Unique LP Identifier Local Virtual Clock)
data structures
In Message Queue
Out Message Queue
State Queue
Current State
subsystems
CPU scheduling
Priority determination and LP dispatching
Messaging
Message/antimessage sending
Message/antimessage receiving
GVT
Commitment horizon determination
Fossil Collection
Termination Detection
State Management
State log/restore
Coasting Forward
Reverse Computation
Network (Message Passing)

Figure 2.7: Reference Architecture for Optimistic Simulation Systems

requirement for a significant synchronization effort.

2.3.2 Additional Components of PDES

By the above discussion, it is clear that PDES simulation kernels cannot rely solely on the elements described in Section 2.2.1. In fact, while the conservative approach might require more than one event queue (if implementing, e.g., the solution in [25] that we discussed before), Time-Warp-based simulation kernels must store all pending events, all processed events (in order to support the re-execution of parts of the simulation trajectory), and simulation states.

In Figure 2.7 we show the essential building blocks for a reference architecture supporting the optimistic synchronization protocol. From the picture we can see that at least the following additional data structures and services must be supported by an optimistic simulation kernel.

Input and Output Queues In addition to the (input) event queue, an output queue must be used, in order to keep track of events (and their destination) which have been sent during the execution of other events. This information is mandatory, as during the rollback operation antimessages must be sent to undo the effects of inconsistent operations at remote LPs. To simplify the rollback operation, many implementations [67, 129, 22, 32] rely at least on a per-LP output queue, so that when scanning for antimessages to be sent, the check can be performed only on the send time, without checking (on a per-message basis) the LP identification code as well.

Messaging Subsystem To decouple from the application model the fact that LPs can be stored either locally or on a remote kernel instance, a messaging subsystem takes care of messages "routing". In this way, the simulation model can simply rely on a uniform API for events' scheduling. It is the simulation kernel that determines where (and how) the message must be delivered. Additionally, the messaging subsystem can internally handle the output queue, so that the execution of a rollback operation can be decoupled from the antimessage-sending procedure.

State Queue & State Management Subsystem In case of a simulation kernel implementing the rollback operation by means of state saving & restore, state queue is the fundamental data structured used to recover an LP's consistent execution whenever a causal inconsistency is detected.

The state queue is handled by the *state management subsystem*, the role of which is related to:

1) maintaining a timestamp-ordered list of states, adding new nodes when a new snapshot is required by the system;

2) performing rollback operations (i.e., determining what is the state which

has to be restored from the log, or executing reverse events up to the rollback time);

3) performing coasting forward operations (i.e., fictitious reprocessing of intermediate events in between the restored log and the point of the causality violation);

4) performing fossil-collection operations (i.e., memory recovery, by getting rid of all the events and state logs which belong to an already-committed portion of the simulation).

GVT Subsystem The *Global Virtual Time (GVT) subsystem* accesses the message queues and the messaging subsystem in order to periodically perform the global reduction aimed at computing the new value for the simulation's commitment horizon. In addition, this subsystem cares about termination detection, by either checking whether the new GVT oversteps a given predetermined value, or by verifying some (global) predicate (evaluated over committed state snapshots), which tells whether the conditions for the termination of the model execution are met. Finally, this subsystem is also in charge of performing the so-called fossil-collection procedure, aimed at recovering memory buffers currently keeping obsolete messages and logs, namely those related to the newly-committed portion of the computation.

Event Scheduler A central point relates to the CPU-scheduling approach used to determine which LP, among the ones hosted by a given simulation-kernel instance, must take control for actual event processing activities. As discussed, the common choice is represented by the Lowest-Timestamp-First (LTF) algorithm [96]. It selects the LP whose pending next event is associated with the minimum timestamp, compared to pending next events of the other LPs hosted by the same kernel. LTF has the advantage of avoiding the generation of causality violations across the LPs hosted by the same kernel instance. This is because these LPs are dispatched in a similar way to what would happen on top of a sequential simulation engine, which imposes a timestamp-ordered sequence of CPU-schedule operations for all the events. Hence, rollbacks can be generated only in relation to events scheduled between LPs hosted by different kernels, which contributes to the reduction of the amount of rollbacks.

Different design/implementation variants for LTF exist, along with the basic (stateless) approach in [96] which exhibits $O(n)$ time complexity and relies on traversing pending next events across the input queues of all the LPs. For example, an $O(1)$ statefull approach has been recently proposed [147], which is based on reflecting variations of the priority (i.e., of the next-event timestamp) of the LPs into the CPU-scheduler state—which is done in constant time—and in determining the LP with the highest priority (again in constant time) by running a query on the current CPU-scheduler state.

Random-Number Generators As discussed before, both incarnations of optimistic synchronization (i.e., state-saving-based and reverse-computation-based) require that (pseudo) random number generation is carried piece-wise deterministically. To this end, simulation kernel must provide random number generators which are aware of the rollback operations, and can therefore rollback their internal state as well. This can be done either by storing the internal seed along with a state snapshot, or by implementing a generator which is associated with a reverse function, able to undo the effect of a generation on the internal state.

2.4 Results Achieved within this Thesis

This thesis explicitly tackles PDES systems based on optimistic synchronization implemented relying on state saving & restore, and targeted at multicore systems. The final goal of this thesis is to provide methodologies and system supports which will allow almost any sequential DES model to be run on top of a parallel system without any modification to the original code implementation.

We specifically address simulation models implemented relying on the ANSI-C programming language. This choice allows us to face a scenario where the model writer has a very large freedom, having the possibility to directly manipulate any portion of the memory, without calling any service provided by the simulation kernel, as it would be possible by relying on methods overloading of higher-level languages. At the same time, this poses the highest burden on the simulation kernel, as it has to handle many different programming possibilities to ensure consistency in the final result. Overall, at the end of the thesis, the reader will be provided with a set of results which allow the programmer to implement the simulation model relying on the ANSI-C standard in a perfectly sequential-like style, without any concern about the issues of optimistic synchronization, and having the model being run on a parallel architecture in a fully transparent way.

Particularly, after having presented in **Chapter 3** a survey of some literature results which are of interest for this dissertation, in **Chapter 4** we present an extension to an existing memory management subsystem which allows the programmer to scatter the LPs' simulation state at runtime, by relying on standard calls to the `malloc` memory management API. The subsystem proposes a solution to ISS which is completely transparent: In fact, the user can invoke `malloc` API at any time during the execution of an event, and the simulation kernel is able (by means of software instrumentation) to detect which are the simulation state's regions that are modified during the execution of the event. Additionally, state checkpointing is not immediately executed, rather memory updates are "marked" by relying on a dirty bitmap, so that the checkpointing interval can be tuned at runtime by relying on all the solutions that have been proposed in literature. Transparency is supported by relying on static binary instrumentation of the simulation model, intercepting assembly instructions which (might) involve an update of the scattered simulation state.

In **Chapter 5** we deal with interactions with the outside world in a portion of the simulation trajectory which is not yet committed. We will show how it is possible to allow the execution of operations which are intrinsically non-rollbackable (like, e.g., writing on a file) during the optimistic execution of any simulation event. This will be done by relying on an "output daemon", i.e. a separate process which executes along the main simulation kernel instance. The additional benefit of this approach is that generated (and committed) output can be materialized even on a remote machine (and from multiple, distributed sources). For performance reasons, the operations to support the interactions between the worker threads and the output daemon are implemented as non-blocking algorithms [62]. A discussion of the latency for generating the committed buffers is provided as well.

Chapter 6 addresses the problem of global variables in optimistic simulation models. In particular, the discussion will build a subsystem which allows to drop the constraint expressed by the first part of Equation (2.5), i.e. it will allow to have portions of the global simulation state S which does not belong to the "private" simulation state S_i of any LP. This will be realized again via static binary instrumentation to find which are the possible instructions dealing with global variables update. Then, the actual global variables are realized via multi-version lists, which are again accessed (for performance reasons) via non-blocking algorithms.

The second part of Equation (2.5) is dropped in **Chapter 7**, where we will discuss a protocol and an execution support which jointly allow simulation events to access other LPs' simulation states. This is non-trivial, as different LPs may have reached different ST instants, provided that the simulation model writer does not have to provide any information about the "will" to access different per-LP states. For this part, we rely on an operating-system kernel-level fully innovative memory management architecture, which allows to track inter-dependencies with a much more reduced cost.

To complete the voyage towards transparency, in **Chapter 8** we start studying the implications of the previous advancements. In particular, we analyse what is the impact of optimism on the precision of simulation results, and show how some of our proposals allow to (transparently) experience the same precision of the parallel run as if it were perfectly sequential, yet with the benefit of concurrency.

All the implications of this dissertation on *model programmability* are provided in **Chapter 9**, where we show what the user really has to face for developing a simulation model when our advancements are in place. Finally **Chapter 10** draws the conclusions and throws down the challenges of future work.

All our proposals have been implemented within the ROme OpTimistic Simulator (ROOT-Sim) [67], a multi-threaded optimistic simulation kernel written in ANSI-C based on the Symmetric Multi-Processing (SMP) paradigm and ultimately relying on MPI for remote communication, which has been implemented during the years by our research group. As mentioned, some of the propos-

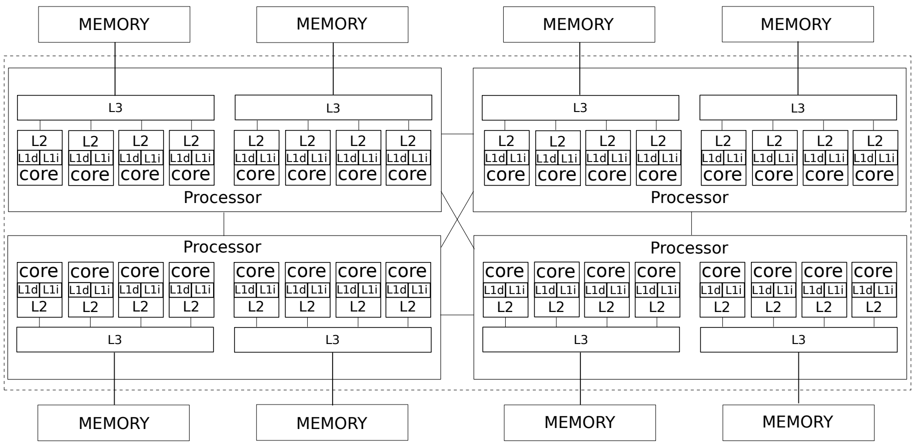


Figure 2.8: Diagram of our Experimental Hardware Architecture

als rely on static binary instrumentation, which has been carried out using the Hijacker Software Instrumentation tool [123], which we have developed as a support tool for our research. For the sake of clarity, in the next Section we will give a glance to all the software which will be used throughout this work, while for completeness, Hijacker is described in **Appendix A**, and ROOT-Sim is presented in **Appendix B**.

2.5 Hardware Setup and Base Software

Our reference computing platform is an HP ProLiant server equipped with four 2GHz AMD Opteron 6128 processors working at 64 bits. Each processor is composed by 8 cores, for a total amount of 32 cores. Each core has a private 128 KB L1 cache (64 KB data-cache and 64 KB instruction-cache) and a private 512KB L2 cache. The last level of cache (LLC), having 5118 KB capability, is shared among four cores within a single processor, for a total of 10236 KB within the same processor. The machine is equipped with 64 GB of RAM based on a

	L1 Data	**L1 Inst**	**L2**	**L3**
Associativity Ways	2	2	16	48
Type	Data	Instruction	Unified	Unified
Size (KB)	64	64	512	5118
Shared vs Private	Private	Private	Private	Shared (4 cores)

Table 2.1: Opteron Cache Details

NUMA (Non-Uniform Memory Access) architecture, where each group of cores that share the LLC sees 8 GB as *close memory* and the remaining 56 GB as *far memory*[8]. A diagram of the machine's architecture is shown in Figure 2.8, while details related to the caching system are provided in Table 2.1.

This hardware architecture has been used only for simulation during all the experiments (i.e., no other process, except for the system ones, where running), thus entailing a scenario where the hardware environment is dedicated to high performance simulation.

The operating system installed on the machine is 64-bit *Debian* 6, with *Linux* Kernel version 2.6.32.5. The compiling and linking tools that have been exploited are *gcc* 4.3.4 and *binutils* (*as* and *ld*) 2.20.0.

Concerning the simulation configuration, we have configured ROOT-Sim to perform GVT and fossil collection every second of WCT.

2.5.1 The ROOT-Sim Platform

We have integrated all our solutions into the ROme OpTimistic Simulator (ROOT-Sim) [127, 124, 67], a multi-threaded ANSI-C/MPI-based open-source

[8]In NUMA architectures several CPU cores share memory resources, and the memory is split in a way that each bank is "close to" a subset of the CPU cores, commonly called node. Each node accesses its close memory banks in fast way, while slower access is experienced for memory banks that are close to others node. The access type is called non-uniform because all the nodes see the whole memory, but each node accesses different memory portions with different latencies.

optimistic simulation platform based on the Time Warp protocol [76] and tailored for UNIX-like systems. For a complete description of the simulation, we refer to Appendix B, while we give here the essential information which allows us to make a general discussion of the experimental results which will be provided into the forthcoming Chapters.

ROOT-Sim is designed as a general-purpose solution, supporting differentiated simulation models adhering to a very simple and intuitive programming model. The platform transparently handles all the mechanisms associated with parallelization, like mapping of LPs on different worker threads on the same machine, and/or different simulation kernel instances (i.e., operating system processes) on remote machines, thus enabling a complete parallel and distributed DES execution on heterogeneous environments.

The programming model supported by ROOT-Sim is based on standard ANSI-C programming and relies on the following API:

`void ProcessEvent(int me, simtime_t now, int evt_type, void *content, int size, void *state)` – a callback that gives control to the application to process an event. This is the main entry point which allows the simulation model to call the proper event handlers. `me` identifies the dispatched LP, `now` is the current LVT, `evt_type` is event numerical code, `content` is the buffer keeping `size` bytes of event payload, and `state` is the pointer (automatically set to the current value of `state_base_address`) allowing the LP to access its state in memory.

`int ScheduleNewEvent(int where, simtime_t timestamp, int evt_type void *content, int size)` – this function injects a new simulation event into the system, destined for any LP identified by `where` (the other parameters have the same meaning as before).

`int OnGVT(int me, void *state)` – a callback that passes control to the application, delivering to LP the last simulation state belonging to the committed computation part.

As it can be seen from the API, the application programmer is not requested to reason on any aspect related to parallelism, thus the model code can be implemented in a sequential-like fashion. She is only requested to understand that what is coded within the `ProcessEvent` callback will be executed speculatively. Hence, a committed portion of the state trajectory can be only seen through the `OnGVT` callback. The execution of non-rollbackable operations within the `ProcessEvent` callback (i.e., during a not-yet-committed portion of the simulation trajectory) will be discussed in Chapter 5.

2.5.2 Hijacker and Ad-Hoc Assembly Modules

Hijacker [123] is a static binary instrumentation tool. It works at compile time, and is able to handle *relocatable objects*, i.e. it does not allow to alter finally-linked executables (this has been a specific design choice, to prevent it to be used on closed software), but it can be seen as an additional compilation stage within the whole compilation tool-chain.

Essentially, Hijacker is both instruction-set independent and executable-format independent, as it performs all its instrumentation activities on an internal (intermediate) representation of the executable. It allows the user to rely on customizable xml-specified rules to carry on the instrumentation process. Additionally, native instructions are grouped into *families*, allowing the user to specify, e.g., a rule which targets all the instruction accessing memory in read mode.

Specific rules can be used to inject into the executable new code snippets, either in any location (in case of, e.g., new functions being inserted into the program) or at specific positions (e.g., right before or after a specific instruction/ instruction family). The provided code, nevertheless, must be specified directly in the host architecture's assembly language. This allows the user to rely on the whole compiling tool-chain to produce it from higher-level source code, or enables the assembly programmer to directly code its new facilities.

To this end, whenever needed, we have decided to directly code our code in assembly, to be passed to Hijacker. This has given us the control to implement specific hardware-related operations in a more efficient (and simple) way, rather than coding bit bashing operations using more complex (higher-level) constructs. In particular, we have specifically targeted our assembly code at x86/x86_64 instruction sets [72, 73]. We have concentrated as well on the ELF executable formats for x86 and x86_64 architectures [165, 105]. This combination of formats currently represents the vast majority of modern computing architectures targeted at high-performance computing, as it is shown in Figure 2.9 and Figure 2.10[9].

2.5.3 Benchmark Applications

Different experimental evaluations have been carried out relying on various benchmark applications, which we have developed specifically for the ROOT-Sim platform. We hereby present a quick description of them, which will allow the reader to notice that they exhibit different interaction patterns and different memory requirements (namely, large versus small simulation states, or constant versus varying overall size) which will allow us to explicitly test the different solutions using the most appropriate one(s).

[9]Figures are drawn with data taken from `http://www.top500.org` as of March 2014.

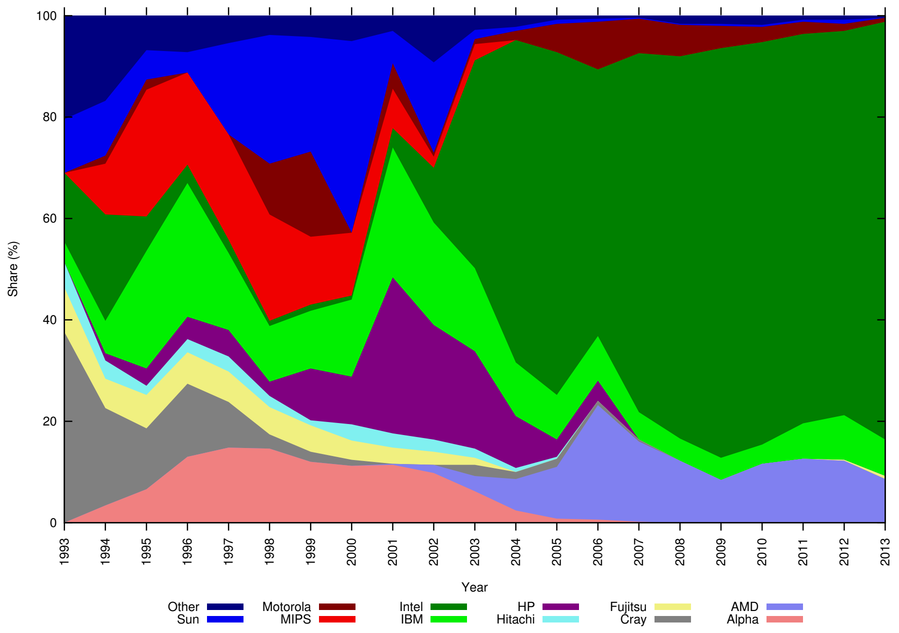


Figure 2.9: Top500: CPU Vendors share over time

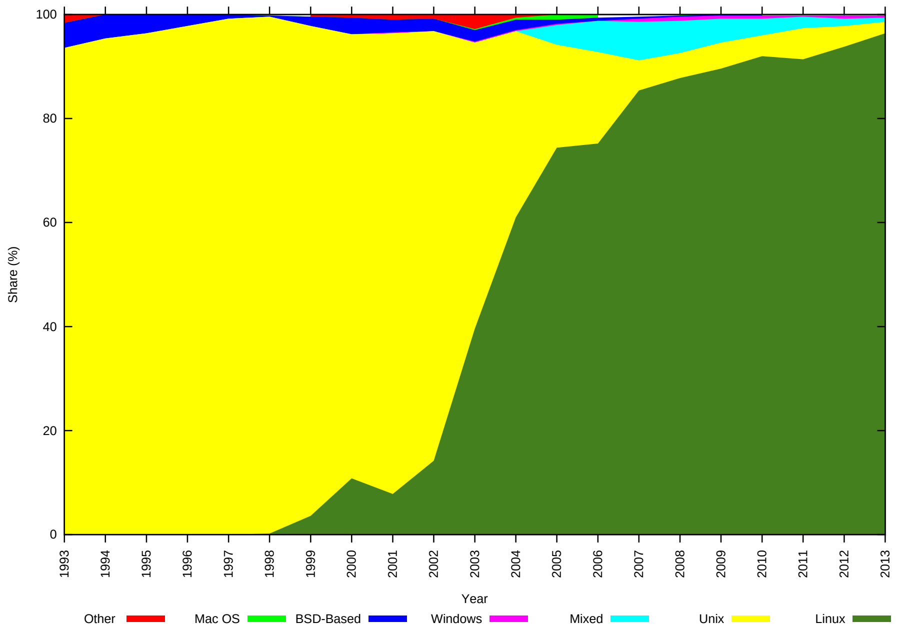


Figure 2.10: Top500: OS share over time

PCS Personal Communication System (PCS) benchmark, which models a mobile network adhering to GSM technology. Each LP models the state's evolution of an individual hexagonal cell, and the whole set of cells provides wireless coverage to a square region of variable size.

Each cell handles a parametrizable number N of wireless channels, which are modelled in a high fidelity fashion via explicit simulation of power regulation and interference/fading phenomena, according to the result in [82].

Upon the start of a call, a call-setup record is instantiated via dynamically-allocated data structures, which is linked to a list of already active records within that same cell. Each record is released when the corresponding call ends or is handed off towards an adjacent cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Upon call setup, power regulation is performed, which involves scanning the aforementioned list of records for computing the minimum transmission power allowing the current call setup to achieve the threshold-level signal-to-interference ratio (SIR) value. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a meteorological model defining climatic conditions (and related variations).

The event types which can occur at any LP are:

- `START_CALL`, which simulates a new call installation on a target cell;
- `END_CALL`, which simulates a call termination;
- `HANDOFF_LEAVE`, which simulates the leave of an on-going call (i.e., of an active device) from the current residence cell. The devices move according to a random mobility model, across adjacent hexagonal cells;
- `HANDOFF_RECEIVE`, which simulates the installation of a call handed off from an adjacent cell;

- `RECOMPUTE_FADING`, which simulates the effects of climatic variations onto the fading and (consequently) interference phenomena for ongoing calls.

This application is highly parametrizable. Beyond the already mentioned number N of wireless channels per cell, the set of configurable parameters entails:

- τ_A, which expresses the inter-arrival time of subsequent calls to any target cell;
- $\tau_{duration}$, which expresses the expected call duration;
- τ_{change}, which expresses the residual residence time of a mobile device into the current cell.

These parameters affect the *utilization factor* of available channels, expressed as:

$$utilization\ factor = \frac{\tau_{duration}}{\tau_A * N} \tag{2.6}$$

This impacts the granularity of the events, since the more the busy channels, the more power-management records are allocated and consequently scanned/ updated during the processing of different events. On the other hand, higher values of the channel utilization factor lead to higher memory requirements for the state image of individual LPs. Both the above dependencies (namely, CPU demand and memory) are bounded by the total number N of per-cell managed channels. In fact, when a call-setup operation is requested due to a call arrival/ handoff arrival, if all the channels are already busy, then the call is dropped, mimicking the real-world scenario where communication is interrupted whenever the base station has no available resources to support the communication.

Terrain-Covering Ant Robots This agent-based simulation model is a variant of the Terrain-Covering Ant Robots (TCAR) model presented in [85]. This type of simulation model is particularly interesting for the what-if analysis of 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 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 this model, the terrain is represented as an undirected graph, therefore an agent (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 agents 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, we have used hexagonal cells, rather than square ones. This allows for a better representation of the agents' mobility in the real world, as the real ant robots (as physically realized in [161]) 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 2.11.

The model relies on a node-counting algorithm, where each cell is assigned a counter which gets incremented whenever any robot visits it, i.e. tracks the number of *pheromones* left by ants, to notify other ones of their transit. When-

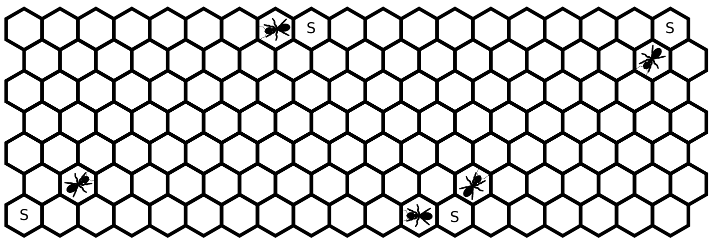


Figure 2.11: TCAR Simulation Model: Terrain and Agents

ever an agent (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, the ant robots adopt a greedy approach, so that when a robot is in a particular cell, it targets the neighbour 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 [161]. In the original model, whenever an agent is in a given cell, it accesses the information stored in the neighbour cells (i.e., trail counters) to make its decision.

As mentioned, the original TCAR model adopts a *pull* approach for gathering trail counters from adjacent cells. Considering the traditional programming model, where LPs communicate by means of (transparently handled) message passing (i.e., LPs' simulation states are disjoint), a large number of events should be exchanged to proceed in the simulation, whenever an agent must change its position. Given that the optimistic synchronization protocol [76] shows higher efficiency when the number of exchanged messages is reduced, we have rather adopted a *push* approach, relying on a notification message which is used to inform all neighbours of the newly updated trail counter whenever an agent enters a cell. Then, each LP stores in its own simulation state the neighbours' trail-counters values. In this way—by relying on only one message—the agents are able to make their decisions locally.

The set of events which are generated/executed by the simulation model are the following:

- `REGION_IN`: an ant robot enters a given cell. When this event is executed, the trail counter is incremented. Then, an `UPDATE_NEIGHBOURS` event is scheduled at all adjacent cells, with an associated timestamp which is equals to the `REGION_IN`'s one. This means that every neighbour is immediately notified of the presence of a new ant robot in this cell at a given simulation time.
- `UPDATE_NEIGHBOURS`: upon receiving this event, the LP taking care of its execution finds in its local simulation state the entry describing the trail counter for the sender of this event. Its value is updated with the one piggybacked by this event. This allows any ant robot in the cell to have (locally) a global view of the state of the neighbours.
- `REGION_OUT`: this event is associated with an ant robot leaving the cell simulated by the LP which will process the event. The logic associated with this event entails finding which is the neighbour to be reached (by consulting the locally stored information on neighbours' trail counters) and therefore scheduling a `REGION_IN` event to the destination cell. We note that, since the time spent by an agent in the cell is modelled by the difference between the timestamps associated with a `REGION_IN` event and its subsequent `REGION_OUT` event, and given that a `REGION_IN` event in any neighbour cell entails the immediate (i.e., at the same timestamp) update

of all trail counters in the neighbours, upon the execution of a `REGION_OUT` event the ant robot can safely consult the locally-stored neighbours' trail counters, being sure that they contain the most up-to-date information, obtained using the aforementioned *push* approach.

At simulation startup, every LP determines what is its position in the square region (in terms of hexagonal coordinates) and checks whether they are boundary regions or not. In the positive case, they store this information in order to prevent ant robots to leave the terrain.

The cells which (at configuration time) are selected as sources for unleashing the ant robots (e.g., the cells associated with the position of rescue teams on the terrain) detect this, and schedule at themselves a `REGION_IN` event, at simulation time 0. This allows the actual simulation to start.

NoSQL Data-Store Simulator This model, which is implemented on top of the framework described in [34], implements a NoSQL Data Store based on distributed/replicated serves, each keeping a subset of the whole set of keys in the entire data-set. Particularly, we consider a model where atomicity of the distributed transactions is ensured by running the 2-phase-commit (2PC) protocol across all the nodes keeping keys that belong to the write set of the committing transaction.

Each cache-server keeps track of a set of M data-object, each one modelled as a $\langle key, value\rangle$ pair. Also, this information is kept and managed as a hash with bucked data structure (indexed by key values), where buckets are implemented as (list of) arrays in order to speedup the process of searching $\langle key, value\rangle$ pairs when processing specific simulation events.

Each cache server is modelled via an individual LP, which is in charge of simulating resource usage (e.g., CPU usage) at that cache server as well as the

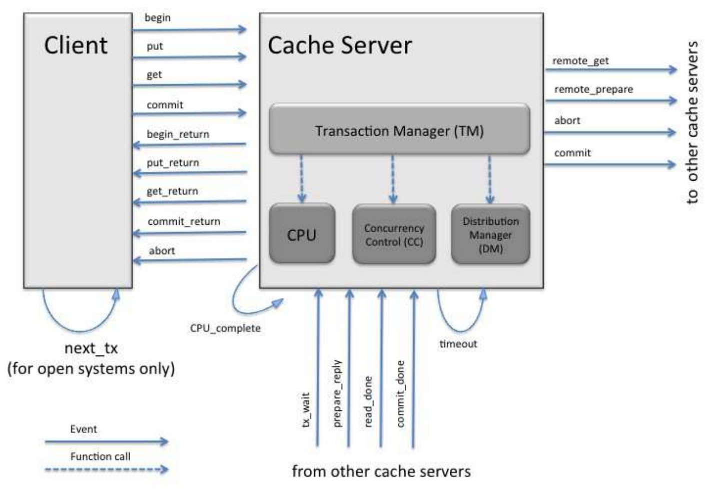


Figure 2.12: Client and Cache-Server LP

evolution of the state of the data-objects kept by the server. A cache-sever LP can be sketched as in Figure 2.12. By the scheme we can identify four main software components:

- the transaction manager (TM);
- the distribution manager (DM);
- the concurrency control (CC); and
- the CPU.

The main types of events occurring at any cache server are:

- `TX_BEGIN`, which simulated the setup of the transactional context;
- `GET`, which simulates a read operation within a transaction;
- `PUT`, which simulates a write operation within a transaction;
- `PREPARE`, which simulates the start of the 2PC distributed coordination protocol;
- `COMMIT`, which simulates the finalization of the transaction.

We note that Get events can be scheduled across different LPs, which is the case when attempting to read a $\langle key, value \rangle$ pair which is not hosted by the local server that is in charge of running the transaction. Also, as expected, interactions across different LPs also take place via the exchange of Prepare and Commit events. Even though in this model the management of each transaction requires dynamic installation of a proper transaction management record (which is supported via dynamic allocation of a proper data structure), the most part of the state of an LP is used to represent data objects. Hence, the larger the data-set kept by each cache server, the larger the actual LP state.

For a more in-depth description of the framework, we refer the reader to [34].

Chapter 3

Literature Survey

La literatura es una batalla silenciosa en la que uno ha de ganar, o de perder;
palmo a palmo, un territorio quo no es suyo con armas que no le pertenecen.
(Literature is a silent battle in which everyone has to win or lose,
a territory that is not yours, not with weapons that belong to him.)
— Juan José Millás, Literatura y Necesidad, Revista de Occidente, 1989

As Chapter 1 has clearly highlighted, PDES is a very multifaceted topic. An all-embracing discussion of the achieved results in literature from around 35 years of research would be too much off topic for this dissertation. Therefore, in this section we will provide an overview of all the results which relate together the programming model supported by PDES systems, the degree of transparency offered to the programmer, the level of parallelism and the performance of simulation. Some of these results will be used to support our proposals, while some others will be discussed to emphasize on their pros and cons, to show what are the different directions that we have taken. In this discussion, we will focus on the optimistic flavour of PDES supported by state saving & restore, as it is what we directly address. We will propose just a small digression on other topics which form up the ground base of this dissertation.

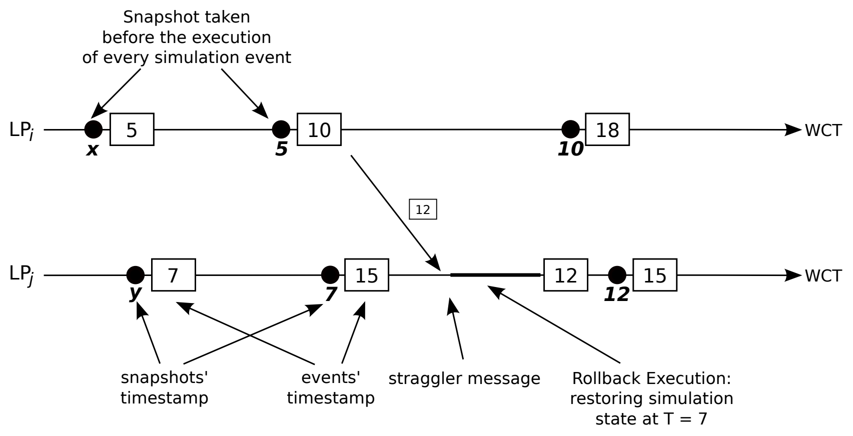

Snapshot taken before the execution of every simulation event
LP_i
x
5
5
10
10
18
WCT
12
LP_j
y
7
7
15
12
12
15
WCT
snapshots' timestamp
events' timestamp
straggler message
Rollback Execution: restoring simulation state at T = 7

Figure 3.1: Copy State Saving Approach

3.1 State Saving

As mentioned in Chapter 1, optimistic synchronization can be supported via state saving & restore. The way simulation states' snapshots are taken (in terms of frequency, and mode) can significantly affect performance and memory usage.

3.1.1 Copy State Saving (CSS)

It is the simplest technique to support State Saving & Restore, and appeared for the first time in [76]. When using CSS, a simulation state snapshot is taken immediately before a new event is executed, as shown in Figure 3.1.

In this way, upon the execution of a rollback operation restoring the simulation execution at ST $T_{rollback}$, it can always be found a simulation state s associated with timestamp T_s such that $T_s < T_{rollback}$, and there is no (processed) event e associated with timestamp T_e such that $T_s < T_e < T_{rollback}$.

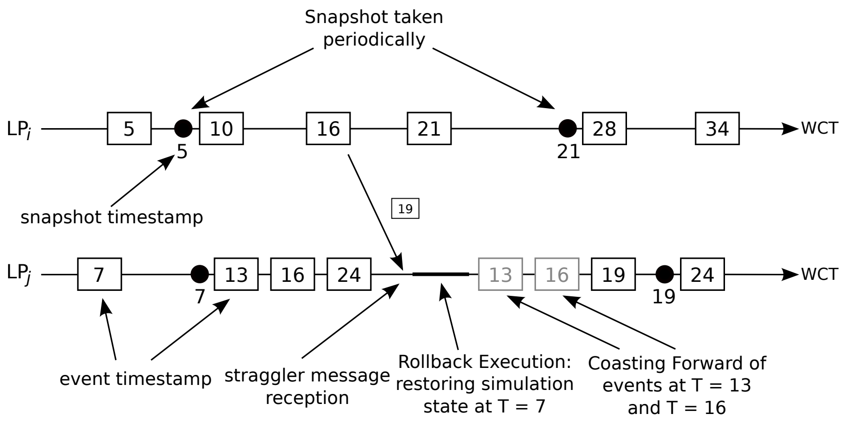

Snapshot taken periodically
LP$_i$
5
5
10
16
21
21
28
34
WCT
snapshot timestamp
19
LP$_j$
7
7
13
16
24
13
16
19
19
24
WCT
event timestamp
straggler message reception
Rollback Execution: restoring simulation state at T = 7
Coasting Forward of events at T = 13 and T = 16

Figure 3.2: Sparse State Saving Approach

Of course, taking a snapshot after the execution of each event is extremely memory greedy. Therefore, to compensate for this, the fossil collection operation (and thus the GVT calculation) should be performed more often, resulting in the aforementioned performance decrease.

3.1.2 Sparse State Saving (SSS)

To overcome the high resource demand from CSS, various approaches which all fall under the name of Sparse State Saving (SSS) have been proposed. The basic idea is to take snapshots *sparsely* [95, 13], rather than before each event, as shown in Figure 3.2. The event-period according to which a snapshot should be taken can be either fixed—in the case of Periodic State Saving (PSS)—or variable—in the case of Adaptive State Saving (ASS).

By relying on SSS, whenever a rollback operation should be performed, we have two possibilities:

1) There exists a state snapshot associated with timestamp $T_s = LVT_{rollback}$.

Then the rollback operation is carried on exactly as in CSS;

2) There is no state snapshot associated with timestamp $T_s = LVT_{rollback}$.

In the latter case, the state snapshot associated with the higher timestamp among the ones lower than $LVT_{rollback}$ is restored. Then, as shown in Figure 3.2, some events must be reprocessed in order to re-align the clock of the rolling-back LP to the value $LVT_{rollback}$, an operation which is known as *coasting forward*. It is important to note that, during the re-execution of these events, no messages should be delivered by the rolling-back LP. In fact, since the events have already been processed, the involved messages have been already sent. And since the timestamp associated with any reprocessed event e_{repr} is such that $T_{e_{repr}} < LVT_{rollback}$, the antimessages $\bar{e}_{repr}$ have not been sent. Then, if during the coasting forward operation, messages are sent to other LPs, they are going to receive multiple copies of the same messages, thus creating an error in the simulation's results. This re-execution without message sending is called *silent execution*. Although in this scenario it would be practically correct to send the antimessages for any event e_{repr}, from a logical point of view it is not, as they do not belong to a portion of the simulation trajectory which is discovered to be inconsistent. Furthermore, doing so could generate biases in the execution of the simulation.

Additionally, it is important that the re-execution of any event follows the same (original) execution trajectory. If, for example, the logic associated with an event relies on some probability distribution function (based on pseudo-random number generation), it is important that re-executing the same (logical) call to the random generator provides the event with the same exact result. In the negative case, the logic associated with the event might produce a different result with respect to the previous execution. This behaviour, known as *piece-wise* *determinism* (PWD) [41], is necessary to correctly reconstruct the very same simulation state before executing the straggler event, and can be supported by the underlying simulation kernel by exposing its own version of a random library, which is aware of the rollback operation.

PSS has the undeniable advantage that memory consumption due to state saving is reduced. Yet, the rollback overhead is increased, due to the cost of the coasting forward operation. The efficiency of the approach depends on the checkpointing period χ. If χ is too small, memory is inefficiently used. On the other hand, if χ is too large, we should expect a performance decrease.

Various ASS techniques have been proposed, which try to fine tune the value of χ depending on the actual execution dynamics of the simulation model. The approach described in [121] selects the best checkpointing interval by relying on an analytic model based on LP execution time. By assuming that the execution of events is *non-preemptive*[1], and by assuming that the *rollback length*[2] is independent of each other, the optimal checkpointing interval is:

$$\chi_{opt} = \left\lceil \sqrt{\frac{2\delta_s}{\delta_c} + \left(\frac{N}{k_r} + \gamma - 1\right)} \right\rceil \tag{3.1}$$

where:

δ_s is the average time to take a state snapshot;

δ_c is the average time to execute the coasting forward operation;

N is the total number of committed events;

k_r is the number of rollbacks executed;

[1]This has been the traditional behaviour of events' execution. See Chapter 7 for a more detailed discussion of this topic.

[2]Rollback length describes how many (optimistically-executed) events are undone by a rollback operation. The average rollback length can be used as a measure of the amount of "wasted work" in an optimistic simulation run.

Algorithm 3.2 Optimal χ Selection

```
if $n = 0$ then
    $\chi_n \leftarrow \chi_{init}$
else if $k_{obs} = 0$ then
    $\chi_n \leftarrow \lceil (1-\rho)\chi_{n-1} + \rho\chi_{max} \rceil$
else
    $\chi_n \leftarrow \max(1, \lceil (1-\rho)\chi_{n-1} + \rho \min(\chi_{min}, \chi_{max}) \rceil)$
end if
```

γ is the average rollback length.

Under the same assumptions, the work in [140] proposes to observe in a WCT interval T_{obs} the number of rollback operations k_{obs} and the number of executed events R_{obs} (both committed and uncommitted). A numerical sequence of checkpointing intervals χ_n is generated, where the first element is given by:

$$\chi_{init} = \left\lceil \sqrt{2 \frac{R_{obs}}{k_{obs}} \frac{\delta_s}{\delta_c}} \right\rceil \tag{3.2}$$

The next values are then computed according to Algorithm 3.2, where:

$\rho \in (0, 1)$ determines whether we are giving more importance to the history of χ_n rather than to more recent observations;

χ_{min} is the minimum threshold;

χ_{max} is the maximum threshold.

This scheme does not take into account the fact that the execution time of different typologies of events can vary. This aspect is captured in [152], where the *Event Sensitive State Saving* (ESSS) is proposed. This technique emphasizes that it is convenient to take a state snapshot when the *granularity* of the next event[3] increases. Then, starting from the model in [140], and classifying the

[3]By *granularity of an event*, we mean the average execution time of one type of event, with respect to the average event's execution time that considers all kinds of event to take the same WCT to be executed.

events in N different classes, the simulation kernel groups in the same class $n \in N$ all the events showing a similar behaviour (in terms of required WCT for execution). A proper χ_{opt} is then selected depending on the most-occurring class. Assuming that each class n of events is associated with an *event frequency* f_n and a *state saving probability* p_n, the optimal checkpointing value is computed as the geometric average of all the classes:

$$\chi_{opt} = \left(\sum_{n=1}^{N} p_n f_n \right)^{-1} \tag{3.3}$$

This approach, therefore, tries to reduce the coasting-forward time by avoiding to reprocess chains of events containing ones that require a high amount of WCT to be reprocessed.

A different approach is presented in [45], which regulates the checkpointing interval by using a *heuristic algorithm*, based on the periodic re-calculation of the cost function:

$$E_c = C_{ss} + C_{cf} \tag{3.4}$$

where:

C_{ss} is the average WCT to perform a state saving;

C_{cf} is the average WCT to execute the coasting forward operation.

Taking into account observation periods which are completely independent of the checkpointing intervals, the system periodically re-computes the value of Equation (3.4). If its value increases, then the value of χ is increased by one unit (up to a maximum threshold). If there is no fluctuation in the value of Equation (3.4), then χ is decreased by one unit. Therefore, if the model's dynamics change—and this change is reflected in a variation of the rollback probability— the time required to perform state saving and coasting forward operations changes, and the value of χ is adapted accordingly.

An additional approach in [133] proposes to observe LPs' *event history*, taking into account the variations between the timestamp of two consecutive events, to determine which is the best moment for taking a snapshot. In particular, a rollback operation can involve any ST instant, specifically it can fall into any interval bounded by the timestamps of two consecutive events. If LP i's clock has value LVT_i, and if the next event's timestamp is T_{next}, if the ST interval $I = [LVT_i, T_{next}]$ is such that the difference $T_{next} - LVT_i$ has a positive (non-negligible) variance with respect to the average value, there is a higher risk that a rollback might affect that ST interval I, and it is therefore convenient to take a log.

In [134] a cost model is proposed, to select the checkpointing position in an optimized way. It is based on a heuristic which tries to minimize the rollback length: the system decides to pay the cost of a checkpoint at a certain ST instant only if the estimation of its possible (future) restore cost is higher.

The model takes into account the following parameters:

- the position of the last checkpoint;
- the granularity of the events executed in between the last taken checkpoint and the current LVT;
- the probability that state S will be restored in the future, due to a rollback operation.

which are combined in the equation:

$$CR(S) = \begin{cases} \mu_s + P(S)\mu_s & \textbf{if } S \text{ is saved} \\ P(S)\left[\mu_s + \sum\limits_{e \in E(S)} \mu_e\right] & \textbf{otherwise} \end{cases} \tag{3.5}$$

where:

μ_s is the save & restore cost of state S;

μ_e is the granularity of event e, which is part of the cost of the coasting forward operation;

$P(S)$ is the estimation of the probability that the current state S will be restored in the future, which depends on the application dynamics and on the interval $I(S)$ which spans from the last checkpoint's timestamp and the current LVT;

$E(S)$ is the set of events executed in the interval $I(S)$ which will be re-executed during a coasting forward operation.

Equation 3.5 is computed in both forms (as if it were necessary to take a snapshot, and as if it weren't) before executing any event e. The result associated with the most convenient option determines whether the checkpoint will be taken or not.

In [166] we find a work which addresses an orthogonal problem to checkpointing interval, but which is perfectly compatible with the aforementioned SSS approaches. This work proposes a transparent memory-management architecture targeted at optimistic synchronization which allows the user to rely on dynamically allocated memory to store the LPs' simulation states. This work is actually one of the basis upon which our solution in Chapter 4 is built, so we remind the reader to that Chapter for a more specific discussion.

Other proposals oriented to transparency for checkpoint/restore operations in the context of general memory layouts cope with optimistic synchronization in the High-Level-Architecture (HLA) [144, 145]. They rely on kernel-level memory protection mechanisms offered by the Operating System, used to detect memory accesses and to trigger incremental copies of the accessed pages.

Another proposal which targets checkpoints of scattered-memory simulation states is the one in [156], which nevertheless offers a degree of transparency reduced with respect to [166, 144, 145], as the user has to explicitly notify the simulation kernel about which memory buffers are being used for storing LP's state variables.

3.1.3 Incremental State Saving (ISS)

If the above mentioned techniques address the problem of state log/restore by tuning the checkpointing interval to minimize the WCT required for a rollback operation, they do not take into account that, even if seldom-executed, a checkpointing operation can be onerous due to the size of the the state S_i being saved. Additionally, if S_i is large, but only a small portion of it was modified since the last checkpoint, there is a large amount of wasted time spent copying redundant information.

The goal of Incremental State Saving (ISS) is to limit checkpointing overhead by reducing its execution time, and by limiting at the same time the amount of memory used for a single log.

The first ISS approach has been published in [10], and is based on the fact that each event $e \in E$ is augmented with the following information:

- The value of modified state variables *after* the event is processed;
- The value of modified state variables *before* the event is processed;
- The ST T_{gen} when the event is generated;
- The ST T_{exe} when the event must be executed, with $T_{exe} > T_{gen}$;

The simulation kernel must handle a more complex structure than the event queue, where (for each LP) a list of modifications of the states is stored. The nodes of this queue are linked to the events that caused the updates. Upon the receipt of a straggler message associated with timestamp $T_{straggler}$, the rollback operation is carried on in a way different from the previously presented one. In particular, all the events of the rolling back LP i such that $T_{straggler} \leq T_{exe} \leq LVT_i$ are scanned, and the corresponding state variables that where modified are put back in place, taking care that the same variable is updated only once[4].

Of course, this technique is not transparent at all, as the model programmer must be aware of the concept of rollback and state saving, and has to access the kernel's data structures to make copies of the state variables before modifying them. This is necessary, because the simulation kernel is not aware of where the simulation state is stored, and of which parts are being updated by an event.

The work in [141] makes an additional step towards transparency, by relieving the application-model writer from directly modifying simulation kernel's data structure. In particular, it proposes to implement incremental checkpointing in a transparent fashion by relying on Object Oriented *overloading* mechanism. This choice narrows its applicability to OO programming languages (like, e.g., C++ or Java). The basic idea is based on two essential points:

- all (user-defined) functions which process events (i.e., the event handlers) must be redefined by means of *overloading*;
- state variables must be *encapsulated* within classes defined by the simulation kernel.

Encapsulation allows the simulation kernel to discriminate state variables

[4]Although this might seem an incorrect algorithm for state restore, we emphasize that the original proposal in [10] was targeted at a specific scenario, namely *digital logic simulation*, where additional property guarantee that the final simulation state is restored correctly. We refer to the original paper [10] for a complete discussion of the algorithm, and to Chapter 4 for a more general approach to incremental restore.

from other ones which might be used to store only temporary results used for computation. The application must therefore notify the kernel of which are the state variables by using the special signature `State<class T>`, which wraps the class `T` that the system will handle as part of the state. The programmer will be allowed to manipulate wrapped objects by using overloaded methods, which will make a copy of the accessed data buffers before the actual update is performed.

In this scheme, the user must be aware of the notion of rollback and of the fact that the simulation kernel will perform incremental checkpoints, yet differently from [10] the operation is performed by relying on a service exposed by the simulation kernel which gives a certain degree of freedom in the definition of the simulation state.

The proposal in [179], targeted at x86 computing architectures, supports transparent incremental state saving by relying on software instrumentation. The simulation model's assembly code is parsed and, whenever an instruction that updates a memory region is found, a `call` to an ad-hoc module is prepended, which generates a copy of the old memory's value before updating it. Whenever a rollback operation is performed, the chain of memory updates is scanned backwards, in order to realign the content of the simulation state to time $T_{rollback}$. This technique is oriented to the programmer, so that she does not have to alter the original simulation code because the instrumentation phase automatically detects which are the possible assembly instructions that will alter memory content. Nevertheless, the approach used in [179] suffers from a performance sub-optimization. Let us consider the following code snippet:

```
for(i = 0; i < MAX; i++) {
        state->array[i]++;
}
```

This code could be an acceptable model code, which might be used, e.g., to increment some statistics in the model's state. Yet, each iteration of the loop would entail two memory updates, one to variable `i` and one to state variable `state->array[i]`. The proposed solution would call the ad-hoc module twice per iteration, and would create a node in the state chain for each modification of the `state->array[i]` variable, given that its granularity is word-based. Therefore, the single event's execution delay is increased depending on the memory-access pattern of the simulation model which, even in simple examples like the proposed one, could be non-negligible.

The work in [125], which will be thoroughly discussed in Chapter 4, tackles this issue in the case of dynamically-scattered simulation states by relying on a *dirty bitmap*. Essentially, it relies on assembly-code instrumentation as well, but given that an ad-hoc memory manager is (transparently) interposed between the simulation model and the underlying system memory manager, each memory update is materialized by simply setting one bit (corresponding to the touched memory area) to 1, stating that the memory area was updated. In this way, the checkpointing operation is carried out periodically, saving only the areas that were actually updated since the last checkpointing operation. This allows the simulation kernel to rely on any of the aforementioned optimizations regarding the checkpointing interval χ, as it has been shown in [168]. The latter approach relies on an integral function for fine-tuning the checkpointing parameters, enforcing as well stability of the decision towards fluctuations in the execution dynamics.

In Table 3.1 we show a comparative summary of all the state saving techniques presented so far, which highlights which properties are provided by each of the above-discussed approaches.

SS Technique	PSS	ASS	Scattered Simulation State	Incrementality	Arbitrary Granularity	Transparency	Instrumentation-Based	Analytic Model	Temporal Measurements	Heuristics	Autonomicity	Stability to Fluctuations
[76]												
[77]												
[13]	✓	✓										
[121]	✓	✓	✓						✓	✓		
[140]	✓	✓	✓						✓			
[152]	✓	✓	✓						✓			
[45]	✓	✓	✓							✓	✓	
[133]	✓	✓	✓						✓			
[10]	✓				✓	✓						
[145]	✓				✓		✓					
[141]	✓				✓							
[179]	✓	✓	✓		✓		✓	✓		✓		
[125]	✓	✓	✓	✓	✓	✓	✓	✓				
[134]	✓	✓	✓							✓		
[168]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 3.1: Comparison of State Saving Techniques

Advancements by the thesis

As it will be discussed in Chapter 4 we will explicitly allow the programmer to rely on dynamically-allocated memory to scatter the simulation state, and to make it grow/shrink depending on the actual model's execution dynamics. This will be done transparently at linking time by relying on specific malloc wrappers, which will redirect calls to the standard memory-allocation library to our specific memory manager. The memory manager will work at *chunk* granularity, i.e. upon simulation startup a large buffer (for each LP) will be pre-allocated, for serving subsequent memory request by each LP.

Furthermore, we will transparently integrate (via static software instrumentation) a memory-update tracking subsystem, which will allow the simulation kernel to know precisely which portions of the simulation states have been updated since the last log operations. This information will be kept as small as possible, to cope with memory usage, by relying on a *dirty bitmap*, i.e. a compressed data structure where each bit tells whether a specific chunk has been modified.

By relying on this information, the simulation kernel will be able to transparently execute Incremental State Saving, thus reducing the overhead required for taking a simulation snapshot and additionally minimizing the memory footprint related to state saving.

3.2 Consistent Output Generation

The issue of output commitment in rollback-recovery systems has been thoroughly studied in literature, although often in other topics. An excellent survey of the proposed approaches can be found in [41]. These solutions have been oriented to fault tolerance, where rollback occurrence is only due to failures, potentially causing the loss of volatile state information causally related to the output messages. Overall, these solutions support reproducibility of state trajectories (and related outputs) in order to keep the system state aligned with what is actually observed by the external world. Instead, in optimistic PDES systems we experience the problem of masking the outcome of specific state trajectories (those that are rolled back) towards the external world. Further, the approaches surveyed in [41] deal with the Lamport's causality model [89], where no predetermination of events' ordering is imposed by event timestamps, as instead occurs in PDES systems.

Still in the context of fault tolerance, the work in [79] presents a protocol for supporting output commitment with special emphasis on prompt delivery of output data, which is one of our targets as well. It places special attention on state-information copies on stable storage, to overcome the failure of processes and minimizing the time for restarting the execution of the system in case of failure.

The work in [178] addresses the problem of external-world interactions in the context of Transactional Memories (TM). In particular, the work focuses on what a TM system (either software or hardware) should support to allow the execution of irrevocable actions, such as I/O and system calls, when the side effects cannot be rolled back. The proposed solution allows the programmer to mark a specific transaction as *irrevocable*, i.e. the interactions with the external world are immediately finalized, and the correctness is guaranteed by avoiding to abort such transactions. Whenever a transaction is marked as irrevocable, the runtime support checks whether another irrevocable transaction is currently being run. In the positive case, the new transaction's execution is delayed until the other irrevocable transaction commits. In the negative case, the transaction is immediately executed, and in case some conflict on accessed data is detected, the contention manager only aborts other transactions which are not marked as irrevocable. This solution is not transparent, and does not allow multiple (concurrent) interactions with the outside world, which is desirable in the context of high-performance computing.

In the context of operating systems supports, the work in [181] proposes an architecture for enabling applications to define program-specific custom policies to carry on speculative execution, e.g. for I/O operations. Despite the clear lack of transparency—the programmer is required to manually specify policies to determine which speculative branch is correct, which ones should be rolled back, and to manually mark portions of code which can be executed speculatively—this work mostly relies on system calls to communicate between the application and the operating system, requiring a large number of mode/context changes which can significantly affect the overall performance, while we specifically decided to work completely in user space to maximize the simulation throughput. Additionally, speculation is implemented via an ad-hoc version of the `fork()` system call, in order to create a new speculative copy of the process, while in optimistic simulation speculation is already intrinsic in the execution model.

As hinted before, in the context of optimistic PDES engines, one approach to deal with I/O is based on ad-hoc API. One example along this direction is the work in [135], where the optimistic engine is able to provide the user-level software with past, committed (and globally consistent) state images. This is done by passing control (and the snapshot) to the applications via a proper callback. Once taken control, the callback can invoke output operations providing data related to the committed portion of the simulation.

Recently, the issue of transparency in optimistic synchronization in the context of HLA based simulations has been tackled [146], where the problem of direct external interactions (e.g., with the underlying Operating System) is also addressed. However, the provided solution is based on temporary suspension of processing activities at the federates until the interaction is conservatively detected to be safe, and thus committable. A similar approach, but less transparent, has been adopted to PDES in [32], where an ad-hoc API is provided to the application programmer in order to alert the operating system kernel that a non-revocable I/O operation needs to take place. Suspending (or throttling) the execution is not compatible with the optimistic synchronization protocol, as it might vanish all the benefits coming from speculative execution.

In [75], any output message is routed towards this object in the form of an event, which gets processed only after GVT advancement (namely when the event is detected to be committed). While there is a methodological similarity, this approach is based on ad-hoc APIs used to trigger the interaction with the special object, thus providing a degree of transparency which is strictly lower than ours. Additionally, relying on an external daemon gives the benefit of removing most of the work for materializing the output from the simulation engine execution path.

As for non-blocking algorithms, avoiding mutual exclusion has been considered a benefit since the early 1970's [39]. Lamport [88] gave the first non-blocking algorithm for the problem of a single-writer/multiple-reader shared variable. Herlihy [62] proved that for non-blocking implementations of most interesting data types (linked lists among them), a synchronization primitive that is universal, in conjunction with reads and writes, is both necessary and sufficient. A universal primitive is one that can solve the consensus problem

[44] for any number of processes. In our implementation we rely on Compare & Swap (`CAS`), which is a universal primitive

Advancements by the thesis

We will present, as it will be stated in Chapter 5, an innovative subsystem allowing committed interactions with the outside world from (uncommitted) event handlers. This will be done by introducing the notion of an *output daemon*, i.e. a separate process which executes along the main simulation kernel instance, and which communicates with it by relying on shared memory. This solution allows us to perform output materialization operations outside of the critical path of the simulation framework, thus increasing the overall efficiency of the simulation.

As an additional benefit, this solution allows a (geographical scale) ordering of the generated output buffers, which could be used as well in conjunction with conservative synchronization schemes.

We will implement, for performance reasons, all the interactions between the output daemon and the simulation kernel by relying on non-blocking algorithms.

3.3 Data Sharing across Concurrent Logical Processes

The issue of bypassing state disjointness for concurrent objects in PDES systems has been dealt with by several studies. The work in [18] discusses how state sharing might be emulated by using a separate LP hosting the shared data and acting as a centralized server. This proposal also introduces the notion of *version records*, where multi-versioning is used for shared data in order to cope with read/write operations occurring at different logical times, and to avoid unneeded rollbacks of the centralized server in case of optimistic synchronization. This is an approach similar to the one proposed in [107], where a theoretical presentation of algorithms to implement a Distributed Shared Memory mechanism is presented in terms of protocols to keep replicated instances of a variable coherent. In particular, one of the provided algorithms proposes to realize variables as multi-version lists where write operations install new version nodes and read operations find the most suitable version. The above approaches map read/ write access to shared variables to message-passing (namely, event schedule operations). This places a hard burden on the centralized node(s), which in the case of a simulation model performing frequent read/write operations on shared variables can produce a non-sustainable overhead.

Another proposal has been presented in [43], where the notion of *state query* is introduced. A LP needing a portion of the state which belongs to a different object can issue a query message to it, and wait for a reply containing the suitable value. In case this value is later detected to be no longer valid, an anti-message is sent so to invalidate the query. Again, this approach relies on message passing, and is not transparent to the application programmer.

The work in [52] proposes to integrate the support for shared-state in terms of global variables, by basing the architecture on [28].This proposal provides no transparency, as the application-level code must explicitly register a LP as a reader/writer on the shared variables, and the synchronization between LPs accessing shared variables is unlikely to scale well on multi-core architectures, as it is based on locks.

The Software Transactional Memory (STM) paradigm [149] allows multiple threads to access global information while ensuring consistency with respect to concurrent accesses. The user has to explicitly mask a transaction by relying on constructs provided by the underlying middleware, which in turn takes the burden of ensuring consistency of the executed read/write operations. These write operations do not work in-place, i.e., data updating is performed by the middleware (again, by relying on ad-hoc programming API functions) on separate buffers (i.e., write-sets) which are then copied (i.e., externalized) into the global buffer after some safety predicate is computed during the commit phase. Any externalized operation is considered committed, i.e. differently from the PDES paradigm, there is no need to support (at that time) a rollback operation.

The work in [30] proposes a framework targeted at multi-core machines and based on Time Warp, where so called Extended Logical Processes (Ex-LP), defined as a collection of LPs, have public attributes that are referred to variables which can be accessed by LPs in other Ex-LPs. The work proposes to handle shared attributes accesses by relying on a specifically targeted Transactional Memory (TM) implementation, where events are mapped to transactions and the actual implementation of the TM is based on [52].

3.3.1 Advancements by the thesis

In Chapters 6 and 7, we will propose a solution which allows the model writer to share any memory area, without the need for a-priori knowledge of whether some sharing on a specific area can occur. This entails both global variables and different LPs' (private) simulation states.

This increases the level of transparency, again allowing a truly sequential-style programming model to be exposed to the programmer. In fact, she is allowed to make any LPs that takes control touch any valid memory location within the global simulation state without the need for any particular care, just like it occurs in sequential-style programming and related sequential execution scenarios.

Concerning global variables, this will be done by redirecting their accesses (both in read and write mode) to specific management functions via static binary instrumentation. These management function will, in turn, handle multi-versioning lists, the accesses to which will be implemented as non-blocking algorithms.

As for LPs' private states, we will present both an operating-system-level innovative memory management architecture (provided as a loadable kernel module), which will allow different worker threads to share the same view of the data, yet with different access privileges. This will allows us to "materialize" the memory accesses by triggering a segmentation fault, which will be nevertheless handled without relying on actual operating systems signals. Rather, the kernel module will intercept it, and it will pass the control back to the simulation kernel, allowing an innovative synchronization protocol to take place, in order to synchronize the involved LPs in a *rendez-vous* phase which will allow direct memory access to different LPs' simulation states.

3.4 Non-Blocking Algorithms

To increase the performance of our solutions, given that we target multi-threaded simulation kernels deployed on multicore machines, we will often rely on non-blocking algorithms [62]. Avoiding mutual exclusion has been considered a benefit since the early 1970's [39]. Lamport [88] gave the first non-blocking algorithm for the problem of a single-writer/multiple-reader shared variable. Herlihy [62] proved that for non-blocking implementations of most interesting data types (linked lists among them), a synchronization primitive that is universal, in conjunction with reads and writes, is both necessary and sufficient. A universal primitive is one that can solve the consensus problem [44] for any number of processes. In our implementation we rely on Compare&Swap (`CAS`), which is a universal primitive.

A subtle problem associated with most lock-free algorithms is the ABA problem. It was first reported in association with the introduction of the `CAS` instruction on the IBM System 370 [24]. It occurs when a thread T1 reads a value A from a shared object and then an interrupting thread T2 modifies the value of the shared object from A to B and then back to A. When T1 resumes, it erroneously assumes that the object has not been modified. Given such behaviour, there is a serious risk that T2's execution is going to violate the correctness of the object's semantic. Practical solutions to the ABA problem include the use of hazard pointers [108] or the association of a version counter to each element in platforms supporting a double-word compare-and-swap primitive (`CAS2`). We explicitly rely on the latter solution to avoid the ABA problem in our non-blocking implementation.

An important work, which will be very useful to our needs, is the one in [58], where a non blocking implementation of a linked list is presented. In particular, insert operations are executed by locating the correct position where to insert a new node, and then the new node is linked to the list by relying on a `CAS` on the previous node. Instead, to perform a delete operation, a couple of `CAS` operations are executed, one to logically delete the node, and one to physically delete it.

Chapter 4

Efficient and Transparent Incremental State Saving

Those who cannot remember the past are condemned to repeat it
— George Santayana, The Life of Reason, 1906

As mentioned in Chapter 2, a critical aspect concerning PDES when the rollback operation is supported via state saving & restore is how simulation-state snapshots are taken and managed.

This operation has a huge impact on both the performance of the overall simulation, and the transparency that the user can experience. We will cope with both these aspects in this Chapter, showing how it is possible to realize a checkpointing subsystem which allows the simulation kernel to handle checkpointing operations *incrementally*, and without asking the user to provide any information about the in-memory location of the LPs' states. Additionally, this prevents the model writer from being required to have any awareness of the optimistic synchronization mechanism supporting the parallel execution of the simulation as far as recoverability of the local state of the LP is concerned. Furthermore, we will show how it is possible to allow the simulation model writer to scatter the simulation state on dynamically-allocated memory. This will be done by allowing her to rely on standard C `malloc` API.

An important implication of this choice is that the simulation state can vary during the execution of the simulation. Therefore, if in a certain execution phase the model requires more memory, it can be obtained by relying on a `malloc` call, even if the memory request is later discovered to belong to an inconsistent portion of the simulation trajectory.

This subsystem, named *Dirty Dynamic Memory Logger and Restorer* (Di-DyMeLoR) is based on a previous work [166] the goal of which was to support the usage of dynamic memory for storing the simulation state. The goal of the original work, and the design principle which has been followed while implementing Di-DyMeLoR, is that a simulation state snapshot must be taken periodically. This means that, differently from the proposal in [179], we do not take a (word-based) log whenever a memory update operation is performed. Rather, we "materialize" that operation relying on a *dirty bitmap*.

This proposal relies on software instrumentation for identifying memory-update operations. This task has been carried out by relying on the Hijacker static binary instrumentation tool. Although we will give some insights about Hijacker in this Chapter, in order to clarify certain aspects of the proposal, a complete overview of the tool is presented in Appendix A.

Di-DyMeLoR is also related to a number of works in the field of program execution tracing (see, e.g., [46, 6, 132, 188]) for debugging, vulnerability assessment and repeatability. These approaches provide detailed analysis of changes in the state of the program, and of the execution flow. However, this is achieved via performance intrusive techniques relying on dynamic instrumentation and/or kernel level services, unsuited in contexts (e.g., parallel simulation) where performance cannot be sacrificed. Debugging supports showing basic operating mode comparable to our one (namely, the employment of trap mechanisms based on code insertion/replacement to detect memory write accesses) are those addressing data watch points (see, e.g., [174]). However they have performance targets different from ours since optimizations mostly cope with search techniques for verifying whether a memory reference falls inside a region that is currently subject to a watch point. In other words, aspects related to the identification of areas that have been dirtied and to incremental log/restore operations are not considered.

4.1 Overview of Di-DyMeLoR's Architecture

From an architectural point of view, Di-DyMeLoR can be seen as a wrapper of ANSI-C `malloc` services[1], which is transparently interposed—via simple linking-time directives—between the simulation model's code and the traditional malloc library. These facilities are present in most compiling tool-chains, and can be triggered via ad-hoc directives/scripts. Figure 4.1 provides an illustration of this approach, along with a visualization of the supported API.

For each LP, Di-DyMeLoR keeps a metadata table composed of `malloc_area` entries. Each entry keeps information about a block of contiguous memory chunks. Each contiguous block is used to serve memory requests of a given size for that LP, thus different entries of the table are used to manage chunks of different sizes. The sizes which are handled by Di-DyMeLoR are fixed, in particular they are all the powers of 2 in between a `min_size` and a `max_size`

[1]Specifically, Di-DyMeLoR supports `malloc`, `free`, `realloc`, and `calloc` standard services offered by the standard C library.

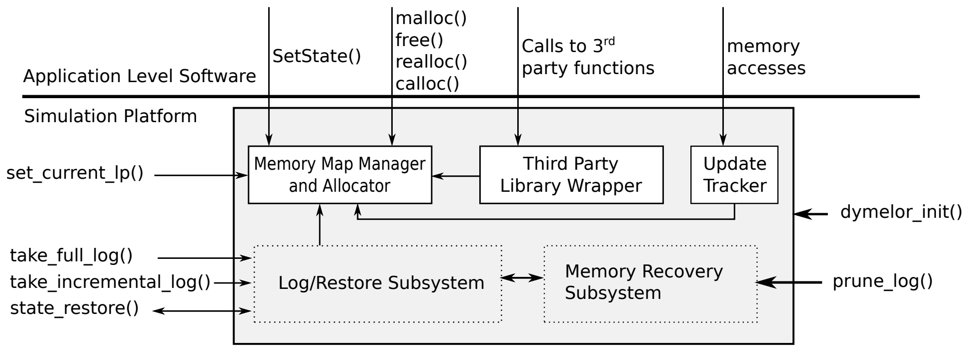


Figure 4.1: Di-DyMeLoR's Architecture

which can be specified at compile time. The default values are 32 B and 32 KB.

When a `malloc` request is issued, Di-DyMeLoR rounds up the requested size to the nearest power of 2, and then the corresponding block is allocated (via a call to the underlying real malloc services). In this way, when one chunk is required, actually a contiguous number of chunks of the same size is pre-allocated, so that future requests will be handled more quickly. This pre-allocation strategy tries to capture a "size-locality" property, in the sense that it is more likely for a simulation model to allocate chunks of a predefined set of sizes, due to the fact that the data structures used by the model are often statically specified in the code, and in a finite number. Therefore, pre-allocating chunks for any given power of 2 at simulation startup might cause a relevant under utilization of the whole memory map.

Additionally, since chunks of the same size are pre-allocated in a block, memory contiguity of the state layout is exalted for each LP. This can favour performance during both event processing and log/restore operations. In addition, pre-allocation of contiguous chunks allows Di-DyMeLoR to use very concise metadata for the identification of the status of each chunk (busy or free) within a block. In particular, a simple bitmap of *status bits* is used to identify chunks

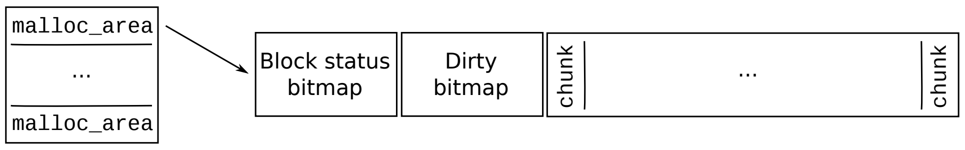


Figure 4.2: Main Memory Map Data Structures in Di-DyMeLoR.

which have been already delivered to the application, and are therefore currently in use. We will refer to this bitmap as *status bitmap*.

To further optimize memory usage, the status bitmap is placed at the head of the pre-allocated block of chunks, and gets allocated only in case of real allocation of the corresponding block (see Figure 4.2 for a schematization of the relation between the main data structures describing the LP memory map). Actually, the table of `malloc_area` entries can be expanded in case its entries have been saturated, and the LP continues requesting more chunks during event processing activities.

4.1.1 Management of the Memory Map

In order to allow rollbackable memory-management operations, the wrapper must know the identity of the LP which is scheduled whenever a call to the memory-management API is issued. Di-DyMeLoR requires the simulation kernel to explicitly notify which is the running LP identifier at scheduling time. The identifier handled by Di-DyMeLoR is an integer in the range $[0, N-1]$ which, as discussed in Chapter 2, is a classical means for mapping LPs. N is the total number of LPs locally hosted by the local simulation kernel instance, let it be a single-threaded process within a classical multi-process PDES platform, or a thread within a multi-threaded platform organization [172]

Upon the initialization of the simulation, the simulation kernel must issue a call to the API function `dymelor_init(int num_LPs)`, which is used to notify the number N of locally-hosted LPs, and which in turn allocates the aforementioned metadata table, consisting in an array of `num_LPs` entries containing the following fields:

`base_state_address` which identifies the address that should be passed to the event-processing callback upon dispatching the LP to allow it to correctly access its state in memory;

`state_layout_info` which identifies the address and the current size of a metadata table keeping information on the memory layout for the LP state (i.e., in-use chunks, dirty chunks, and reserved—but not currently in use—chunks).

The API `void set_current_LP(int LP_id, time_t sim_time)` allows a simulation worker thread to notify Di-DyMeLor what is the identity of the local LP that is currently about to execute its next simulation event. In this way, the wrapper can identify the LP metadata it must refer to upon subsequent `malloc`/`free` invocations by the simulation model's software, and can be informed about its LVT value. This is required to make memory deallocations correctly rollbackable, based on the relation between the advancement of the GVT and the simulation time associated with `free` calls (see Section 4.5 for a more detailed description of the memory-recovery approach).

LP metadata, accessible via `state_layout_info`, are organized into table entries structured as follows:

```
struct malloc_area {
    int my_index;
    int dirty_area;
    size_t chunk_size;
    int total_chunks;
    int in_use_chunks;
    int dirty_chunks;
    int next_free_chunk;
    simtime_t last_access;
    void *where;
    struct malloc_area *prev;
    struct malloc_area *next;
};
```

Each entry (which is associated with the `my_index` index, which tells the position of the entry in the table) is used to manage a block of a given-size contiguous memory chunks, and different blocks host chunks with size corresponding to different powers of 2 (as supported by standard configurations of the malloc library). The `chunk_size` field indicates the size associated with the `malloc_area` entry. The `where` field is initially set to `NULL`, meaning that the chunks' block associated with that specific size has not yet been reserved. We therefore say that the `malloc_area` entry is currently *invalid*.

When a `malloc` call is issued by the simulation model's code, the chunk size that best fits the request is identified, a block of contiguous chunks is allocated by the memory-map manager via the underlying standard malloc library, and its address is registered within the `where` field, thus *validating* that specific `malloc_area` entry. This approach implements a classical pre-allocation strategy, where a block of pre-allocated chunks is reserved for a specific LP, thus improving memory locality for its state.

By default, the virtual address returned by the memory allocator upon the very first `malloc` call for a specific LP is registered as the `base_state_address` for that LP. This choice takes into account the fact that, as discussed in Section 2.3, at simulation startup the very first activity which involves a simulation model is the setup of the LPs' simulation state, triggered by the `INIT` event's delivery. It is reasonable, then, that during the execution of `INIT` event, the simulation model's code will rely on a `malloc` call to reserve the initial memory for the simulation state.

However, if this is not the case, or if during the simulation's execution one LP decides to start using a completely different simulation state (i.e., by a couple of `free`/`malloc` calls), Di-DyMeLoR offers the `SetState(void* addr)` API, which allows the programmer to dynamically change the base state stored in `base_state_address`. Of course, relying on `SetState()` can affect transparency, as the simulation model programmer must rely on a specific API call to notify the kernel of a state layout change. Yet, our experience tells us that most part of DES simulation models have a static initial shape of the simulation state, which is then made grow or shrink at runtime, by linking new dynamically-allocated buffers to the initial one, via pointer, lists, or more complex dynamic structures. This behaviour is fully-transparently captured by Di-DyMeLoR, and has in fact driven our design principles.

We note that, in order to make the simulation state completely rollbackable, the invocation of `SetState()` is stored into the log buffers, as it will be discussed later, keeping track of the previous `base_state_address`, which can be therefore restored in case of a rollback operation.

For both time and space efficiency, each chunk within a pre-allocated block is associated with a single bit that indicates its current status, in terms of whether it is in use or not[2]. The resulting status bitmap is placed at the head of the pre-allocated block of chunks, along with a dirty bitmap, which is used to materialize memory-update operations, and which will be later described. These support data structures are created and managed only in case the corresponding block of chucks is actually allocated, thus considerably reducing the initialization time. Figure 4.2 shows the exact memory layout for the aforementioned data structures.

Upon a memory allocation request, the `in_use_chunks` counter is updated, and the field `next_free_chunk` in the involved `malloc_area` is used to identify the most convenient position for starting the bitmap search in order to select a free chunk. The manipulation of `next_free_chunk` follows the classical a first-fit policy used by the Linux kernel for managing processes' file descriptors, aimed at reducing both free-chunks and bitmap fragmentation by aggregating in-use chunks in the initial part of the block.

When a block of chunks of a given size is filled up, the metadata table is expanded via a standard `realloc` operation, leaving available at least one new `malloc_area` entry, which gets linked via the `prev` and `next` fields, creating a list of entries used to manage chunks of a given size. Also, a new block of contiguous chunks of that size is allocated. In this scenario, we detect that chunks of that size are highly useful for serving memory allocation requests for the LP. Consequently, whenever we expand the metadata table and reserve a new block of chunks, we double the block's size (i.e., the number of chunks hosted

[2]This is a main difference from the original malloc library, where a more complex header is associated with each managed chunk to maximize flexibility in memory usage (e.g., by dynamically partitioning or aggregating chunks according to the so-called "boundary tagging" scheme [91]). Nevertheless, Di-DyMeLoR exploits this flexibility by ultimately relying on the malloc library for actual virtual memory allocation.

by a block)—as in classical operating-system schemes targeting pre-reserving for the swap area destined to data sections of active processes. This strategy further enhances memory contiguity for the LP state, minimizing costs associated with malloc library accesses, due to memory blocks pre-allocation.

Upon a `free` call, the associated chunk (and the corresponding block) is not actually released. Instead it is marked in the status bitmap as available for future allocations. In this way, memory deallocations are correctly rollbackable until they get eventually committed due to GVT advancement. Operatively, this is achieved by also exploiting the `last_access` field within the `malloc_area` entry, which is used to record the logical time associated with the last memory allocation/deallocation operation within the corresponding block, and to determine whether a block formed by chunks that have all been released can be really deallocated, during the execution of the fossil collection operation.

The explicit design choice to avoid per-chunk metadata would require the scan of all the `malloc_area` entries to check whether the entry is active, and (in the positive case) whether the chunk being released via the `free` call belongs to it. To avoid such a scan, Di-DyMeLoR is equipped with a software-level direct-mapped caching subsystem, with cache lines formed by the tuple: $\langle chunk\_address, chunk\_end\_address, malloc\_area\_index \rangle$. Upon chunk allocation, the cache line is filled so that, in case of a subsequent `free` operation associated with that same chunk address, the wrapper retrieves the corresponding `malloc_area` in $O(1)$ time (unless for cases where the same cache line is overwritten during the run). A cache line is reset only when the corresponding chunk gets really deallocated, i.e. when the whole memory block containing it is released.

To allow incremental state saving, Di-DyMeLoR must know which portions of the memory map of a given LP were modified since the last log operation. To this end, the `dirty_area` flag indicates whether any type of operation (namely allocation, deallocation or chunk update) has occurred in the `malloc_area` since the last log. Therefore, this flag is set upon the invocation of a `malloc` or `free` call by the simulation model code, and is additionally set whenever a memory-update operation is performed, according to the scheme which we will present in Section 4.2. Additionally, the `dirty_chunks` field explicitly counts the current number of in-use chunks that have been updated in the `malloc_area` since the last log operation. The dirty bitmap, which has the same size of the status bitmap, associates one bit with each memory chunk, allowing a fine-grain identification of the memory areas which where updated since the last log operations, again avoiding a costly per-chunk data structure.

4.2 Simulation Model Instrumentation Technique

As mentioned, the final goal of Di-DyMeLoR is to transparently support incremental state saving. While the aforementioned approach is suitable to let the simulation kernel transparently know where the simulation state of each LP is located, to detect which memory regions are accessed (in write mode) during the execution of an event, we rely on the static software instrumentation facilities offered by Hijacker [123], which are thoroughly discussed in Appendix A.

Nevertheless, to clarify how Di-DyMeLoR updates its data structures whenever a memory-update operations takes place, we give here some insights on the instrumentation process, which specifically targets ELF (Executable and Linkable Format) [165, 105] objects generated by standard compilers for x86 and x86-64 architectures. At the very base, Hijacker works by parsing the object generated after linking together all the application-level source modules, and by identifying every memory-write instruction inside this object, namely `mov` instructions with a memory location as the destination. The instrumentation process is then supported via the insertion of a `call` instruction to an `update_tracker` module (which is part of the Di-DyMeLoR subsystem). It's purpose is to perform the identification of the exact memory address and the size (amount of bytes) involved in the memory-update operation.

Although this is a typical way for tracking memory update references (e.g., in the context of program debugging techniques [174]), the usage of this approach in optimistic simulation systems poses (more) stringent performance issues. In particular, the `update_tracker` should perform its job via very few machine instructions, in order to avoid a significant impact on event execution latency.

To cope with such a performance target we have decided not to employ runtime disassembling of the memory reference instruction, which could be onerous (compared to the event execution latency of non-instrumented software) especially due to the complexity and variable format/length of the x86/x86_64 instruction set. Instead, we cache some of Hijacker's disassembly information into a table which is accessed by `update_tracker`. Therefore, most of the disassembly overhead is paid only once at compile time, leaving to the `update_tracker` module the task of gathering a reduced set of information which are only available at runtime.

In particular, x86/x86_64 architectures identify a memory address as the linear combination of (up to) five parameters, namely `segment`, `base`, `index`, `scale` and `displacement`, as depicted in Figure 4.3. They maintain the following information:

`segment`: a segment register. This is not directly specified in the instruction, yet the addressing mode will use the segment where the currently being

$$\left\{\begin{array}{c}\text{CS:}\\\text{DS:}\\\text{SS:}\\\text{ES:}\\\text{FS:}\\\text{GS:}\end{array}\right\}\left[\left\{\begin{array}{c}\text{EAX}\\\text{EBX}\\\text{ECX}\\\text{EDX}\\\text{ESP}\\\text{EBP}\\\text{ESI}\\\text{EDI}\end{array}\right\}\right]+\left[\left\{\begin{array}{c}\text{EAX}\\\text{EBX}\\\text{ECX}\\\text{EDX}\\\text{EBP}\\\text{ESI}\\\text{EDI}\end{array}\right\}*\left\{\begin{array}{c}1\\2\\4\\8\end{array}\right\}\right]+[\mathit{displacement}]$$

Figure 4.3: x86/x86_64 addressing mode

executed instruction is located;

`base:` this value is stored into one of the 8/16 general-purpose registers of the processing unit[3] (which is therefore referred to as *base register*) and is commonly used to represent a base address from which to compute a final memory location to access;

`index:` this value is stored into one of the general-purpose registers as well, hence being called *index register*. It is commonly used to represent the index of an array, the base of which is stored into the base register, or in the offset;

`scale:` this variable, which can only assume the values 1, 2, 4, 8 only, is a multiplier of the `index` value. It is directly coded into the instruction's binary representation, and is used, e.g., to represent the width of the data which compose the array being accessed;

`displacement:` this value, directly stored into the instruction's binary representation, is finally added to the outcome of the memory address evaluation.

It is clear that this complex addressing format simplifies the CPU user when dealing with more complex data structures like `struct`s or arrays. Yet, as hinted before, some portions of the addressing format can be only evaluated at

[3]Depending on whether the architecture is a 32 bits or a 64 bits one.

runtime, namely the base and the index registers. All the other information can be gathered at compile time by looking at the instruction's opcode[4] which tells which of the four fields (the segment not being specified in the instruction) are relevant for the address evaluation and what is the actual size of the involved memory operation.

Hence, cached data from the disassembling of one single instruction, are organized as follows:

```
struct update_tracker_entry {
    unsigned int size;
    char flags;
    char base;
    char index;
    char scale;
    long displacement;
};
```

The `flags` field is used to identify which of the aforementioned four parameters are actually relevant and should be considered by `update_tracker` for computing the exact address for the memory-write operation. Also, the `size` field immediately indicates to `update_tracker` the (compile-time defined) size of the memory area to be dirtied by the current memory-write instruction.

We have two exceptions to this approach. One is for `movs` and `stos` instructions, used for moving arbitrary-size memory blocks. These instructions keep the information for identifying the destination address and the current size

[4]For simplicity, we call *opcode* the actual opcode, the prefixes, the ModR/M and the SIB byte of the x86/x86_64 instruction set, which can be all present, or only some of them, depending on the actual instruction and addressing mode. We refer the reader to [72, 73] for a complete discussion of the instruction set.

of the memory block being written into predefined registers, namely `edi` and `ecx`, which are directly accessible by `update_tracker`. The second one involves `cmov` instructions, which perform the memory update only if a particular condition is met. To cope with this specific case, we replace the `cmov` instruction with an assembly code block which mimics the execution of this instruction (i.e., it performs the equivalent check), and if the condition is met, then the memory is updated by relying on a traditional `mov` instruction, which is in turn instrumented according to the same aforementioned policy. We note that this approach is sub-optimal with respect to the execution performance of the `cmov` instruction. Nevertheless, we note that to mark the memory area as updated, we necessarily must evaluate the condition. At that point, relying again on the `cmov` to perform the memory update would add an additional (although minimal) cost, just to compute a value which is already available.

Recalling that the execution of `update_tracker` module is a performance-critical operation directly impacting the event-execution cost, we have adopted the following strategy for minimizing the performance overhead. In particular, for each `mov` instruction involving a memory update, a set of `push` instruction in injected before the actual `call` to the `update_tracker` module. The purpose of the `push` instructions is to let `update_tracker` find on the stack a memory area structured as `struct update_tracker_entry`, where the value of the fields describe the original `mov` instruction which caused the actual invocation of the module.

Upon its activation, `update_tracker` checks inside its own stack frame the information needed to compute at runtime the memory address and the size of the write operation. Given that this computation can unpredictably change the value of the `eflags` register on board of the CPU, this register value is saved by `update_tracker` upon its activation together with general purpose ones, and is put back in place right before returning control to the memory write instruction for which the tracking process has been activated.

In the memory model offered by Di-DyMeLoR, locations associated with automatic variables (allocated inside the stack) do not belong to the object memory map, since they do not survive across different invocations of the event handler. Hence, all those memory-write instructions that can be detected at compile-time to access the stack (e.g., `mov` instructions addressing memory via base pointer or stack pointer displacement) are not actually instrumented, by relying on a special configuration rule. Anyway, in some cases write access into the stack cannot be recognized at compile time. For this reason, after having computed the address for the memory-write operation, `update_tracker` compares it with the current value of the stack pointer. In case the access is an actual stack update, `update_tracker` simply returns. Otherwise, the information about the identified memory address and the size of the area being dirtied is passed to the memory map manager. This is done by invoking an internal routine which flags the dirty bit corresponding to the involved memory chunk(s). For this task, the software cache described in Section 4.1.1 is exploited again in order to perform a reverse query which translates a generic memory address into the chunk(s) actually containing the memory buffers and the associated `malloc_area` entry. This allows fast identification of the bitmap to be involved in the update operation. Additionally, `dirty_area` is set, and `dirty_chunks` is incremented by the number of chunks which were involved by the memory-update operation.

While Hijacker is able to rebuild all the internal references between instructions and data, which are altered by the insertion of additional instructions in the ELF text section, the x86/x86_64 instruction set allows to compute at runtime the destination of the so-called *indirect branches* (also referred to as *register jumps*), where the destination address is dynamically identified via the content of CPU registers. To cope with this issue, we have implemented a second run-time monitoring mechanism for supporting on-the-fly correction of destination addresses of indirect branches. Like in the aforementioned approach, this mechanism is based on the insertion of a `call` instruction to a second assembly-level monitoring module, called `branch_corrector`, prior to each indirect branches in the original software. This monitoring module relies on a pushed data structure which is associated with a single register jump instruction, and keeps the information regarding which are the registers whose values determine the destination address for the jump operation. This process is carried out in a way similar to the one adopted for the generation of support information for `update_tracker`.

By exploiting the information in this data structure, `branch_corrector` evaluates the original destination address for the `jmp` instruction (by reading the CPU registers that specify the destination value). Then it corrects this address on the basis of the amount of bytes by which the original destination was shifted inside the instrumented object layout. To provide a lightweight mechanism for address correction, we generate a table at compile-time, which is visible to `branch_corrector`. Each entry inside this table identifies an interval of addresses for which the instrumentation process gave rise to the same amount of shift inside the final (instrumented) memory layout. Such an offset is also maintained in the table entry. The table is ordered by interval extremes, and `branch_corrector` performs a logarithmic-cost binary search to retrieve the interval containing the original destination for the register jump, and the offset to be applied for the correction.

Such a correction cannot however be applied by modifying the values of the CPU registers involved in the `jmp` instruction. This would otherwise result in an inconsistent processor state for the simulation model. We have rather adopted a different approach where the original indirect-branch instructions (whose relevant information is anyway logged inside the pushed data structures available to `branch_corrector`), are replaced at compile-time by Hijacker with (regular) offset jumps (not relying on CPU registers), where the destination address is maintained inside one field of the instruction binary representation, and is appropriately set by the on-the-fly correction mechanism.

To support the rewrite operation of the appropriate instruction field at run-time, without impacting typical settings associated with memory protection, the indirect-branch instruction has been moved inside a run-time re-writable ELF section (specifically created by exploiting compiler facilities). Also, a jump-label instruction has been inserted in place of the offset jump inside the original (non-rewritable) section(s) of the application code, which passes control to the offset jump right after the `brach_corrector` module has re-written the correct destination address (the offset) inside the ad-hoc re-writable section. Of course, in case of simulation kernels which are implemented according to a multi-threaded scheme, the instrumentation process involves the generation of one new writeable section per each thread, to avoid race conditions. The whole instrumentation process is illustrated in Figure 4.4.

We note that efficient solutions for correcting register jumps (e.g., via the avoidance of run-time disassembling) have practical relevance since register jumps are typically generated by standard compilers for machine language translation of `switch`/`case` constructs [164], which are relevant in simulation applications for, e.g., flow control inside the event handler(s) on the basis of the type of dispatched event.

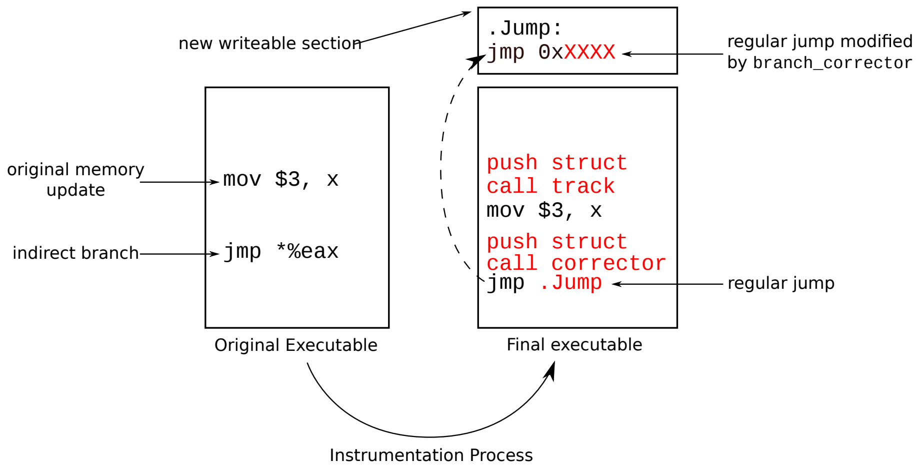


Figure 4.4: Simulation Model Instrumentation Process

4.3 State Log Operations

Basically, a state log operation is supported by packing the information to be logged inside a contiguous buffer allocated via the underlying malloc services. Incremental State Saving is supported by a set of operations which depends on the current value of the data structures explicitly used for tracking dirty data/metadata. Specifically, whenever a snapshot must be taken for a given LP, each active `malloc_area` is checked to determine which, among the following cases, is verified:

1) `dirty_area` is set and `dirty_chunks` is zero. In this case the `malloc_area` is packed into the log buffer together with the status bitmap indicating the current allocation of chunks inside a given block. Yet the dirty bitmap and the currently in-use chunks are not logged.

2) `dirty_area` is set and `dirty_chunks` is greater than zero. In this case the `malloc_area` is packed into the log buffer together with the status

bitmap, the dirty bitmap and the chunks that are currently in use, which have been dirtied. All the other in-use chunks are not logged.

3) `dirty_area` is not set. In this case, no information associated with the area is logged at all.

Finally, all the data structures used for tracking dirty data/metadata are reset, independently of which, among the aforementioned cases, occured. This is because all the information related to the modification of the `malloc_area` has been saved in the just-taken log, so the system can restart tracking new changes from scratch. Then, the state log is stamped with the current LVT value, and is inserted into the timestamp-ordered state-log chain. Additionally, the current value of `base_state_address` is stored as well in the snapshot, thus allowing the recoverability of `SetState()` invocations.

Again, we emphasize that incremental state log operations are not required after the execution of each simulation event, rather they can be taken periodically. In fact they are based on recognizing memory portions that have been dirtied since the last log, independently of the amount of events actually performing the dirtying operations. Hence, state reconstruction at whichever simulation time can be supported via a mix of state restore from the log, and classical coasting forward.

4.4 State Restore Operations

When a restore operation needs to be executed to restore the execution at ST $T_{rollback}$ due to the occurrence of a rollback operation, the log chain is searched to determine the most recent log with time less than or equal to $T_{rollback}$, and all the logs with time greater than $T_{rollback}$ are simply discarded since they refer to causally-inconsistent memory maps. To restore the incremental log, the following steps are then iterated by backward traversing the chain of logs:

1) A `malloc_area` found inside the log buffer, which has not been restored, is put back in place inside the metadata table. The associated status bitmap is also copied back from the log buffer (recall that a logged `malloc_area` is always associated with the corresponding status bitmap inside the log buffer to guarantee recoverability of chunk allocation/deallocation operations).

2) Each dirty chunk which is found inside the log and which is associated with the `malloc_area`, which has *not yet* been restored in a previous iteration while backward traversing the log, is copied back in its correct position inside the corresponding memory block.

The iterative restore procedure stops when all the active `malloc_area` entries have been restored and all the in-use chunks that have been dirtied are also restored. Although in principles this could entail an indefinite number of iterative backward steps along the log chain, in practice the restore operation can be immediately finalized if we find a full log while backward re-traversing the log chain. In fact, all the in-use chunks that have not yet been restored are immediately available inside the full log for copy-back operations. Therefore, we periodically take a full snapshot of the simulation state, to avoid a (possibly) infinite restore operation. To correctly select the optimal interval according to which take a full log, in order to minimize checkpoint/restore latency and thus maximize the overall simulation throughput, an approach like the one proposed in [168] can be adopted. The steps executed to perform an incremental restore operation are depicted in Figure 4.5

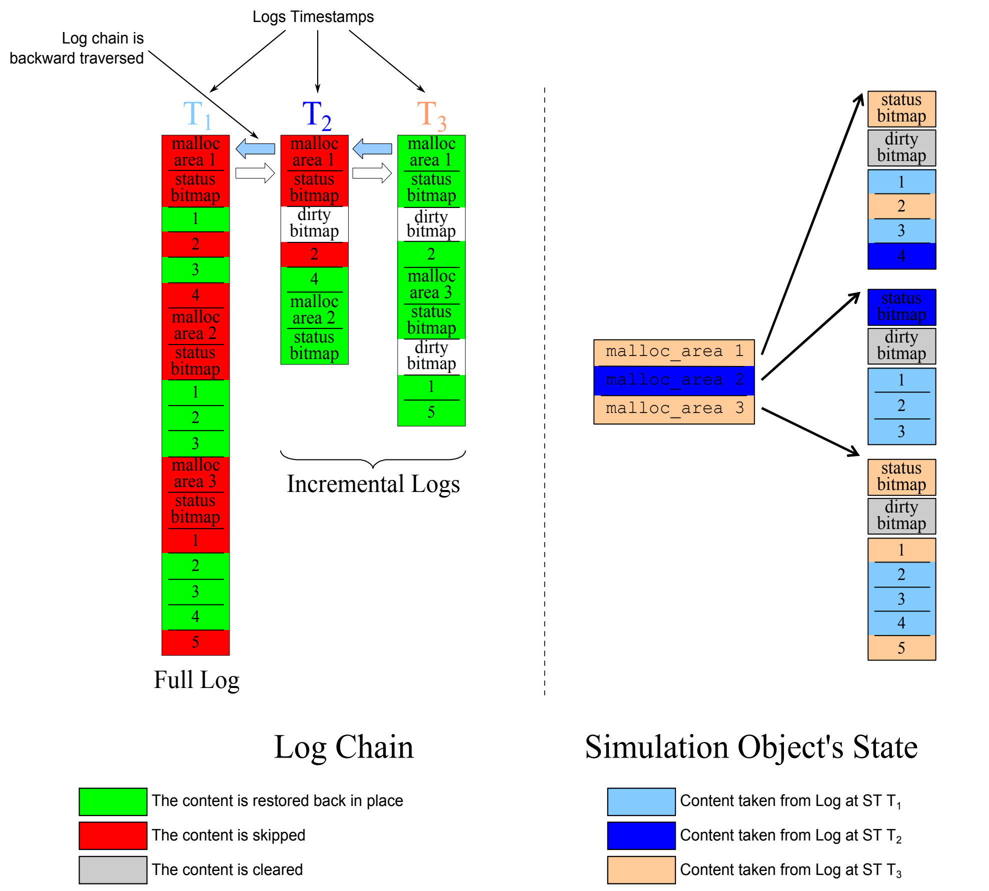


Figure 4.5: Incremental Restore Process

Actually, to optimize the detection of already-restored chunks, which must therefore not be copied-back again from the log, the iterative restore procedure is based on temporary bitmaps (each one associated with an active `malloc_area`) on which a couple of fast bitwise OR-XOR operations are executed each time a dirty bitmap (associated with that same `malloc_area`) is extracted from the incremental log.

4.5 Memory Recovery

As explained, incremental logs are linked all together within per-LP lists, sorted by simulation time value. Obsolete logs can be discarded, thus allowing virtual memory recovery, via the `void prune_logs(time_t new_GVT)`. This function scans the log queue for each managed LP, finds the oldest full log with time less than or equal to the value of `new_GVT`, and prunes all the logs with a lower simulation time.

Given the organization of the aforementioned recovery procedures, maintaining at least one full-log with time less than or equal to the newly computed GVT value allows correct recoverability of the LP state.

Whenever the log chain is pruned, an internal API function is triggered, to check whether some memory blocks (allocated via the underlying real malloc library) are keeping chunks which have all been free'd by the simulation model. In this case, the `free` operations can be finally committed, i.e. the buffers which were being kept to allow the recoverability of the `free` operations can be discarded.

4.6 Third Party Library Wrapper

The possibility to rely on third-party libraries depends on whether they modify the internal simulation state of the calling LP or not. In case they do, we need to capture memory changes caused by their execution, let them be new allocations or memory updates. However, libraries are not instrumented, and therefore we cannot rely on allocation wrappers and `update_tracker` for memory-update detection.

We have explicitly addressed the case of update operations performed by third-party software, just focusing on `stdlib`. Specifically, Di-DyMeLoR provides a set of function wrappers for all those functions which produce in-memory changes by the application-level software through pointers passing. The wrappers simply throw back the call to the underlying standard-library function, and then pass control to the memory-map manager with explicit indication of the address of the updated buffer, and the size of the updated memory block. In case the size cannot be retrieved by the library function signature (as for pointers to buffers used for strings), the memory-map manager updates the dirty bits for all the currently allocated contiguous chunks starting from the pointed address. This is obviously a conservative way of managing the memory map since some chunks that have not been really dirtied by the library are actually considered as dirty ones, thus being subject to log/restore operations. However, correctness is not jeopardized, given that the wrapped `stdlib` library functions are all stateless, thus posing no issue on the side of memory log/restore.

As for `stdlib` functions which allocate new memory buffers (e.g., `strdup`), Di-DyMeLoR provides a set of wrappers as well, which in turn re-implement the library functionalities by relying on its memory allocator, thus allowing the new memory buffers to be located within the actual LPs' state layout. Allocation/deallocation and update operations of these buffers are therefore made recoverable. We are currently working on techniques for allowing the application code to automatically rely on any (stateful) third-party library.

4.7 Experimental Evaluation

4.7.1 Benchmark Applications and Configuration

To study the effects of Di-DyMeLoR when considering differentiated execution and memory access patterns for the application layer, we use two different configurations of the PCS application. In one configuration we simulate 1024 cells, each one managing up to 1000 wireless channels, where the expected duration of a call $\tau_{duration}$ has been set to 120 seconds, the residual residence time for an active call in the current cell τ_{change} has been set to the value 300 seconds, while the inter-arrival time τ_A has been varied during the simulation so as to generate a configuration where the actual load on the cells depends on the period of the day.

Specifically, 17 hours of operation of the cellular system have been simulated (from 00:00 AM to 17:00 PM) with variations of τ_A in the interval $[0.64, 3.20]$, with peak intensity of the workload during the morning until lunch time, and minimum load very early in the morning (around breakfast). Consequently, the utilization factor has been varied in the interval $[0.31, 0.06]$. For this configuration of the PCS model, climatic conditions have been set as good and steady, thus not causing the need for frequent recalculation of fading coefficients. We will refer this configuration to as “Variable τ_A”.

On the other hand, the second configuration of PCS has been parametrized by having the expected inter-arrival time τ_A fixed to the value 0.8 (giving rise to channel utilization values on the order of 25%), which leads to focusing the simulation on a morning operation scenario, but where the climatic conditions exhibit variations that lead to periods where frequent recalculation of fading coefficients needs to be operated. We will refer this configuration to as "Frequent fading recalculation". Both the above configurations lead to run time dynamics that vary, e.g., in terms of event granularity and portion of the LP state that needs to be updated by the events, however this is achieved in different manners in the different scenarios.

Additionally, to study the behaviour of Di-DyMeLoR on a simulation scenario where the size of the simulation state is large, we have relied on the NoSQL benchmark. In our experiments we considered a configuration with 64 cache-servers, each keeping $M = 50000$ data objects. This leads a global occupancy of about 64 MB of current simulation state in our implementation. Considering that each data object is replicated 2 times, the total number of different $\langle key, value\rangle$ pairs kept within the simulated data-grid platform is equal to 1.6 million. In our experiments, we set to 100 the expected number of read/write operations executed by any transaction, and to 0.2 the probability to access a remote data-object within the data-grid system (namely a $\langle key, value\rangle$ pair whose copy is not held by the local server neither as master nor as backup in the replication scheme). Also, we run experiments where the data-access profile of the transactions changes over time. Specifically, we initially have a read intensive phase, where each transaction updates only the 5% of the accessed data-objects (while accessing the remaining 95% in read mode). On the other hand, in a subsequent execution phase, the transaction profile becomes write intensive, meaning that the data objects accessed in write mode amounts to the 50% of the total number of per-transaction accessed objects. We note that changes in the read vs write intensiveness of transactions in a data-grid system is prone to take place in relation to real-life events (e.g., associated with relevant promotional sales on stocks of products in e-commerce applications). For each phase (read vs write intensive) we simulated on the order of 320000 transactions. We note that in this model events have a relatively fine granularity (on the order of 10/15 microseconds) independently of the transaction profile. In fact, differently from the PCS benchmark, no complex calculations need to be performed for handling the events. Rather, event processing mostly entails retrieving the information on the target data-object, and updating its state (e.g., by temporary locking the data-object in case of the prepare phase in the 2PC protocol). This leads to a scenario where fine grain events are coupled with relatively large size of the LPs states. In principles, this scenario is not favourable to full checkpointing, unless for the case where a large fraction of the state is updated by subsequent events, which is the case for the write intensive phase. Particularly, when the transaction profile is write intensive, a larger number of data-objects need to be locked in the 2PC scheme, and then updated upon committing the transaction. Given that the state of each object is kept in an entry of a bucket (an array) in the hash table, then updating the entry leads to mark as dirty the whole chunk keeping the array. This leads to increasing the cost of incremental logging (compared to read intensive scenarios).

4.7.2 Results

We report in Figure 4.6 and in Figure 4.7 the cumulated committed events achieved by the parallel run versus WCT for the PCS benchmark. These values have been computed as the average over ten runs (performed with different pseudo-random seeds), with a minimal variance observed across different runs.

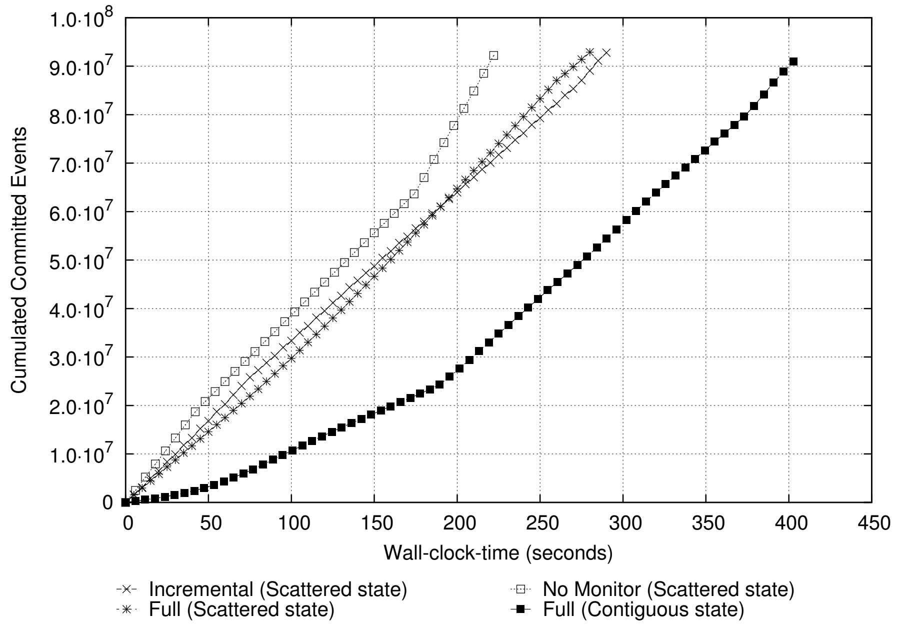


Figure 4.6: PCS – Variable τ_A

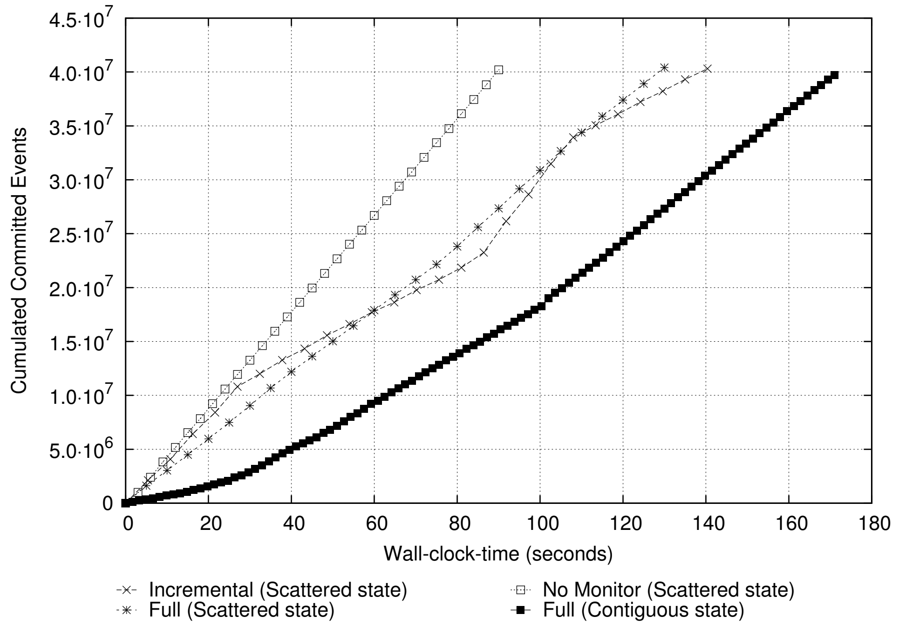


Figure 4.7: PCS – Frequent Fading Recalculation

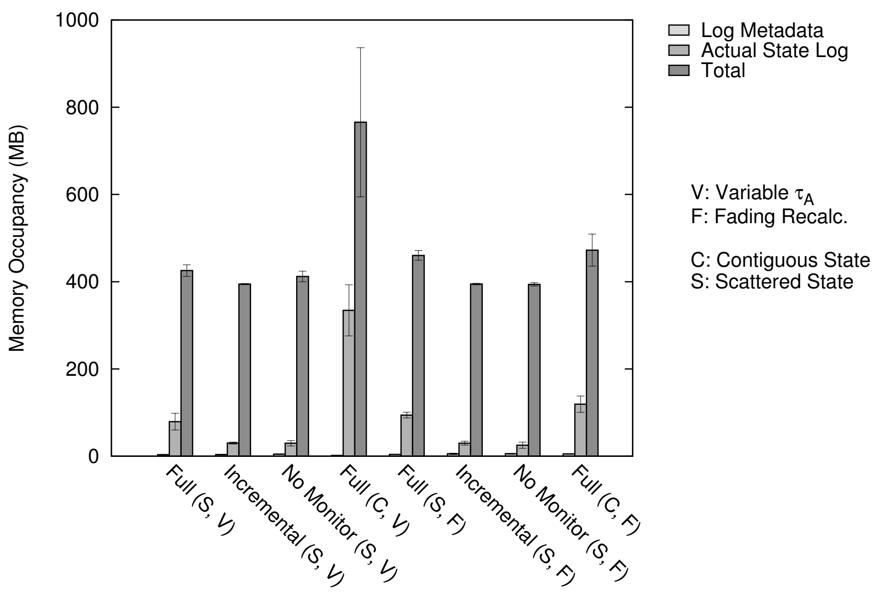


Figure 4.8: PCS – Memory Usage

This parameter (and the slope of the associated curve) indicates the speed according to which a given platform configuration commits simulation events, and hence how fast the configuration supports model execution.

We report four plots referring to:

1) the case when Di-DyMeLoR is used to take incremental snapshots, with the value of χ optimized according to [140];
2) the case in which ROOT-Sim is configured to take full logs (according to the implementation described in [166]) rather than incremental ones as for the services offered by Di-DyMeLoR, again with the value of χ optimized according to [140];
3) the case where we support the incremental log mode by having the simulation model directly calling the memory map manager to notify which

portions of the state have been updated by event processing (this is referred to as “No Monitor” in the figures);

4) the case in which the application code was modified to avoid using dynamic memory, hence leading to the situation where the state buffer for each LP is pre-allocated at startup in the form of an array of entries.

The plots for cases 1 and 2 express performance levels related to the incremental/full log scheme. On the other hand, the plots for case 3 allow us to assess the effects generated by the memory update facilities which are exploited to take incremental logs transparently to the application code. In fact, case 3 represents scenarios that benefit by optimal checkpoint interval calculation and incremental state log/restore, but require the intervention of the programmer in relation to some of the tasks enabling incremental logging, thus offering a transparency level which is strictly lower than the one offered by Di-DyMeLoR. Hence, this case allows quantifying the performance penalty associated with state-management full transparency. Finally, case 4 is representative of scenarios where no facilities other than the bare minimal log and restore operations are supported, and without any infrastructure allowing for dynamic memory handling, thus requiring the state to be contiguous and statically sized to the maximum value admitted by the model parametrization. This is a baseline for our experimental evaluation.

By the results, we see that, depending on the specific phase within the simulation run, (e.g., early morning vs lunch time for the variable τ_A configuration) incremental and full modes alternately exhibit better execution speed (which is indicated by the different slope of the cumulated committed events curve while the run is in progress). In particular, for the scenario where $\tau_{arrival}$ is varied, the incremental mode is slower while simulating heavier load periods. This reflects the fact that, during heavier load periods and in the weekend, each GSM cell, and hence each LP, exhibits a reduced state size due to the minimal number of records allocated for ongoing calls, and therefore the cost for memory-update tracking is not amortized by the cost of state checkpointing. This is not the case for day-time periods, where the state size of the LPs can grow significantly (especially for rush hours), up to the limit of slightly less than 70 KB (per each LP), and the update pattern of the state upon the occurrence of the events allows the incremental log mode to outperform the full one, once the corresponding log period get optimized.

Configuration 3 removes all the transparent facilities which are provided by Di-DyMeLoR—in terms of identification and notification to the memory map manager of the portions of the LP state that have been dirtied. Finally, compared to the baseline configuration 4, Di-DyMeLoR has a throughput increase up to 25%, which indicates how an enhancement in programmability (via the transparent support for dynamic memory allocation) is strictly coupled with a non-negligible performance increase. It is nevertheless interesting to note that the slope curves related to configurations 1 and 3 are the same in all the phases in which the incremental checkpointing mode actually provides some benefits over the full checkpointing mode. This is related to the fact that, whenever the state is scarcely updated, the cost paid to monitor memory updates is completely amortized.

In Figure 4.8 we report average per-process memory usage for all the considered configurations. In particular, we show average memory occupancy for the whole simulation process (i.e., simulation-platform layer and application-level model), for state logs, and for metadata log. In all the runs we have set the GVT (and memory recovery) period to 1 sec, which gives rise to negligible coor-

PCS configuration	execution time (seconds)	speedup by the parallel run with Di-DyMeLoR
variable τ_A	6400	21.33
frequent fading recalculation	4442	31.72

Table 4.1: Di-DyMeLoR: Speedup values

dination overhead (given the tight coupling of the underlying architecture) while allowing prompt release of memory buffers. Also, the memory usage samples refer to the state of the processes as observed right before performing memory recovery. As we can see, memory requirements for metadata is very reduced (on the order of 1%) in any configuration, highlighting memory efficiency by the data structures keeping track of memory allocation. The overall average memory occupancy shows a greater variance when dealing with phase-interleaved configuration of the PCS benchmarks, due to the fact that some phases execute more coarse-grained events and therefore require less logs per time unit. In both the frequent fading recalculation and the variable τ_A configurations, the full snapshot execution mode has a higher memory requirement, which is a predictable result due to the higher amount of information which is stored into a snapshot. However, such a memory consumption remains significantly lower than the one for the baseline case 4, especially for the variable τ_A configuration, which gives rise to better locality still favouring performance. At the same time, the configurations relying on the incremental snapshot mode and the one where the application layer calls the memory map manager to explicitly update the dirty portion of the memory map show a memory usage for logs which is very comparable, indicating similar dynamics in terms of logging frequency, which confirms that the impact on performance by Di-DyMeLoR is essentially related to the overhead for transparently handling memory updates via instrumentation.

We also report in Table 4.1 the execution time for running the PCS appli-

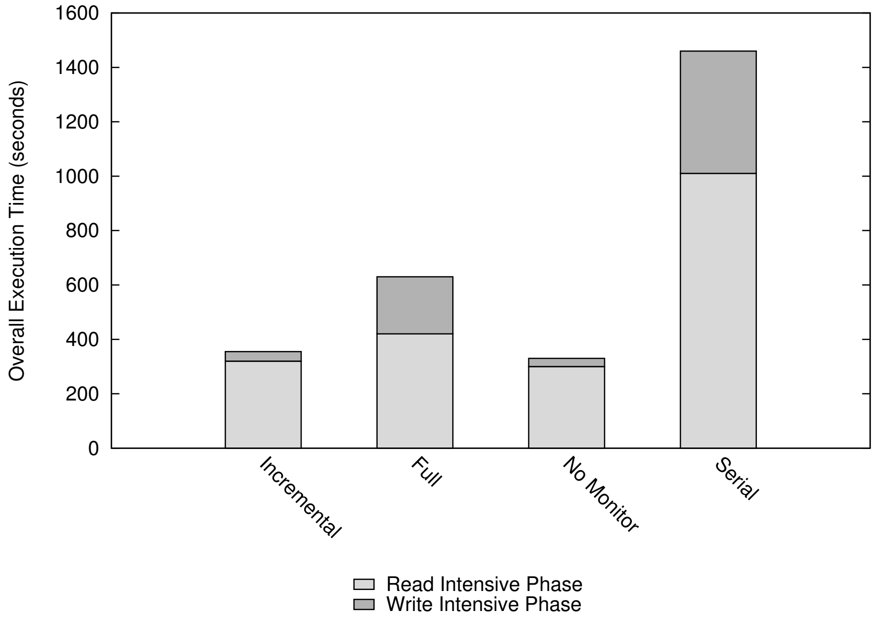


Figure 4.9: NoSQL – Execution Time

cations (very same code used for the parallel runs) in serial mode on top of a calendar queue scheduler. By the data, the parallel runs with Di-DyMeLoR allow significant speedups, especially for the frequent fading recalculation setting. Overall, the experimental study has been carried out with competitive parallel executions.

As for the NoSQL case, in Figure 4.9 we report total execution times (again, averaged over 10 different runs) for the configurations where incremental checkpointings are taken by relying on Di-DyMeLoR, where the traditional (full) checkpoints are taken, where incremental (non-transparent) checkpoints are taken, and we additionally report serial execution times for comparison.

By the results, we see that the difference between the transparent incremental and the non-transparent incremental (i.e., No Monitor in the Figure) execution times (in both write-intensive and read-intensive phases) is negligible. This is because, since the simulation states are very large, the cost for tracking memory accesses is completely amortized. When running the write-intensive phase, the speedup offered by the incremental checkpointing mode with respect to the full one is in the order of 25%. This is related to the fact that, although only 50% of the simulation state is accessed in write mode, the (large) size of the simulation state still provides a benefit when the log operation is executed incrementally. This becomes even more evident in the read-intensive phase, where only 5% of the simulation state is accessed in write mode, and therefore the speedup exhibited by the incremental approach is in the order of 40%.

Comparing the results to the serial run, we have that the speedup with respect to the incremental checkpointing mode is in the order of 70% and 92% for write-intensive and read-intensive phases, respectively, while with respect to the full checkpointing mode it is in the order of 58% and 53%, showing again that the experimental study has been carried out with competitive parallel executions.

Chapter 5

Interacting with the Outside World

Our senses enable us to perceive only a minute portion of the outside world
— Nikola Tesla, The Transmission of Electrical Energy without Wires as a means for Furthering Peace, 1905

Optimistic Synchronization divides the execution of the simulation into two main distinct parts. On the one hand, we have the *committed portion* of the simulation, which is composed at any WCT instant t by all the committed events e_C associated with timestamps $T_{e_C} < GVT(t)$, where the GVT value is computed according to Definition 2.1. This committed portion of the simulation can be regarded as *safe*, meaning that no rollback operation occurring after WCT t can affect its validity. On the other hand, we have the *speculative portion* of the simulation, which is composed of executed (but uncommitted) events e_U which are associated with timestamps $T_{e_U} \geq GVT(t)$, which is a part of the simulation trajectory which can suffer from causality violations due to the absence of any consistency control before the execution of events.

While we have shown so far how the rollback operation is able, whenever a causality inconsistency is detected, to undo a portion of the speculative simulation trajectory in order to bring back the execution to a consistent state, we have not dealt with the problem of *interaction with the outside world* in this particular uncommitted portion.

In fact, the validity of the rollback operation relies in that all the actions executed during the (speculative) execution of an event can be undone. Yet, this is not always the case. We can enumerate non-minimal series of operations which cannot be undone:

- generation of output on a file or on a console screen;
- packet delivery to a network interface;
- interaction with any hardware device (e.g., a printer);
- invocation of the `exit()` function to tell that the simulation is complete;
- more in general, invocation of any system call exposed by the underlying operating system.

In case any of the aforementioned operations is executed within an event which is later detected as belonging to an inconsistent portion of the simulation trajectory, its effect cannot be undone. This is because the services which provide the API for executing such operations (like, e.g, a disk, a network interface, or even the operating system) are not aware of the speculative nature of the operations.

Given that we want to provide the simulation-model developer with full transparency, we must assume that she is neither aware of the parallelization technique used to deploy her sequential code onto a parallel architecture, nor of the adopted optimistic synchronization strategy. Therefore, we can suppose that she could actually invoke non-rollbackable output operations within any event handler present in the simulation model. In fact, this would be a natural way for the developer to, e.g., audit the current evolution of the simulation state trajectory for statistics collection. We propose a support which enables a timely and consistent interaction, which can be significant for the aforementioned scenario and for all the ones where the evaluation of (possibly unstable) predicates should be done on a global scale, via audit on individual LPs' state trajectories.

In this chapter, we cope with the transparency versus performance issue for what concerns the production of output streams in optimistic simulation runs. Particularly, we focus on how to efficiently and transparently tackle the output commitment problem [41] (see Figure 5.1) in optimistic simulation, thus allowing the simulation-model developer to execute any non-rollbackable operation within an event handler. Our approach moves the burden of materializing the actual output from the model developer to a specifically-targeted runtime subsystem which, at the same time, provides:

- *consistency*, by finalizing the actual output operation only if it is associated with an event which gets committed, and by system-wide ordering the output messages along the simulation time axis (as if they were produced by a sequential run of the same model);
- *efficiency*, by introducing only a small overhead at the side of the engine running the simulation model;
- *timeliness*, by placing only a reduced delay from the generation to the materialization of the output on the specified stream, unless for applications constantly exhibiting I/O-bound profile along the whole run.

Although our proposal explicitly targets shared memory systems, as it will be clearly depicted by the forthcoming discussion, the proposed architecture is perfectly compatible with a distributed scenario as well. In fact, the discussion will show how the same architecture can be easily mapped onto a simulation framework which can be even geographically distributed, requiring only a small implementation effort at the side of the additional subsystem which we are about to propose.

Traditionally, PDES simulation frameworks (see, e.g., [67, 103, 156, 75]) have relied on ad-hoc APIs for delivering consistent output related to the (progressively advancing) state trajectory while executing the simulation model, in order to (incrementally) provide, e.g., model statistics. This forces the model writer to look more deeply into the internal details of simulation supports, making the development of the application more cumbersome and more constrained to the specific programming model supported by the framework. On the other hand, in case the modeller would rely on non-consistent output production via the exploitation of standard I/O libraries, the output streams would need to be post-processed for cleaning them from messages produced by events that were possibly rolled-back, and for providing a unique globally timestamp-ordered trace, if needed.

Our proposal is based on the concept of an ad-hoc *daemon*, i.e. a user-space process *separated* from the actual simulation framework. Once equipped with our kernel's output subsystem, the simulation kernel worker threads within the framework can communicate with the local daemon process via in-memory, fast-access, shared data structures. They are used to support an additional level of temporary buffering which is specifically optimized to reduce the cost of managing output streams (and their consistency) at the side of the simulation kernel (as opposed to the typical buffering rules/schemes used by standard I/O libraries). Particularly, via this additional buffering subsystem we achieve

- *reduced interaction time*, by developing a non-blocking algorithm for accessing shared memory segments from the simulation kernel worker threads and the output daemon—a scenario where a near-to-zero-cost logical output device is accessible by the simulation framework;
- *enhanced scalability* (even at geographical scale), allowing to process the commitment of the output on a multi-level basis; particularly, the daemon exploits GVT computation already natively performed by the simulation kernel in order to finalize the production of the output streams, which avoids the need for I/O-specific consensus protocols to be run.

As an additional advantage, the output daemon can run as a *stream forwarder*, allowing the user to retrieve output on a separate (dedicated) machine with respect to the cluster of machines used for model execution, in case a distributed simulation run is executed.

5.1 Output Management

5.1.1 Involved Issues

Consistent output management in optimistic simulations is a non-trivial task, due to the presence of the rollback operation, which is not handled by common libraries for managing output streams. Let us consider a simulation scenario where two LPs are present, namely LP_1 and LP_2. LP_1 is scheduled for the execution of event e_1, associated with timestamp T_1. During the execution of e_1, some output is produced via a call to the standard library's `printf()` function. After e_1's execution is completed, LP_2 generates a new event e_2 destined to LP_1, and associated with timestamp $T_2 < T_1$. According to the optimistic synchronization scheme presented in [76], e_1 is undone and the execution is

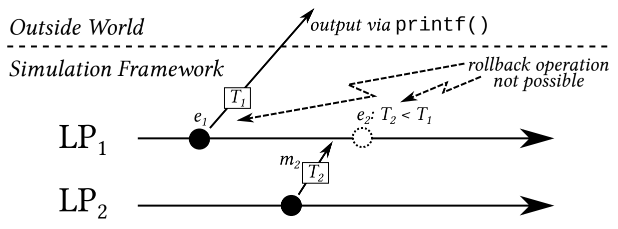


Figure 5.1: Non-Rollbackable Interaction with the Outside World

restarted from e_2. However, the `printf()` invocation is already completed, the output has been already materialized on the standard output stream, it is inconsistent as it is associated with an undone event, but no corrective action can be carried out. This is depicted in Figure 5.1.

The task of generating consistent output is even more complicated if we consider two different points of view on the same issue. On the one hand, the application-level programmer is interested in the *ease of use*, i.e. she does not want to rely on complex APIs which force her to be aware of the notion of rollback, and to look at the internal organization of the simulation engine. On the other hand, consistently managing output streams must not make the simulation engine pay a relevant performance penalty, in order to minimize the overhead on the actual simulation execution. This tradeoff between *application transparency* and *simulation performance* must be well balanced, in order to effectively and efficiently support the management of output streams.

In order to address *transparency*, the architecture which we hereby propose is again based on linking time facilities that, before creating the final simulation model's executable, redirect every output call in the application-level code from the standard printf-family functions to an ad-hoc simulation kernel's module, which represents the kernel's output subsystem within the architecture.

On the other hand, in order to enforce *high performance* of the simulation execution, all the operations associated with the management of the output must exhibit an overhead—both direct and indirect—as small as possible. As for direct overhead, we have explicitly designed our solution in order to minimize the use of system calls during the execution of the simulation for buffering the produced output until the generation-associated event is committed. In particular, at simulation startup, we pre-allocate a set of shared memory segments which are used by simulation kernel worker threads to store any output on any stream produced by the application-level code. These buffers are read, during the simulation run, by a separate user-space process, which we refer to as *output daemon*. This daemon is in charge of collecting all the output produced by the LPs, sort it, and (in case of a rollback operation) remove the involved strings. Whenever a set of events gets committed, the output daemon is able to use this information to actually materialize the relevant (committed) output on the associated streams.

As it will be thoroughly discussed later, this explicit design choice gives us several benefits:

1) output buffering is completely handled in user space (by relying on shared memory) avoiding mode/context changes during the execution of the simulation;

2) simulation efficiency is preserved, as the operations for producing the output (independently of its eventual commitment) do not involve costly procedures, giving most of the available CPU time to actual (relevant) computation;

3) considering that each worker thread of the simulation kernel is assigned a private shared memory segment, and considering that they only write on it, and the output daemon only reads from it, a *non-blocking algorithm* for the

interaction between kernel instances and the daemon can be implemented, providing additional benefits due to the avoidance of contention on logical data (i.e., no locking primitives must be exploited to allow kernel–daemon communication).

On the indirect-overhead side, two main issues have been addressed, namely data locality and CPU sharing. In the simulation-kernel instances, the output is produced on a contiguous buffer (i.e., the private shared-memory segments), avoiding costly cache-invalidation secondary effects during the generation of output on whichever stream. The segment is used in a cache-aligned fashion, avoiding at the same time false cache sharing effects. The daemon, on the other hand, handles the output materialization relying on an efficient modified version of a calendar queue [17], which is in turn able to minimize the computing time required for sorting the output strings, and it is additionally featured with an autonomic agent which is able to determine the best activation interval for reducing CPU-sharing effects on the actual simulation, and to reduce the delay from the generation of the output to its actual materialization.

5.1.2 The Output Management Architecture

Without any loss of generality, we can say that on a local machine a simulation framework comprises K simulation-kernel instances, scattered across M machines, and each machine m hosts a set of K_m worker threads. Of course, K_m can be different for each m, i.e. the computing power must not be necessarily evenly distributed across the machines, depending on, e.g., the actual number of processing units offered by each of them. Upon simulation startup, on each machine m a separate output-daemon instance is started, and K_m shared-memory segments are created (and shared with each kernel instance). These can be seen

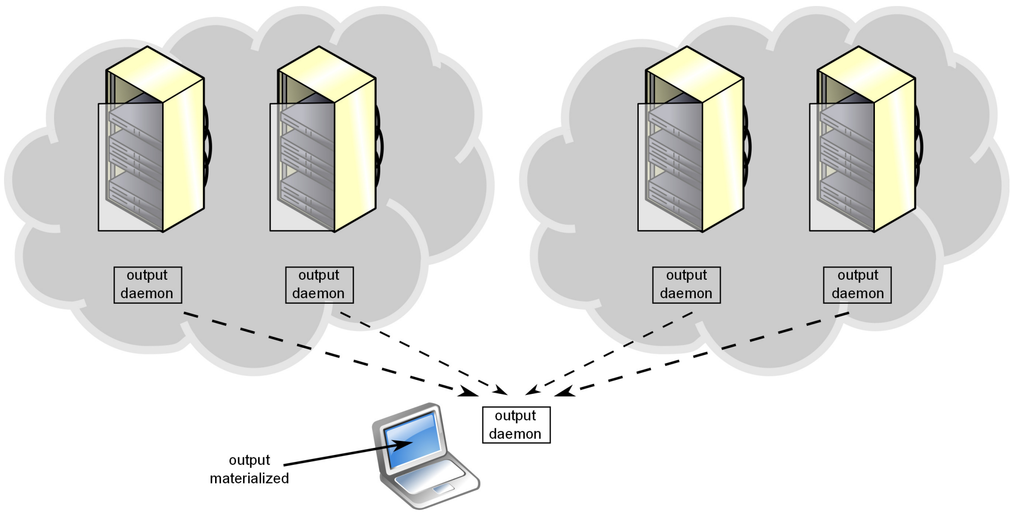


Figure 5.2: Application Scenario of the Output Daemon

as per-kernel private *logical devices*, on which the simulation-kernel instances' worker threads write their LPs' output messages, rather than on the requested stream. In this scenario, considering the whole system, D_M^K logical devices will be installed, and each kernel k hosted on machine m will have is own device D_m^k. This scenario is illustrated in Figure 5.2.

In particular, as mentioned before, upon the invocation of some function of the printf-family (which is redirected to the kernel instance's output subsystem at compile time), the output subsystem stores output-related information in a variable-sized data structure for keeping the information until the associated event gets committed (or rolled back). The structure used for this intermediate buffering is as follows:

```
struct output_msg_t {
    size_t size;
    unsigned int LP;
    double timestamp;
    int fd;
    unsigned int era;
    char buffer[];
};
```

where:

`size` keeps the total size of the entry;

`LP` associates the output generation with a specific LP in the simulation;

`timestamp` stores LVT at which the LP has generated the output;

`fd` is the file descriptor associated with the output stream;

`buffer` is the actual output payload[1].

The `output_msg_t` entry is then written (before returning control to the simulation model) into the per-kernel logical device D_m^k. We note that `timestamp` is not required to be explicitly provided by the simulation model within the output message (namely as a parameter of the `printf()` call). Rather, we assume the corresponding LVT value (as well as the LP identifier) must be provided by the simulation kernel to the output subsystem right before passing control to the LP for actual event processing. In this way, the application modeller is not forced to tag all (or part) of the output messages with timestamping information. She will be anyhow transparently provided with an output-stream content matching (system-wide) the advancement of simulation time.

[1]We note that, if output generation entails processing a format string, this can be easily done by relying on the POSIX `sprintf()` library function.

The logical device can be implemented as a circular buffer, where kernels' output subsystems write new output messages (which are later processed by the daemon in a FIFO order) with the aforementioned ad-hoc header for describing the content of the buffer. The organization of this device is described via the following structure:

```
struct logical_device_t {
   size_t size;
   volatile unsigned long long writing;
   volatile unsigned long long wrote;
   volatile unsigned long long read;
   unsigned char buffer[];
};
```

where:

`size` identifies the (current) size of the circular buffer;

`writing` is an offset pointing the last byte which a worker thread is currently using to deliver a message to the output daemon;

`wrote` is an offset pointing to the last byte in the device already written by the kernel output subsystem;

`read` is an offset identifying the last byte read by the output daemon.

Accessing the Logical Device

If we consider one simulation kernel instance running with only one worker thread, the `logical_device_t`'s fields are updated in isolation (i.e., the field written by the one kernel worker thread is only read by the daemon, and vice versa), then a non-blocking algorithm can be easily developed for managing the logical device. In particular, by checking if `wrote` and `read` store the same value (either in modular or non-modular arithmetic), the daemon and the kernel are able to determine whether the buffer is full or empty, and can immediately access via displacement the correct position on which to perform a read/write operation, without the need for any synchronization primitive. If the buffer is full, then the simulation kernel must wait before finalizing the output generation procedure, unless some resize procedure, like the non-blocking one described in Section 5.1.3, is adopted.

Of course, if the simulation kernel instance has more than one worker thread, there could be a race condition on the update of the `wrote` field. We therefore use the `writing` field, and rely on a fine-grain synchronization primitive, namely the *compare-and-swap operation* (`CAS`), which is provided by most instruction sets currently available. This solution allows us to have as well a non-blocking algorithm, thus providing a good performance when the available worker threads do not try to access the buffer at the same WCT instant, which is an acceptable condition given the (usually) complex nature of events' execution flow. In particular, any worker thread writing an output message on the logical device starts this operation by trying to "reserve" a portion of the buffer. This is achieved by atomically reading the value of the `writing` field[2] into a local (automatic) one. This local copy is then incremented by the size of the output message being written to the logical device. Finally, by relying on `CAS`, we (atomically) try to put back in place the updated value. If the `CAS` fails, then some other worker thread has (concurrently) updated the value of the `writing` field. In case it succeeds, then all the other worker threads will see an updated value of the `writing` field, and thus the memory buffer associated with that amount of bytes is reserved

[2]We emphasize that in many architectures, e.g. the x86/x86_64, reading a primitive type variable (an integer in our case) from memory is intrinsically an atomic operation.

Algorithm 5.3 Accessing the Logical Device in Write Mode

```
procedure Write_Output_Message(msg)
    while true do
        local_writing ← device.writing;
        incr_local_writing ← local_writing + sizeof(msg)
        if CAS(&device.writing, local_writing, incr_local_writing) then
            break;
        end if
    end while
    Copy msg to device.buffer[local_writing]        ▷ Executed in isolation
    while device.wrote $\neq$ local_writing do
        no-op;
    end while
    device.wrote ← incr_local_write;                ▷ Executed in isolation
end procedure
```

for usage by the current worker thread. Then, the copy operation is executed, but the output daemon is not yet informed of this new message in the device.

Then, the worker thread reads the `wrote` field, to check whether its value is equal to the local copy of the `writing` field that it had read at the beginning of the delivery operation. If it is not, then some other worker thread is already performing its copy operation. If it is, all the other worker threads that reserved a buffer on the device before the current worker thread have completed their copy operation. In the latter case, the current worker thread simply copies the value of its local copy of the `writing` variable into the `wrote` field. The whole algorithm to access the logical device in write mode is reported in Algorithm 5.3. Although this latter part of the write access on the logical device actually serializes the updates of the `wrote` field, we note that the delay for this update by the first worker thread accessing the device corresponds to the time required for copying the `output_msg_t` data structure. Since the copy is executed on a contiguous buffer by relying on fast string-copy assembly operations, the expected cost for this operation is minimal, thus causing a very reduced delay when the worker threads have to actually wait for each other.

On a periodic basis, the output daemon wakes up and checks whether some output message has been produced by some locally-hosted simulation-kernel worker thread on its private logical device (again, this can be done by comparing the values of `read` and `wrote`). In case a new output message is present, the output daemon creates a local copy of it (i.e., in its own address space), updates the `read` value, and inserts it in a specifically-modified version of a calendar queue [17], which guarantees the correct ordering of the output messages by all the output kernels. We also note that, in case the two counters match, but the actual content of a new message has already been inserted within the device (which does not impose atomicity with the update of the `wrote` counter), the daemon will simply experience a false-negative on the presence of new messages, which will be resolved at subsequent iteration steps. The final algorithm which performs a read operation from the logical device must account for some additional performance issues which will be described in Section 5.1.3, and therefore we delay the final illustration of the algorithm to that section.

Overall, with the above approach, no spin-locks or other types of atomic operations (which are known to exhibit non-minimal direct and indirect costs, possibly impacting the execution speed at the side of the simulation engine) are used at all. This is done by trading-off with the actual delay for the delivery of the message at the daemon.

Supporting the Rollback Operation

In order for the output daemon to correctly support the execution of rollback operations, the traditional implementation of the calendar queue has been augmented with one additional operation, namely `delete`, which allows to remove elements falling in a given $[from, to]$ timestamp range. For efficiency reasons, upon the invocation of this operation, the buckets associated with $from$ and to are computed, and then a linear search for removing elements is performed, resembling the original direct search described in [17]. Of course, the delete operation removes only the elements which are associated with a particular LP, as specified in the `output_msg_t` structure. At the same time, we note that, depending on the actual simulation execution, scanning all the buckets in the $[from, to]$ range can be a relatively costly operation which can affect the output materialization performance. To this end, the `output_msg_t` structure keeps track of the `era` field, namely a monotonic counter which is (atomically) updated by every worker thread upon the execution of a rollback operation, on a per-LP basis[3]. This can be regarded as a compressed information identifying every output message related to a given era comprised in between two rollback operations. The calendar-queue bucket array is therefore augmented, keeping for every bucket a bloom filter [16] which is used to know whether output messages from a given era are present in the bucket. In this way, before starting to scan the list associated with any bucket in the $[from, to]$ interval, a check on the bloom filter is performed, to determine whether that particular bucket can be skipped.

To enable the execution of a rollback operation, and to enforce a timely materialization of output messages on the related streams, two *control messages* are placed by kernel k (namely by the corresponding kernel's output subsystem) on device D^k_m, namely `ROLLBACK` and `COMMIT`. The former is a control message with a payload structured as $\langle from, to, LP, era \rangle$, which directly triggers the

[3]To execute this atomic increment, again the `CAS` can be used. In our implementation we have relied on the x86-specific `lock; incl mem/64` atomic assembly instruction.

aforementioned `delete` operation on the calendar queue. The latter, notifies the output daemon on which portion of the per-kernel-generated output can be considered as committed[4]. Whenever a `COMMIT` control message is found on a device, the associated commitment timestamp is stored into a separate array, which keeps track of the last commitment time for each locally-hosted simulation kernel instance k. Periodically, the output daemon finds the minimum commitment time $T_{C_{min}}$ among the ones notified by the kernel worker threads, and invokes a `flush` operation on the calendar queue, which iteratively dequeues elements from it until an element associated with a timestamp $T > T_{C_{min}}$ is found. Every output message dequeued during this flush operation is materialized on the associated stream, described by the `fd` field.

We note that these operations can be non-negligible in terms of CPU usage by the output daemon. In order to produce a minimal impact on the simulation's overall performance, the daemon does not process all the available messages (either output or control) from $D_m^k \forall k$ before returning to sleep. On the other hand, after having processed any message, it checks whether its execution has lasted more than a specific threshold value and, in the positive case, it returns to sleep. Of course, the overall performance can be affected by the sleep and working time. As it will be discussed later in Section 5.1.3, an ad-hoc solution can be adopted to autonomically self-tune these parameters for maximizing the simulation's performance, while ensuring a timely materialization of committed output messages.

[4]In fact, the `COMMIT` control message is generated immediately after the completion of the GVT reduction operation.

Working in a Distributed Environment

In case the simulation is executed on a distributed architecture (either a cluster, a desktop grid, or on the cloud), the daemon-based approach is able to communicate only with the kernel instance which is locally hosted on the same machine. In order to collect output messages from all the nodes in the system, the output daemon can be configured to act as a *forwarder*, i.e. whenever a set of output messages is detected to be committed, the messages are forwarded on the network to an additional (remote) instance of the daemon which acts as a *collector*. This additional instance, which can be even geographically far from the actual simulation architecture, receives all the committed messages from every daemon running in the simulation framework, and inserts them into an additional calendar queue. Whenever `COMMIT` messages are forwarded, the collector daemon relies on the same aforementioned logic for materializing the output messages on the associated streams. We note that this approach is effective in two ways:

1) over the network we transfer only committed output, thus we early process rollback operations, and avoid to pay costly network delays for exchanging output messages which might not be committed, enabling for a considerable scalability even at a geographical scale;

2) we can finalize the materialization of the output messages on a remote (possibly not dedicated to simulation) machine, allowing both an enhanced simulation performance (in fact, materializing output on a stream can be a non-minimal-cost operation) and the possibility to easily monitor the execution of the simulation by the user on a separate machine, without suffering a considerable delay.

Working with Conservative Synchronization Systems

The output daemon can be flagged to work in *autocommit* mode. Using this facility, the daemon (running in the aforementioned forwarding mode) can be connected to a conservative distributed simulation framework, where the roll-back issue is not present, but nevertheless for producing consistent output a timestamp-based ordering must be performed on the output messages produced by the various nodes in the system. Therefore, in this scenario, the facilities provided by our proposal would never trigger a `delete` operation, but upon receiving a message, its timestamp is considered as the last committed timestamp for the specific LP. By simply performing a local reduction across the current LVT of the LPs (which, as said is communicated by the simulation engine to the kernel's output subsystem while the simulation proceeds), the set of (timestamp-ordered) output messages to be flushed is identified as the set of those messages with timestamp less than the reduction value, which are therefore immediately flushed (i.e., forwarded) for output materialization. We note that this approach to execute the reduction on the timestamps associated with output messages is similar in spirit with the definition of GVT as for Definition 2.1.

Internal API Overview

To give the final picture, we list the core set of API functions that are provided by the kernel's output subsystem, which can be used for integration with differentiated simulation kernel layers:

`commit_time(GVT):` This function is exposed by the output subsystem in order to allow the GVT reduction subsystem to notify the output daemon of a newly-computed GVT value. This information is used by the subsystem to generate a `COMMIT` control message to notify the output daemon.

`set_LVT(LP, timestamp):` Using this function, the simulation kernel's scheduler can notify the output subsystem about the identity of the dispatched LP, and the timestamp of the dispatched event (hence the logical time to be associated with any output message produced as a result of processing this event).

`rollback(from, to, LP):` Using this function, the simulation kernel's scheduler can notify the output subsystem about the reception of a straggler message or an anti-message. Using this information, the output subsystem can generate a `ROLLBACK` message for the output daemon, and instantiate a new era.

`out_msg(LP, stamp, msg, stream):` This API is used by the simulation kernel to transfer a particular output message destined to a specified stream to the output subsystem, which will in turn write it on the kernel's logical device D_m^k.

`autocommit(flag):` If `flag` is true, every output message received is considered as non-rollbackable, in order to support the integration with a conservative simulation engine. If `flag` is true, the daemon does not wait for `COMMIT` messages before committing output messages. The commitment is triggered by the aforementioned internal logic.

We recall that, in order to provide the application-level developer with complete transparency, calls to output generation library functions must be properly wrapped, in order to correctly redirect them to `out_msg()`. This can be easily done, as already described, by relying on linking-time facilities.

As a final note, since the proposed output message data structure identifies a particular stream using its file descriptor, the kernel instance's output

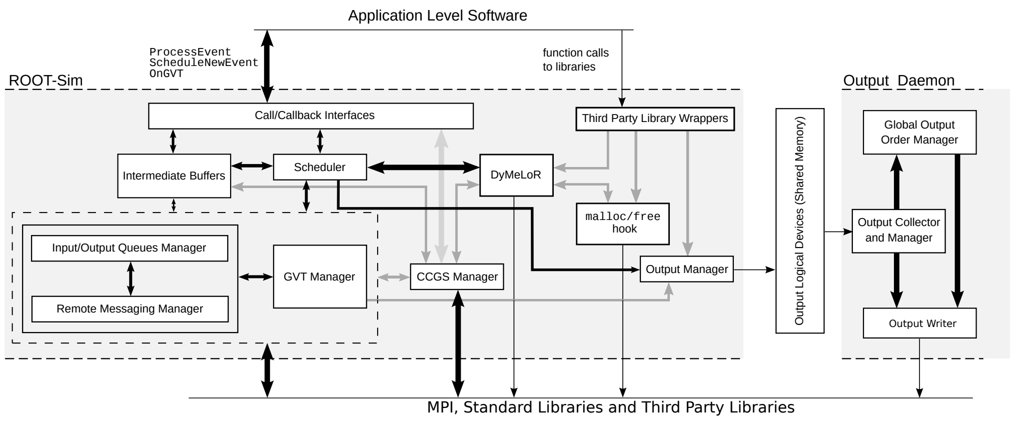


Figure 5.3: ROOT-Sim and the Output Daemon Architecture

subsystem intercepts calls to functions which actually open/close streams (e.g., `fopen()`-family calls) and produces on the logical device additional control messages (namely, `OPEN` and `CLOSE`) which tell the output daemon to process these operations. In this way, the daemon is the only user-space process controlling the actual output streams, for materializing committed output messages on them.

The interconnection with the simulation kernel, particularly the ROOT-Sim kernel we are exploiting in this study, is depicted in Figure 5.3.

5.1.3 Optimizations

In case of an output-intensive simulation model (either because a large number of events involves the generation of some output destined to some stream, or because the output messages are considerably large) the logical device might become the bottleneck of the system. In particular, since we want to ensure *exactly-once* materialization of the output messages, if the circular buffer which is used to implement the logical device gets filled, the kernel's output subsystem trying to write on it must wait for the output daemon to free some space for buffering the output.

Since this is a non-affordable cost, we have added to the `logical_device_t` data structure an additional flag, namely `subst_id`, which is used by the output subsystem whenever the current circular buffer is full. In this case, the output subsystem uses a `CAS` to flag the `subst_id` field to tell other worker threads that a new logical device is currently being created. Therefore, if any other worker thread finds this flag already set, it simply wait for this field to be updated again. The worker thread performing the new logical device initialization allocates a new shared memory segment (of doubled size with respect to the old, full one), stores its id in this field, detaches from the old segment, and starts buffering output messages on this new device. Whenever the output daemon finds a logical device to be empty, it checks whether the `subst_id` field is set to a valid value. In the positive case, it attaches to the new logical device, releases the old one, and starts processing messages from the new one.

If on the one hand this approach can reduce the time spent during the execution of a simulation event inside the kernel's management subsystems, on the other it can affect memory usage if the simulation model's activity is highly output-bound. Although this can be regarded as a secondary problem, considering the large amount of memory available on modern architectures, optimizing the actual tradeoff between CPU usage by the output daemon and memory consumption by the logical device can provide benefits in terms of output materialization delay and problem feasibility.

We therefore propose an additional optimization, which tackles the time spent by the output daemon processing messages from the devices and sleeping, and allows the output manager to autonomically self tune its activation phase. In particular, as for output materialization delay, given that output messages are flushed on the associated streams after the GVT computation (which is, traditionally, a periodic operation), the best scenario arises when the kernels' output subsystems try to write the `COMMIT` message on their devices and find them empty. This situation happens whenever the total execution time of the output daemon is such that every message placed in the device in between two `COMMIT` messages has been processed. We therefore measure the processing time of each message (namely, output message's t_o, a `COMMIT` message's t_c, and a `ROLLBACK` message's t_r) and continuously update their mean values $\bar{t}_o$, $\bar{t}_c$, and $\bar{t}_r$ according to an exponential average. Additionally, we rely on three counters, namely $\bar{c}_o$ $\bar{c}_c$, and $\bar{c}_r$ which describe the (exponential) average number of messages placed by simulation kernel worker threads in a GVT phase. By relying on these values, upon the reception of a `COMMIT` control message, the output daemon computes the expected total execution time which is needed for emptying the logical devices during the next phase:

$$\mathbb{E}(T) = \sum_{x \in \{o,c,r\}} \bar{t}_x \cdot \bar{c}_x \tag{5.1}$$

If the value of Equation (5.1) is lower than a compile-time specified threshold, it is then fragmented into several slots (resembling operating systems' time slices) and the sleep time is set accordingly in order to identify an execution phase which is equal to the GVT phase. The compile-time threshold ensures that, in case an application is extremely output-bound (at least in some phase of its execution), then the output daemon will not significantly affect the overall simulation performance, while trying to minimize the output materialization delay. We note that, by relying on this approach, the output daemon is able to capture variations in the actual output generation dynamics by the application-level software, thus adapting to execution phases which exhibit a higher/lower output generation rate. We are now able to present the final pseudo-code for the read operation on the logical device, which is shown in Algorithm 5.4.

The last optimization concerns timely processing of output-messages rollback, which can prevent the output daemon to process wrong data before noticing that it must be undone. In particular, upon initialization of the per-kernel logical device, each simulation-kernel instance installs a second channel for the delivery of high-priority control messages (namely the `ROLLBACK` ones) which are used to early inform the daemon that some information that it is about to process might be already uncommittable. Whenever the daemon notices that a `ROLLBACK` message is present in the high-priority channel, it starts scanning the associated logical device and marks as *don't care* every output message which should be rolled back. The circular buffer is scanned (without updating the `read` flag) until the `ROLLBACK` control message corresponding to the one found in the high-priority channel is found. When this operation is completed, the normal behaviour is restored, with the exception that whenever a don't-care output message is found in the circular buffer, it gets simply discarded. We note that this rollback optimization avoids enqueueing and dequeueing the output messages which can be early detected as uncommittable, reducing the cost of the operations on the calendar queue[5]. If some events involved in the rollback operation were already in the calendar queue, when the processing of the logical device (which, we recall, stores messages emitted by the associated simulation kernel in FIFO order on per-LP basis) reaches the `ROLLBACK` control message, the `delete` operation described in Section 5.1.2 is executed, which removes the already inserted output messages.

[5]This situation might result in a calendar queue's resize, which is a very costly operation.

5.2 Experimental Evaluation

We have integrated the proposed I/O management architecture within ROOT-Sim. The output management subsystem has been integrated respecting the API described in Section 5.1.2, while the output daemon has been realized as a separate process written in ANSI-C. Upon the ROOT-Sim startup, if requested, the output daemon is launched and the output subsystem installs the logical devices (one for every simulation-kernel instance) .

In order to evaluate different aspects of the proposed architecture, we have conducted experiments on a family of configurations of *Personal Communications Service* (PCS), as described in Section 2.5.3. We have performed a set of experiments where each cell sustains the same workload of incoming calls, whose inter-arrival time is exponentially distributed, and whose average duration is set to 2 minutes. The expected rate for call inter-arrival has been set to achieve channel utilization factor, as described by Equation (2.6), on the order of 30%, while the residence time of an active device within a cell has a mean value of 5 minutes and follows the exponential distribution. For the above scenario, we have run experiments with 1024 wireless cells, modelled as hexagons covering a square region, each one managing 1000 wireless channels. These have been evenly distributed across 32 simulation-kernel worker threads running on the 32-core underlying machine.

For measuring the overall performance of the simulation runs, we have relied on the measurement of cumulated committed events over wall clock time advancement, i.e. a measure of how many events get committed while the simulation's execution is carried on.

To capture different aspects of the impact on execution dynamics by the output-management subsystem and the output daemon, we have modified the

PCS benchmark in order to provide statistics regarding the occurrence of the hand-off event on the terminal console. In particular, with a certain frequency f, the execution of the hand-off event entails the generation of an output string which tells the total amount of per-cell hand-off events so far, and the average duration of a handed-off call. We have explicitly varied the value of f so that, among all the events (not only hand-off events) executed in the simulation run, in between 1% and 35% of them involved some output generation (which corresponds, in our configuration, to a total number of generated string in between 5 millions and 35 millions per simulation run).

We note that, for the above deploy/parametrization, the runs exhibited an efficiency on the order of 80%. Hence the experimentation has been carried out when considering well-behaving optimistic runs (namely not affected by thrashing, which would lead the experimentation to be non-reliable), which however show an amount of rollback that is expected to provide a good test case for all the functionalities (e.g., output-message discarding functionalities) offered by our output management subsystem. Further, running the above model on top of ROOT-Sim (on the same multi-core machine used in this experimental study) has been already shown to give rise to super-linear speedup values (see the experimental data provided in [172]). Hence our experimentation is carried out via competitive parallel runs.

We have compared this execution with two different scenarios, one relying on the traditional ROOT-Sim framework (i.e., without the output subsystem and without the output daemon), and one with the kernel's output subsystem working within the simulation kernel, but without any daemon listening on the other side of the logical device (in this case the device has been configured like a typical `/dev/null` device file, by simply discarding the incoming messages). We consider the former scenario as a good baseline situation, for assessing the overall overhead introduced by the output management architecture, while the latter allows us to evaluate what is the actual impact of having a separate process (the daemon) running in time-sharing on a dedicated simulation environment, for supporting the execution of additional housekeeping tasks. Further, we have also considered runs based on a modified version of ROOT-Sim where the output strings produced by the execution of an event are logged within the event queue (as a list associated with the event-buffer), and are then flushed when fossil collecting the event after GVT computation. This approach has resemblances with the solution provided in [75]. We note however that this approach does not provide global ordering of the output across the whole set of LPs.

By the results, shown in Figures 5.4–5.7 (obtained as the average over 10 runs done with different pseudo-random seeds), we note that the execution with the device configured like a `/dev/null` device file exhibits a reduced overhead with respect to the baseline configuration, showing that the operations internal to the kernel's output subsystem (and the interaction between the simulation engine and the stub) do not impact significantly on the overall simulation performance. When the output daemon is running, on the other hand, the simulation throughput decreases when the application-level software exhibits more output-bound behaviour. This is reasonable, considering that the CPU time required by the daemon for a timely processing of the messages placed into the logical device gets increased. However, the worst case for the overhead is on the order of 22%. Further, the overhead scales well with respect to the increase of the frequency of output production (in fact the overhead is quite similar for the cases where f is set to 12% and 35%). On the other hand, for very reduced output-message frequency (f set to 1% or 7%), as typical of when the application is configured for primarily matching performance requirements via audit reduction, the overhead introduced by our output-management architecture is quite bounded (namely between 2% and 11%). It is interesting to notice that our proposal shows a better performance than the scenario with no output architecture activated, but with I/O calls performed while processing the events. This phenomenon is due to the fact that the `stdio` library is not optimized for integration with high performance computing, while our architecture has been oriented exactly to this scenario. Further, standard I/O calls during event processing would even give rise to non-consistent output, due to the fact that output materialization associated with rolled back events is finalized as well (in the case of optimistic synchronization), or should require an additional post processing to correctly order the output (in case of the conservative synchronization), thus requiring additional time for the end user to be able to perform its audit activities. Similar considerations can be made when considering I/O management via logging of the strings with the processed events and flush operations after GVT computation. Particularly, while this approach introduces negligible overhead with bounded frequency of output production, the overhead is significantly increased for higher frequency of output message generation.

Another aspect which requires attention regards the overhead induced by the evaluation of parameters within a format string being passed to the output subsystem. In Figure 5.8, we present three different curves (whose samples are again computed as the average over 10 runs), one entailing the evaluation of a float, one of an integer, and a case where no parameters should be evaluated at all. We recall that, in our proposal, parameters are evaluated immediately within the kernel's output subsystem (directly called by the application-level code through library call redirection) via the POSIX-compliant `sprintf()` li-

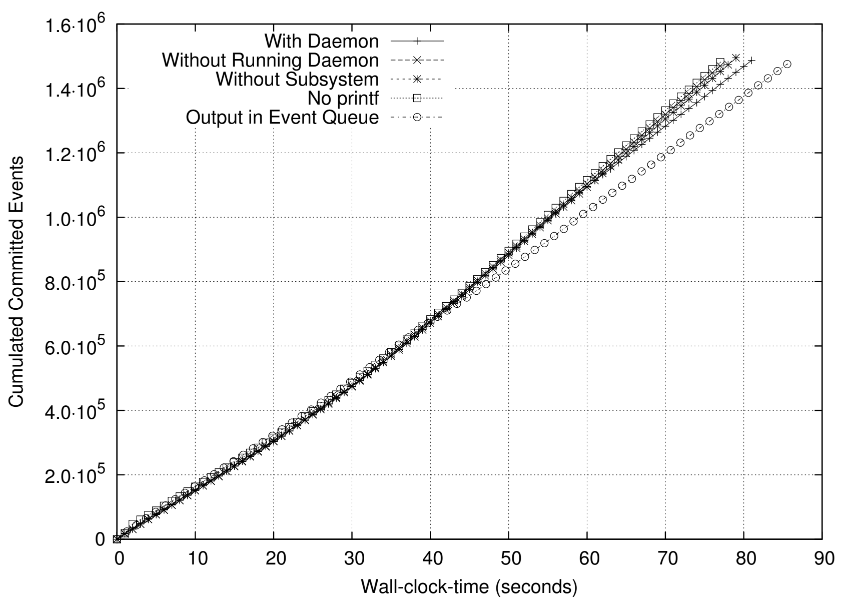


Figure 5.4: Throughput for $f = 1\%$

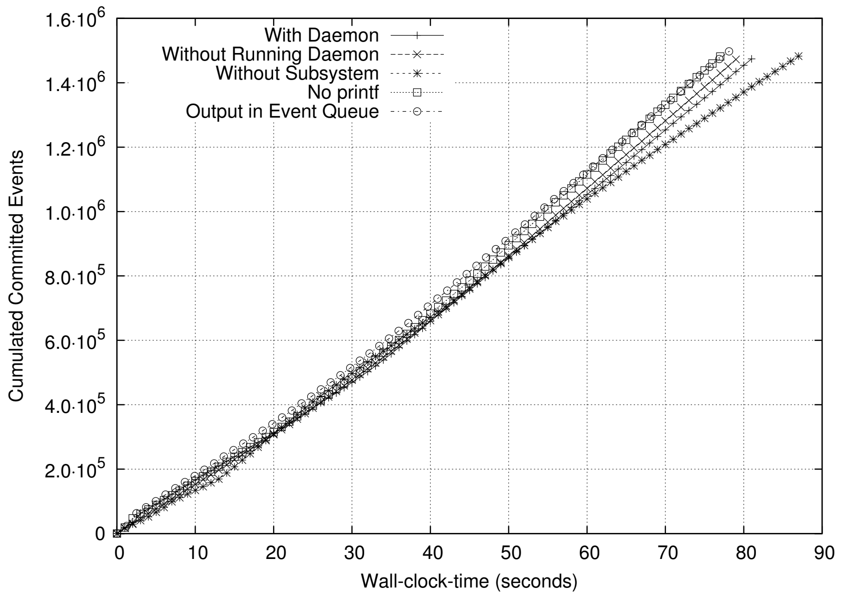


Figure 5.5: Throughput for $f = 7\%$

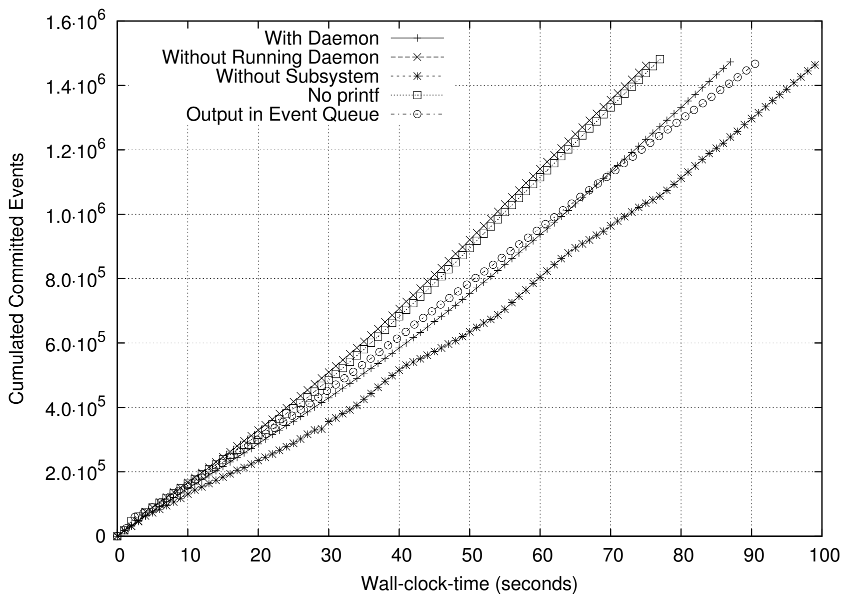


Figure 5.6: Throughput for $f = 12\%$

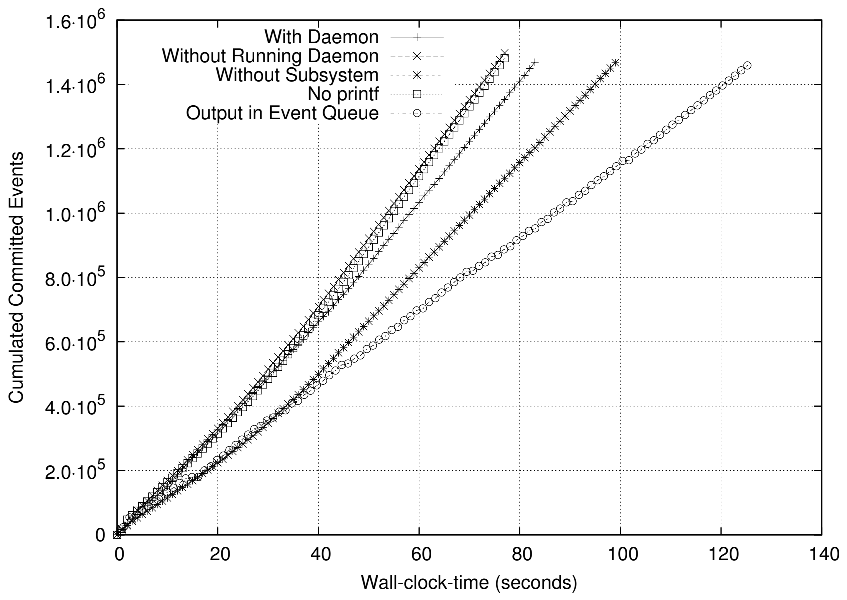


Figure 5.7: Throughput for $f = 35\%$

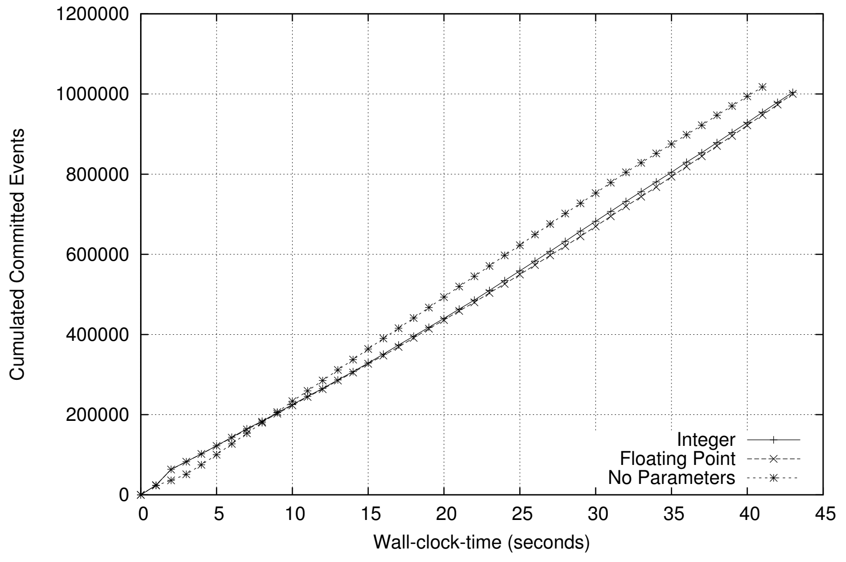


Figure 5.8: Throughput with Different Data Types

brary function. By the results, we can see that the overall execution time is not significantly affected by the presence of additional (more complex) parameters to be evaluated, showing the effectiveness of our proposal even in the case of relatively complex outcome-messages from a simulation model.

To show how the CPU-usage/memory tradeoff is addressed by the output daemon, in Figure 5.9–5.12 we report the total amount of shared memory allocated for the logical I/O-devices during the execution of the simulation (these plots refer to one of the 10 runs, where similar behaviour is anyway observed). As in the case of the throughput discussed earlier, we note that (as expected) the amount of required shared memory is increased when the application exhibits a more output-bound behaviour. However, the reached bound (on the order of 16MB) represents a relatively reduced absolute value—especially when considering that optimistic simulation is known to be memory consuming on the

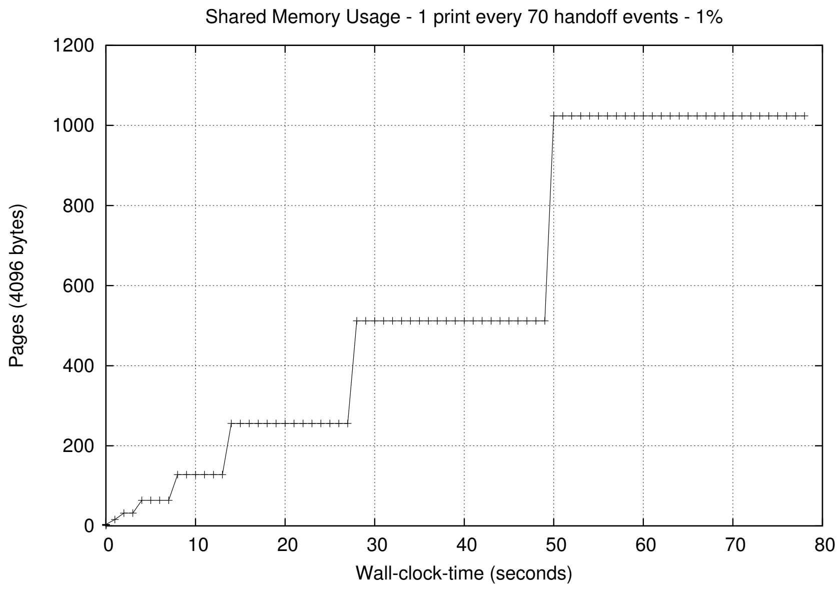


Figure 5.9: Shared Memory Size for $f = 1\%$

side of the engine, and recalling that in the worst-case scenario 35 millions of strings are generated. Further, being this memory virtualized by the underlying operating system, and thanks to the fact that the shared memory segments implementing the logical device are used according to the circular rule (not in scattered mode), we may expect a reduced impact on the actual locality while the frequency of output-message generation gets increased.

In Figures 5.13–5.16 we present plots which shows what is the actual output–materialization delay exhibited by the output daemon (again for one of the ten runs, which is anyhow representative of what observed in the different runs). Both the x and the y axis represent wall clock time. To the x axis we associate the WCT instant t_G at which a specific (later committed) output message was generated, while to the y axis we associate the WCT instant t_M at which the same output message was materialized. For the sake of clarity, we show a 45-

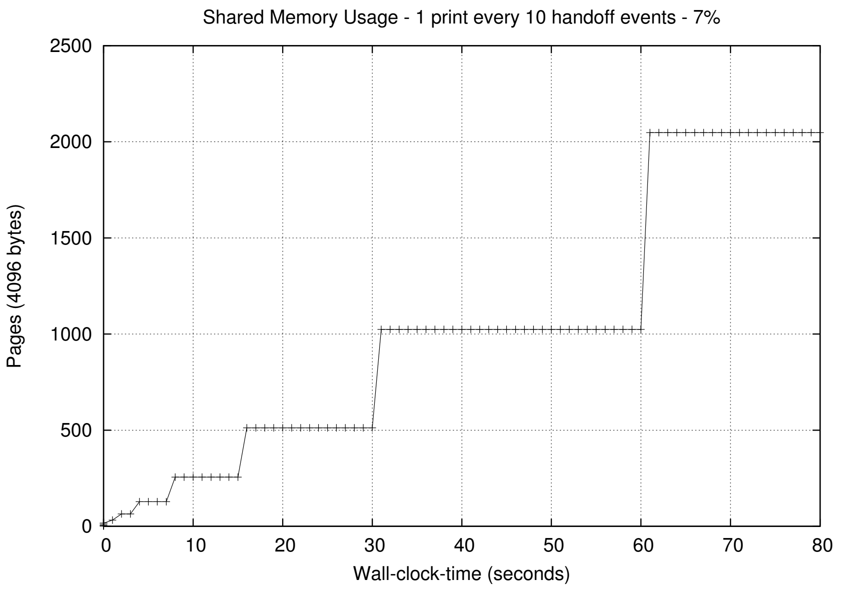


Figure 5.10: Shared Memory Size for $f = 7\%$

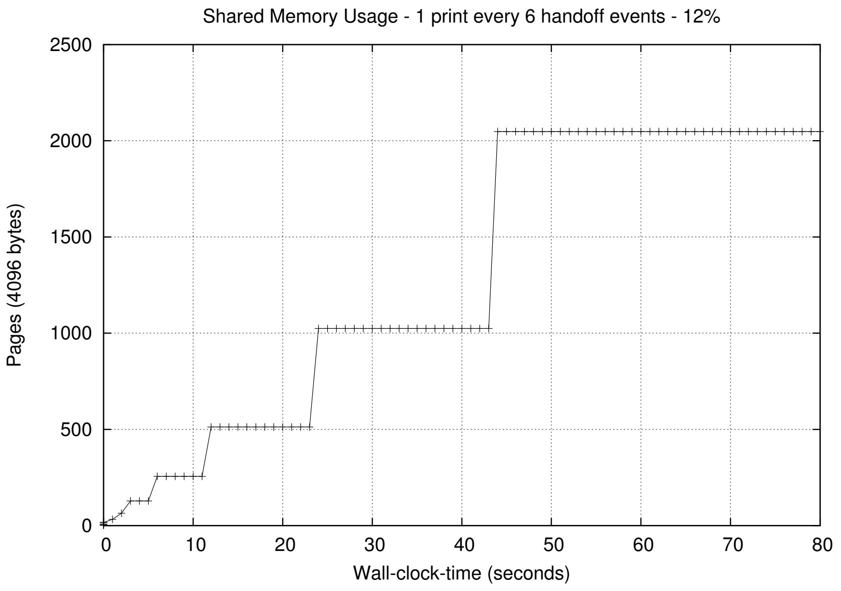


Figure 5.11: Shared Memory Size for $f = 12\%$

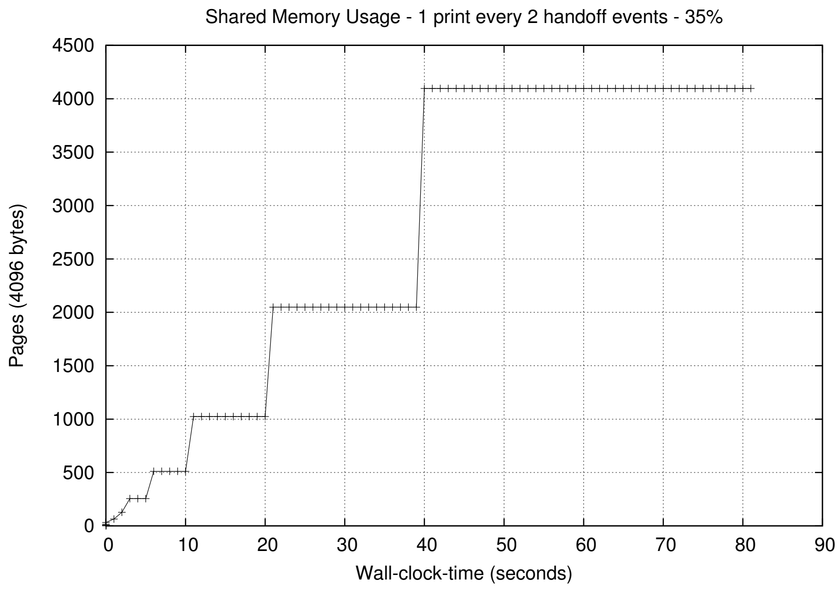


Figure 5.12: Shared Memory Size for $f = 35\%$

degree curve which represent a (theoretical) instantaneous materialization time, i.e. a situation where there is no actual delay between the generation and materialization. In this plot, the steeper the slope, the higher is the materialization delay induced by the operations by the output daemon. It is interesting to note, by the plots, that the output materialization advances in steps. This is related to the fact that the commitment operation of output messages can be started only after a GVT calculation, which is a periodic operation. It is interesting to note that in Figures 5.13 and 5.15, the autonomic self-tuning subsystem for CPU/memory tradeoff optimization is able to capture the best configuration to minimize the materialization delay. In fact, the slope of the curve, during the simulation execution, tends to get gentler. The case in Figure 5.14 is quite different, as the daemon tries to minimize the output materialization delay, but then the CPU threshold is hit, and the daemon is not able to increase its computing power usage, in order not to significantly affect simulation performance, and therefore the curve diverges. The scenario in Figure 5.16 shows that, considering the large amount of output messages, the calendar queue becomes the bottleneck of the system (due to an upper bound on the number of buckets, as described in [17]), and the rollbacks which are encountered during the simulation determine an amount of operations on the calendar queue to delete messages which were already stored. However, we note that for values of f from 1% to 7% (which, as said, would represent a case of orientation of the application layer to performance due to reduced audit on model execution), we get that upon run termination the output stream has been already materialized at the 75% or, at least, the 35%, which would enable pipelined treatment of the output data while the run is still in progress. The careful reader might notice that with a given generation WCT instant t_G, more materialization WCT instants t_M are associated. This is related to the fact that these plots present system-wide materialization delays, where different kernel instances at the same WCT value might generate output messages from LPs running at different LVT values. This skew is therefore reflected in the commitment (wall-clock) time at which a set of messages can be safely materialized on the associated output stream.

To assess the effects of the autonomic self-tuning mechanism for daemon activation, described in Section 5.1.3, we present in Figure 5.17 a scenario where the parameter f (which describes the frequency of statistics generation on standard output) is not constant over the simulation run, rather varies in the range $[1\%, 30\%]$ in an interleaved fashion. In particular, f is set to 1% at the beginning of the simulation, and is then incremented until it reaches the value of 30%, and then again decreased. The plots show that the autonomic self-tuning system is able to cope well with the dynamics variations. In fact there is not

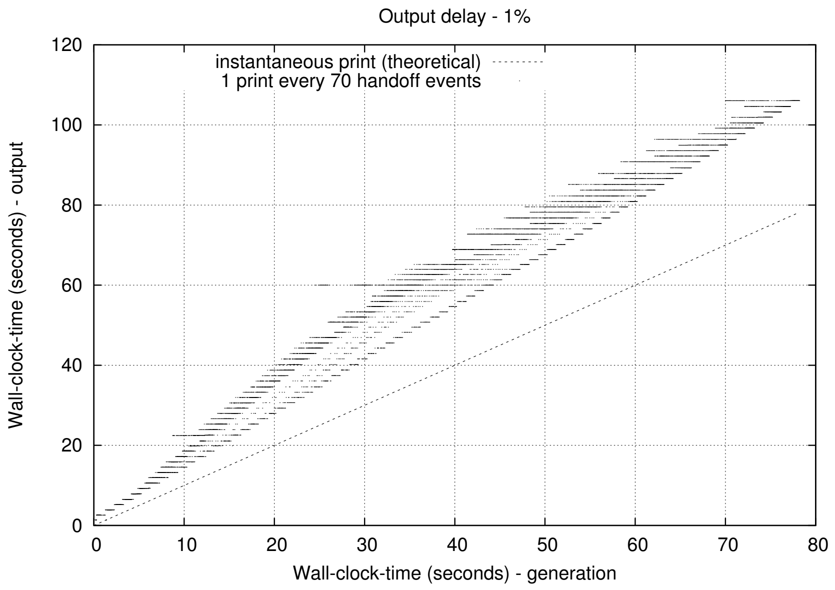


Figure 5.13: Generation/Materialization Delay for $f = 1\%$

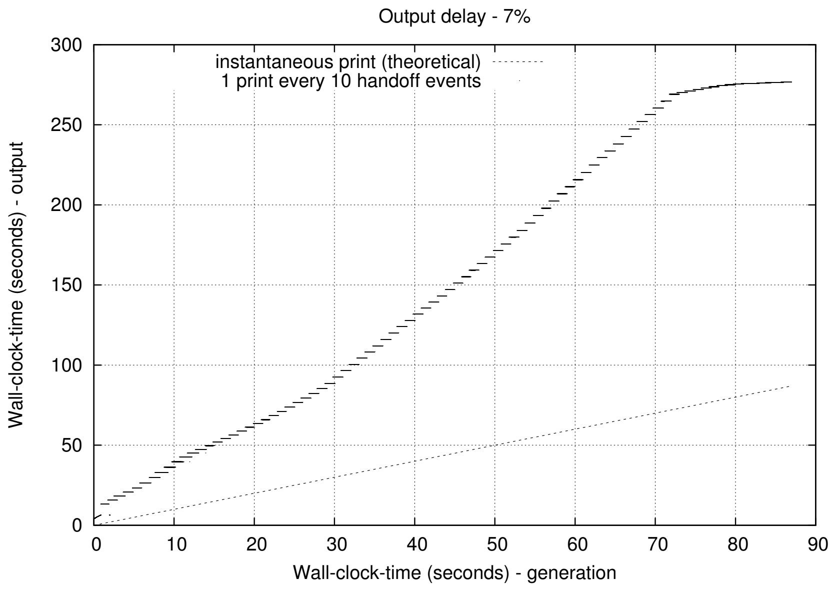


Figure 5.14: Generation/Materialization Delay for $f = 7\%$

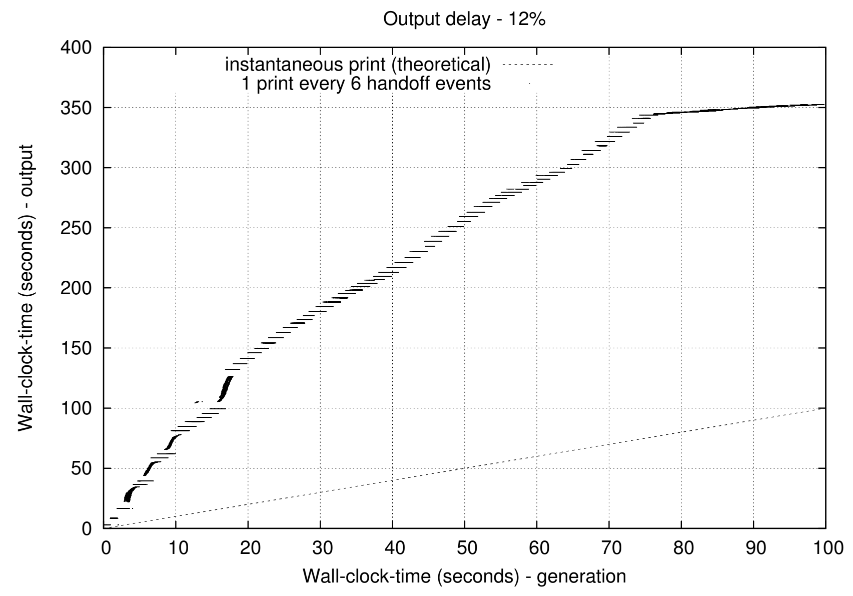


Figure 5.15: Generation/Materialization Delay for $f = 12\%$

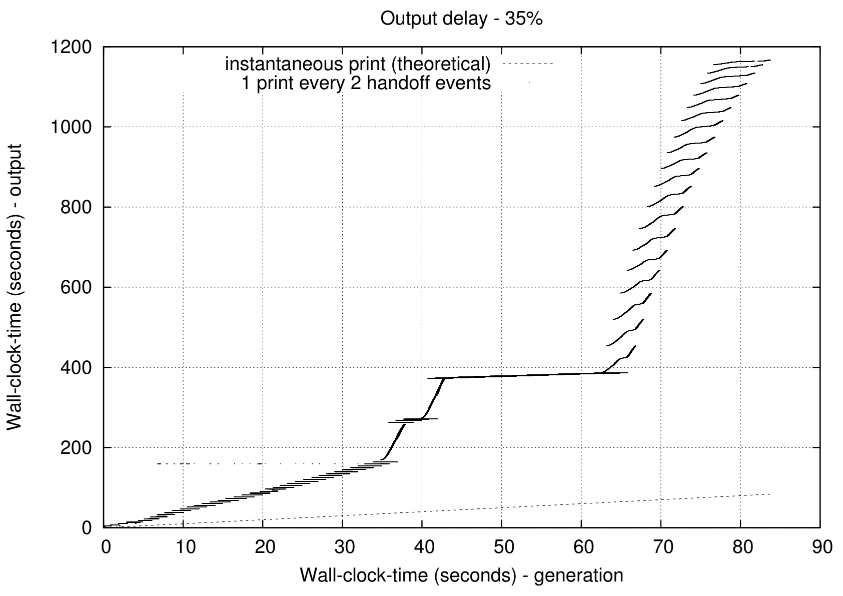


Figure 5.16: Generation/Materialization Delay for $f = 35\%$

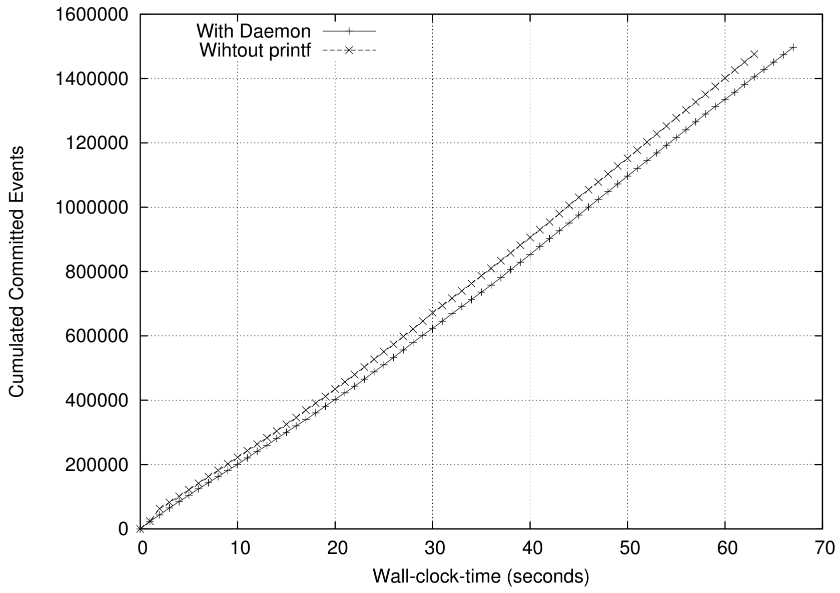


Figure 5.17: Frequency Variation

any significant skew in the simulation throughput, despite the output daemon continuously requests for more or less computing power, depending on the actual load phase on the output architecture.

Algorithm 5.4 Accessing the Logical Device in Read Mode

```
sleep_time ← initial_sleep_time;
awake_time ← initial_awake_time;
exec_time ← 0;
$\bar{t}_o$, $\bar{t}_c$, $\bar{t}_r$ ← 0;
$c_o$, $c_c$, $c_r$ ← 0;

procedure READ_OUTPUT_MESSAGES
    while true do
        if executing_time $\geq$ awake_time then
            SLEEP(sleep_time)
            Recompute Equation (5.1)
            Recompute sleep_time and awake_time
        end if
        timer ← CURRENT_MACHINE_TIME
        if $\neg$empty then
            msg ← next message in device;
            $x$ ← msg.type;
            Execute the operation associated with msg
            $c_x \leftarrow c_x + 1$
            timer ← CURRENT_MACHINE_TIME - timer
            $\bar{t}_x = \alpha\cdot$ timer $+ (1 - \alpha) \cdot \bar{t}_x$
        else
            while device.subst_id == 0 do
                no-op;
            end while
            attach to new subst_id device;
        end if
        exec_time ← exec_time + CURRENT_MACHINE_TIME - timer
    end while
end procedure
```

Chapter 6

Managing Global Variables

Nous sommes condamnés à tout savoir. Nous sommes condamnés au partage.
(We are condemned to know everything. We are condemned to share.)
— Abbé Pierre, Les nouveaux pauvres, 1984

As mentioned in Chapter 1, in classical PDES LPs' states are assumed to be disjoint. Hence, according to this definition, each LP is only allowed to modify its private state variables upon processing new events, and the interactions (namely inter-dependencies) across LPs are only allowed to be instantiated via cross-LP scheduling of simulation events. This is reflected in Equation (2.5), which tells us that there is not the possibility to have any portion of the global simulation state S which does not belong to any S_i.

In practice, this means that when implementing a simulation model, the programmer is not allowed to rely on global variables. Yet, having different LPs sharing (at least a portion of) the state of the simulation model may result in a more flexible programming paradigm, whose relevance has been fully recognized as a crucial issue in the development of parallel simulation applications [49, 107].

In this Chapter we start addressing the issue of transparently and efficiently supporting shared-state in optimistic simulation systems run on top of shared-memory/multi-core machines, a process which will be finished in Chapter 7. We will enable the simulation model writer to access, while executing any simulation event, both the private state of the LP and a global portion of the state, whose instance is represented by the value of global variables admitted within the application-level code. This will allow us to rewrite Equation (2.5) as:

$$S_i \cap S_j = \emptyset, \forall i \neq j \tag{6.1}$$

meaning that, after the further step that will be discussed into this Chapter, the programmer will be allowed to rely on the heap for allocating/deallocating memory chunks belonging to the private state of each LP, as already supported via the approach discussed in Chapter 4, while also being able to rely on global variables for the shared portion of the state.

In order to provide efficient support for the management of shared-state variables, in terms of both forward and backward computation, our proposal relies on an application-transparent multi-version scheme based on non-blocking access/update operations. This allows improving the level of parallelism when the shared-state is accessed by multiple LPs being concurrently run by different worker threads.

6.1 Shared-State Management Architecture

Being our approach targeted at multi-thread simulation kernels deployed on top of multi-core machines, in our Shared-State Management Subsystem (SSMS) we have explicitly decided to rely on a large preallocated memory segment (which is directly accessible by all worker threads) for keeping the current state of global variables. This allows a fast access to the data structures, although requiring some sort of synchronization between instances in order to ensure correctness. To relieve the synchronization burden, we have again decided to implement data structures' accesses as non-blocking algorithms [60], which are expected to ensure better performance than locking ones when accesses are statistically spread across the various portions of the data. To ease the application-level programmer, we have addressed transparency via software instrumentation, so that no additional API or code construct should be used to notify SSMS of accesses to global variables.

To enhance even more execution's performance, SSMS provides a runtime mechanism for detecting actual access patterns. In fact, a set of global variables can be logically intended as a single entity (towards the accesses LPs perform on them). In case such an access behaviour is detected, SSMS clusters the global variables and starts handling them via a single version list, therefore reducing the overhead associated with data structures' management.

6.1.1 Read/Write Detection

In order to provide complete transparency to the application-level programmer, accesses in read/write mode to global variables must be explicitly intercepted. To this end, we rely on instrumentation techniques aimed at modifying the actual instructions executed by software executables, without altering their actual semantics. By relying on Hijacker, at compile time the application-level instruction code (i.e., the assembly byte-stream) is modified in order to replace operations loading data to and from memory with actual function calls which are the entry points of our SSMS.

These entry points are associated with the following provided API functions:

`write_global_variable(void *orig_addr, time_type lvt, ...)`, and `long long read_global_variable(void *orig_addr, time_type my_lvt)`. They allow accessing the versions within the version lists for a given variable at a certain LVT.

We have identified two main groups of instructions/code blocks which have to be handled within the application-level assembly code. First, in x86/x86_64 simple load and store operations are identified by `mov` instructions. Whenever Hijacker identifies a `mov` instruction, it is analysed in order to determine whether it is targeting memory as a source or destination operand, and a call to `write_global_variable` or `read_global_variable` is replaced accordingly. When the `mov` instruction involves a load operation from memory, an additional postamble to the function call is placed, in order to have the actual value returned by `read_global_variable` placed into the correct CPU register where the application-level software is expecting the value to be found. Of course, the register used by the `read_global_variable()` function is pushed/popped on stack, not to alter the actual view on the processor state by the application.

Second, the x86/x86_64 instruction set provides more complex instructions which allow an executable to efficiently modify memory areas in-place. As a relevant example, we propose instructions like `ADD m32, r32` or `INC m32`. In this case, IT replaces the instructions with a block of instructions, entailing a couple of calls to the SSMS's read and write APIs, and re-implementing the same logic with several CPU instructions. This implementation, although easy to carry out by relying on specific rules passed to Hijacker, of course adds some overhead. Nevertheless it allows to integrate our SSMS in a fully transparent way towards the simulation model writer.

High-level programming languages allow to access memory objects in a non-direct way, namely through the use of pointers. Since Hijacker works at compile time, it is not possible to statically determine whether a pointer will target a global variable or not. To cope with this issue, we use Hijacker to instrument any `mov` instruction which can handle pointers through a call to a `globvar_monitor` function which fastly determines if a pointer targets a global variable[1]. In particular, at compile time, via the usage of a custom ld-based linker script we insert symbols called `_bss_start`, `_bss_end`, `_data_start`, `_data_end`, within the application-level ELF executable, which mark off the area containing global variables. Upon a call to the `globvar_monitor` routine, a fast check on these boundaries is performed. If a pointer falls within this area, the operation is redirected to SSMS, on the other hand the original `mov` instruction is executed.

As a last note, x86/x86_64 instruction set provides *string instructions* which allow to perform operations on memory buffers instead of single memory locations. In particular, `movs` and `stos` instructions allow the program to copy or modify large buffers at once. In order to cope with the presence of these complex instructions, SSMS provides two additional APIs, namely `copy_buffer()` and `set_buffer()` which simulate the execution of these operations on version lists if they are found to target global variables (e.g., global arrays). Otherwise, they just execute the original `movs` or `stos` operations. Therefore, at compile time, IT replaces every string operation involving memory update with a function call to these APIs, accordingly. Similarly to the approach adopted in Section 4.2, `cmov` instructions are handled by replacing them with an assembly code snippet which mimics their semantic, and in turn relies on `mov` instructions (adopted to perform the memory update in case the condition is met) which are subject to the same instrumenting procedure.

[1]The way the address of the final memory write is computed is analogous to the one adopted in Section 4.2.

The last operation we perform at compile time is the inspection of the application-level ELF object file in order to extract information concerning global variables. In particular, by exploring the application object we extract from the symbol table `.symtab` all the `STT_OBJECT` / `STT_COMMON` symbols and store their name, address and size in a text file which will be later used at startup time for setting up the version lists. In this way, by exploiting the $\langle name, address, size\rangle$ tuple, we are able to transparently identify any access to global variables which will be likely used by the application-level code during the execution of the simulation model, allowing the programmer to rely on the complete set of constructs provided by ANSI-C. We note that, due to the multi-threaded nature of our reference simulation kernel, a global variable's address is a common information shared among the worker threads.

Since we address assembly `mov` instructions, we note that their opcode immediately provides information about the size of the memory operation. Therefore, we can easily rely on the `long long read_global_variable()` function, as its return type actually represents the largest type which can be accessed by an assembly instruction. Therefore, our injected code will simply copy the return value from this function into the proper used register (in case of an original `mov` operation from memory) using only the necessary bits for the associated `mov`.

6.1.2 Accounting for Third-Party Libraries

The possibility to rely on third-party libraries depends on whether they will be invoked on global variables or not. As in the case of ISS as presented in Chapter 4, we have explicitly addressed the case of read/write operations performed by third-party software, just focusing on `stdlib`. Specifically, we have modified the set of function wrappers presented in Section 4.6 which produce in-memory accesses via pointer passing. The wrappers simply check whether global variables are involved in the operation before applying the policy devised for ISS. In case a passed pointer targets a global variable (which can be again done easily by comparing its value with the address of `_bss_start`, `_bss_end`, `_data_start`, and `_data_end`), operations are redirected to SSMS APIs for accessing version lists.

6.1.3 Memory Map and Version Lists

As hinted before, SSMS explicitly targets shared-memory/multi-core machines. In order to significantly enhance performance, we have decided to avoid requesting to the underlying memory manager (namely `malloc`) memory segments on-demand, whenever SSMS needs to install some data structure. On the other hand, at simulation startup the master worker thread[2] installs a large memory segment which is partitioned according to the definition of the following structure:

```
typedef struct _globval_mem {
    int num_vars;
    globvar_info  variables[MAX_GLOBVARS];
    volatile int  first_node_free;
    globvar_node  *versions;
} globvar_mem;
```

In particular, the shared memory segment is divided into several fixed-sized portions. One portion, namely `variables`, is an array which is used to manage global variables. The choice of having only one memory segment, rather than a

[2]By *master worker thread* we mean one, among the ones available, which carries out this task alone. This can be the worker thread with logical thread id 0.

per-thread one, is because global variables can be accessed by all worker threads. In case of a read operation by some worker thread, finding the latest version would entail scanning all the per-thread data structures, which would entail a non-negligible overhead.

Upon initialization of SSMS, the configuration text file described in Section 6.1.1 is loaded and parsed. The field `num_vars` is used to keep track of how many variables are actually handled, and for each of them an entry in the `variables` array is populated. To allow a fast retrieval of the global variables, we use a fast hash function to determine which entry in the `variables` array will store the information associated with a specific variable. In particular, the position in the array is determined with a fast bitwise operation — namely, `address & (∼(-MAX_GLOBVARS))` — since `MAX_GLOBVARS` is set to be a power of two. Since at startup the total number of global variables is known, `MAX_GLOBVARS` is increased (always keeping it a power of 2) until the collision is less than 20%. This choice uses more memory than needed, but generates a significant speedup when accessing variables.

Anyway, in case collisions are found even after the increment, separate chaining is used as a means for finding a free place. Although this might seem sub-optimized, we note that global variables' virtual addresses are clustered in a contiguous portion of the address space, therefore the least significant bits are more likely to define a different key for each of them in the hash table. Each entry in the `variables` array is structured as:

```
typedef struct _globvar_info {
  void *orig_addr;
  unsigned short int size;
  long long head;
  long long tail;
} globvar_info;
```

where `orig_address` stores the global variable's original address, which is used as hash table's key; `size` describes which is the size (in bytes) of the global variable.

Since we are preallocating memory, version lists must be implemented using nodes scattered around the preallocated segment. In particular, `versions` is an array of fixed-sized nodes which can be used for any list, and `head` and `tail` are indices within this array, which is composed of entries structured as follows:

```
typedef struct _globvar_node {
  volatile int alloc;
  time_type lvt;
  unsigned char value[MAX_BUFF];
  spinlock_t read_list_spinlock;
  long long next;
  time_type read_list[];
} globvar_node;
```

where `lvt` is the ST associated with the version (i.e., the timestamp T_e associated with the event e during the execution of which the version was generated), `value` is the global variable's value, and `next` is used to identify which is the following

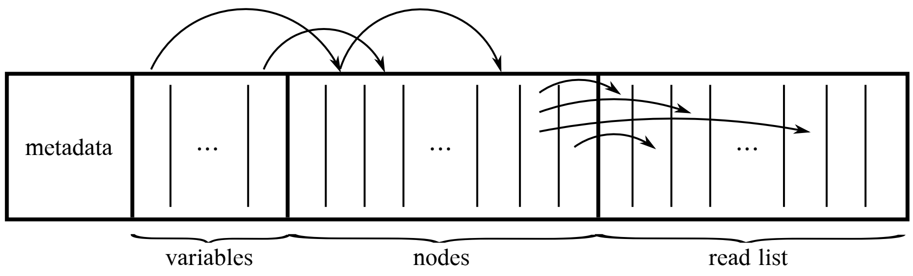


Figure 6.1: Preallocated Shared Memory Map

Algorithm 6.5 Shared Memory Allocation

```
procedure ALLOCATE
    $m \leftarrow$ GENERATE_MARK( )
    $slot \leftarrow$ first_node_free
    while $true$ do
        $alloc \leftarrow vers[slot].alloc$;
        if $alloc \vee \neg$ CAS($vers[slot].alloc$, $alloc$, $m$) then
            $slot \leftarrow$ next slot in circular policy
        else
            break
        end if
    end while
    atomically update first_node_free
    return $slot$;
end procedure
```

node in the list. A node can therefore be seen as a snapshot of the state of a single global variable at a certain ST. In Figure 6.1 we provide a complete picture of the preallocated memory map.

Node versions' entries can belong to any list, and given that lists are accessed without the use of locks, a special allocation function must be used, ensuring that no two worker threads running concurrently are given the same entry for handling two different versions.

The Allocate pseudocode is given in Algorithm 6.5. In order to allow concurrent accesses, it again relies on `CAS`. The `globvar_shmem` data structure holds in `first_node_free` the value of the first element of the `versions` array to start trying to allocate from. Its manipulation is based on the classical algorithm used by the Linux kernel for managing the bitmap of file descriptors associated with a process. Specifically, it is always atomically increased upon allocation, and gets atomically decreased in case an entry is released having index less than the first chunk currently available within that block. Starting from that slot, a kernel instance tries to allocate a node by storing via a `CAS` operation a non-zero value into the `alloc` field of `globvar_node`, which tells whether a node is currently in use. In case the `CAS` fails, the next node in the array is selected and the procedure is repeated, until it eventually succeeds[3]. The companion function RELEASE is much simpler, as it only entails resetting the `alloc` and updating `first_node_free` (using a first-fit approach) via an atomic set operation, implemented again relying on `CAS`.

In order to cope with the ABA problem [24], we have explicitly decided to consider a node allocated if the `alloc` field is non-zero. In particular, we store into it a unique value every time a node is allocated, so that two allocations can be identified as different. The macro `generate_mark` produces an integer value which is based on the *Cantor pairing function*:

$$\frac{(n_1 + n_2)(n_1 + n_2 + 1) + n_2}{2} \tag{6.2}$$

where we set n_1 to the worked thread logical id in the range $[0, N_{cores} - 1]$, and n_2 to the value of a monotonic per-thread counter which is incremented upon each call to `generate_mark`. This function is very fast, as it is mostly based on

[3]To check if the space is up, a counter of available free nodes is kept as well in shared memory, which is managed via a `CAS`-based *atomic decrement* operation.

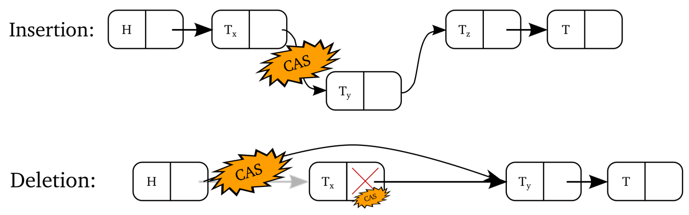


Figure 6.2: Non-Blocking Linked List Operations

integer operations, and allows to generate system-wide unique marks[4].

Once a node is allocated, it gets organized into a non-blocking linked list, which is implemented according to a modified version of the one proposed in [58]. Concurrent insertions are handled via the use of a single `CAS` operation, which is used to introduce the newly allocated node into the list by acting on the `next` field of the predecessor node. As for deletion, two `CAS` are used, one to mark the `next` field of the deleted node as *logically* deleted, and another to *physically* delete the node. We have slightly modified the algorithm in order to take into account our specific needs. In particular, the Find-Node procedure from [58] has been augmented in order to return the `alloc` field, to explicitly cope with the ABA problem, and the Insert procedure does not fail if a node with the same key (i.e., ST associated with the timestamp T_e of the generating event e) already exists. Specifically, the new node is simply linked after the originally existing one. In addition, we note that LPs are more likely to access versions associated with higher STs, since well partitioned/balanced optimistic simulations usually proceed relatively evenly. Therefore, we sort the versions in the lists in descending order, to avoid a complete scan of the list every time we want to find a node in it.

[4]`generate_mark` can of course return two equal values when the counter overflows, but this situation can happen after a significant WCT, so we consider it to be statistically non-significant for the ABA problem.

To avoid the ABA problem in linked lists, "pointers" (i.e., indices) to nodes are composed (every time they are updated) by a unique mark generated via the aforementioned macro `generate_mark` and the real index, allowing to capture the situation where two nodes are still adjacent but one was deallocated and then reallocated during the execution of the non-blocking algorithm by different kernel instances. The operations performed on the versions lists are depicted in Figure 6.2.

6.1.4 Accessing Version Lists

The API offered by SSMS provide two main functions to access global variables, namely `read_global_variable` and `write_global_variable`, which we will refer to as Read and Write from now on.

Read operation's pseudocode is provided in Algorithm 6.6. For efficiency reasons, before letting an LP execute a simulation event, SSMS sets up an *AccessSet*, i.e., a mapping between version nodes and variables. Whenever a variable is accessed for the first time, Find-Node determines which is the most suitable version for the current LVT, and the tuple $\langle slot, version \rangle$ is placed into *AccessSet* in order to speedup the retrieval of the version, avoiding the scan of the list upon subsequent accesses.

As for the Write operation, the pseudocode of which is presented in Algorithm 6.7, its behaviour is twofold, depending on whether it is invoked for the first time since the beginning of the current event's execution. In particular, upon the first access on a variable, the *AccessSet* for that particular event is populated. In any case, a call to Insert-Version is performed which, as stated in Section 6.1.3, creates a new version. The second part of the Write operation entails checking the *ReadList* for ensuring consistency, as it will be clearly

Algorithm 6.6 Global Variable Read

```
procedure READ($addr$, $lvt$)
    $slot \leftarrow$ hash table's entry associated with $addr$
    $hasRead \leftarrow$ false
    if $slot \in AccessSet$ then
        $version \leftarrow AccessSet[slot]$
    else
        while $\neg hasRead$ do
            $\langle version, alloc \rangle \leftarrow$ FIND-NODE($slot$, $lvt$)
            $AccessSet[slot] \leftarrow version$
            spin_lock(read_list_lock)
            if $alloc$ has been changed then
                spin_unlock(read_list_lock)
                continue
            end if
            add $\langle lp, lvt \rangle$ into $ReadList$
            spin_unlock(read_list_lock)
            $hasRead \leftarrow$ true
        end while
    end if
    return $vers[version].value$;
end procedure
```

depicted in Section 6.1.5.

6.1.5 Synchronization and Rollback Operations

In order to strengthen the optimism of our implementation, we allow interleaved reads and writes on a version list, and we explicitly avoid a version k installed at ST T_k to invalidate every version j such that $T_k < T_j$. In fact, we note that consistency is violated only if, at ST T_x an LP reads the version associated with ST T_y such that $T_y \leq T_x$, and at a certain point during the execution a new version node associated with ST T_z such that $T_y \leq T_z < T_x$ is installed.

This means that every process which reads a certain version node must leave

Algorithm 6.7 Global Variable Write

```
1: procedure WRITE($addr$, $lvt$, $val$)
2:     $slot \leftarrow$ hash table's entry associated with $addr$
3:     if $slot \in AccessSet$ then
4:         $version \leftarrow AccessSet[slot]$
5:         $vers[version].value \leftarrow val$
6:     else
7:         $version \leftarrow$ INSERT-VERSION($slot$, $lvt$, $val$)
8:         $AccessSet[slot] \leftarrow version$
9:     end if
10:    for all $\langle lp, lvt' \rangle \in ReadList$ s.t. $lvt' \geq lvt$ do
11:        send antimessage to $lp$
12:    end for
13: end procedure
```

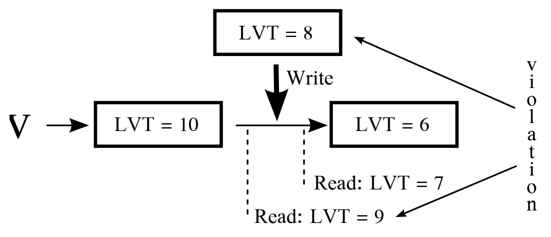


Figure 6.3: Occurrence of the Rollback Operation

a mark of that operation, i.e., visible reads [20] are enforced. In fact, as shown in Figure 6.3, we are interested in undoing only the events which have read a version older than the new one which has just been inserted.

To this end, we augment the classical notion of rollback as presented by the Time Warp synchronization protocol [76], by sending a special anti-message to all the LPs which have read a so-defined causally inconsistent version after any write operation. This is reflected into Algorithms 6.6 and 6.7. In fact, in the READ operation, before returning the variable's value, the couple $\langle lp, lvt \rangle$ is inserted into the *ReadList* for that particular version. This operation is included within a specially designed critical section to ensure consistency. In fact, a spinlock for that particular *ReadList* is taken, ensuring that no other process will start the rollback operation while the *ReadList* is being updated. Otherwise, this scenario would produce a non-trackable read operation.

In addition, after the spinlock has been taken, a check on the variation of the `alloc` field for that particular version is performed, so to avoid the ABA problem due to a critical race between the deallocation/allocation procedure and the *ReadList* update. At the same time, at the end of the Write operation, the *ReadList* of the left node is checked in order to find all the LPs which have read the previous node's value, while they were requesting a version at a ST such that they should have read the one in the version which was just installed. Although the list is linked in only one direction, given the implementation of Find-Node, locating the previous node is immediate, as it is in the current left node.

We note that another step must be undertaken in order to ensure correctness. In particular, whenever a special antimessage is received because of an inconsistent read, any version node installed due to that particular event must be removed. To this end, we augmented the concept of event queue and modified the Write function so that whenever a node is installed during the execution of an event, the event queue keeps track of this operation via pointers to the node created during the event's execution. In case a rollback operation undoes that event, the node is removed from the version list, and the *ReadList* is scanned for sending antimessages to every LP which has read that particular node.

6.1.6 Memory Recovery and Management

We extend the notion of *fossil collection* by defining the *version list pruning* operation. In particular, upon GVT computation, the version lists associated with global variables are scanned in order to find which is the first node i stamped with $t_i \leq GVT$ and that node is selected as the barrier node. Any node marked with a timestamp $t_k < t_i$ is marked as free and removed from the list. For implementations where there is no actual event processing during GVT computation (like the one which we rely on), the version list pruning is thread safe, and can therefore be executed efficiently, with no need to synchronize the access. In particular, the various lists can be evenly divided across the various worker threads, and each one performs the memory recover executing in isolation. This choice provides a more efficient execution and still ensures correctness.

In case the memory buffer preallocated for keeping the version nodes gets filled, we rely on `realloc` to double its size. In particular, whenever a worker thread finds the buffer full, it relies on a `CAS` operation to atomically set the `first_node_free` field to the value -1, which tells all the other worker threads that someone is already resizing the structure. After the `realloc` is executed, the current worker thread relies on a second `CAS` to set `first_node_free` to the first position available in the new portion of allocated memory.

6.2 Correctness of the Approach

The SSMS algorithm allows dispatched LPs to concurrently access global shared variables in an optimistic way and postpones synchronization among concurrent read/write operations executed on the shared-state only whenever a conflict materializes. Therefore the implemented concurrency control scheme maintains a high degree of parallelism by ensuring that:

1) the read/write operations executed by a committed event e on the shared-state appear as they happened at same indivisible point in time associated

with ST T_e in which e has been processed;

2) all the committed events execute the same operations and produce the same outcome as they were processed sequentially without violating logical virtual time advancement.

For this reason, if we model an event e's execution as an atomic transaction τ_e [14] to be considered committed whenever e is committed according to the Time Warp algorithm (i.e., at a WCT instant t it can be established a GVT value $GVT(t)$ such that each event e' executed at a ST $T_{e'} < GVT(t)$ cannot be revoked anymore and $T_e < GVT(t)$), we can adopt the *serializability* consistency criteria [14, 1] over the histories of the committed events as the target correctness criteria of the proposed solution.

Even if in practice SSMS behaves as an STM system, we have not designed it having in mind the typical correctness criteria guaranteed by STMs, namely *opacity* [57]. In fact, guaranteeing that every read operation always returns a value from a consistent state of the shared memory would not prevent an LP to see an inconsistent state of the simulation due to the *risk* associated with the Time Warp algorithm: a user model code may be executed using data arguments that are inconsistent with the logical state of the code [116].

Before showing the proof we formalize the concepts of *history* on committed events and *operation*. A history $H_{GVT(t)}$ over a set E of committed events e at the GVT value $GVT(t)$ consists of:

1) a partial order of operations that reflect the *write*/*read* operations performed within e on the simulation shared-state together with the *begin* (i.e., the invocation of e) and the *complete* (i.e., the commit of e);

2) the version order $\ll$ that specifies a total order on the object's versions created by committed events. A *write* operation on an object x issued by

an event e is denoted by $w_e(x_e)$ while a *read* operation on a version $x_{e'}$ of object x is denoted by $r_e(x_{e'})$.

We can build a Direct Serialization Graph $DSG(H_{GVT(t)}, \ll)$ over a history $H_{GVT(t)}$ as stated in [1] in order to define serializability in terms of topological properties on that graph. In particular a graph $DSG(H_{GVT(t)}, \ll)$ contains a node N_e for each committed event e in $H_{GVT(t)}$ and a directed edge $N_e \rightarrow N_{e'}$ for each pair of committed events e, e' in $H_{GVT(t)}$ such that one of the following dependencies occurs: (i) e' directly *read-depends* on e if there exists an object x such that e' executes a read $r(x_e)$; (ii) e' directly *write-depends* on e if there exists an object x such that e executes a write $w(x_e)$, e' executes a write $w(x_{e'})$ and $x_{e'}$ immediately follows x_e in the total order defined by $\ll$ on x; (iii) e' directly *anti-depends* on e if there exists an object x and a committed event e'' such that e executes a read $r(x_{e''})$, e' executes a write $w(x_{e'})$ and $x_{e'}$ immediately follows $x_{e''}$ in the total order defined by $\ll$ on x. Then a history $H_{GVT(t)}$ is serializable if the associated $DSG(H_{GVT(t)}, \ll)$ does not contain oriented cycles as defined in [14].

Therefore the correctness proof of SSMS is formalized in the following Theorem:

Theorem 6.1. *At any WCT instant t, for the associated value $GVT(t)$ and for each history $H_{GVT(t)}$ of committed events admitted by the SSMS algorithm then the $DSG(H_{GVT(t)}, \ll)$ graph does not contain any oriented cycle.*

Proof. We prove that the $DSG(H_{GVT(t)}, \ll)$ does not contain any oriented cycle by showing that for each edge $N_e \rightarrow N_{e'}$, $T_e < T_{e'}$ always holds.

If an edge $N_e \rightarrow N_{e'}$ is in $DSG(H_{GVT(t)}, \ll)$ we have to distinguish three cases:

1) e' directly *read-depends* on e. In this case SSMS has performed a read operation on an object x by returning the version x_e having the greatest logical virtual time T_e less than $T_{e'}$. Therefore $T_e < T_{e'}$.

2) e' directly *write-depends* on e. e' overwrites a value (by adding a new version $x_{e'}$) of an object x already written by e. This is admitted only if $T_e < T_{e'}$.

3) e' directly *anti-depends* on e. e' adds a new version of an object x after the version read by e. If $T_e \geq T_{e'}$ holds then SSMS forces a rollback for e. Since both e and e' are committed then $T_e < T_{e'}$.

□

By Theorem 6.1 follows that every committed history generated by SSMS does not violate serializability.

6.3 Experimental Evaluation

6.3.1 Test-Bed Application and Configuration

As a test-bed, we have used *Personal Communications Service* (PCS), as described in Section 2.5.3.

Calls inter-arrival time is exponentially distributed, and average duration is set to 2 minutes. The expected rate for call inter-arrival has been set to achieve channel utilization factor on the order of 15%, while the residence time of an active device within a cell has a mean value of 5 minutes and follows the exponential distribution.

To evaluate the efficiency of our proposal, we have extended the simulation model having a set of global variables handling global statistics. In particular, upon each event's execution, the total number of calls, the total number of handoffs, and the global cumulated power is updated in the shared state. In addition, we have re-implemented the model in order to have a centralized LP keeping in its disjoint simulation state the global attributes, as in the classical definition of PDES states as of Equation (6.1). Every LP willing to update a shared attribute issues a message request to the centralized LP, which in turn sends back the current value. Any update on the current value is then sent as another message to the centralized LP. In this scenario, every message exchanged with the centralized LP is marked with the same timestamp as the event's which generated the read/write flow. Therefore, this baseline version of the benchmark is a simple zero-lookahead request-reply approach, which we do not expect to scale well with respect to the number of simulation kernel instances being run.

For the above scenario, we have run experiments with 64 wireless cells, modelled as hexagons covering a square region, each one managing 1000 wireless channels. We note that the choice of setting the total number of LPs to 64 is related to the fact that, when the number of LPs is decreased, they exhibit a higher degree of parallelism. This in turn affects the variations of LVT associated with each LP. In fact, the lower the number of LP, the higher the probability of a skew in the value of their clocks. Therefore, this experimental setup allows us to evaluate our proposal with a non-negligible rollback probability, so that our experimental results will better capture the effect of the version-lists rollback operation on the overall performance. We have measured the cumulated event rate (expressed as the amount of cumulated committed events per Wall-Clock-Time unit), which is a classical indicator of the speed of the optimistic simulation run.

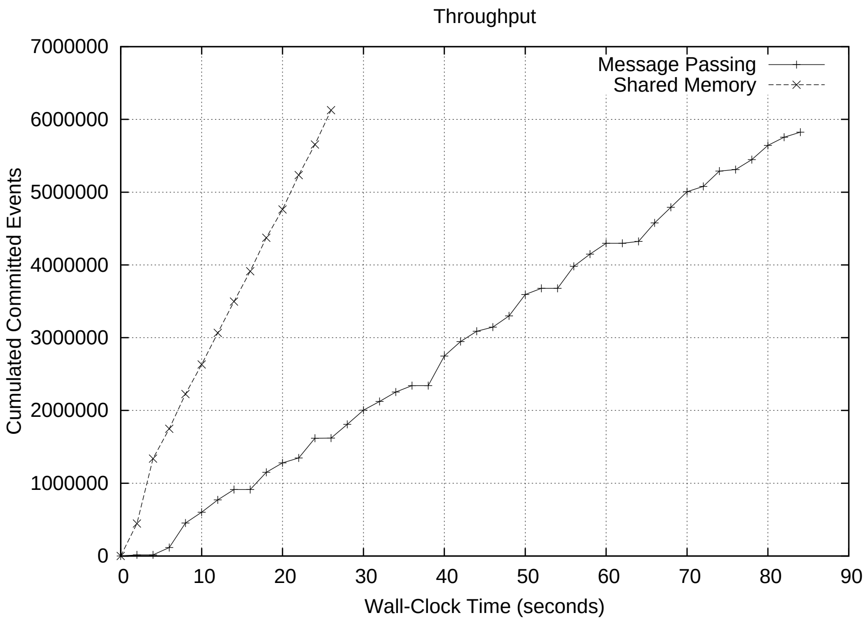


Figure 6.4: Throughput Running on 32 Kernel Instances

6.3.2 Results

In Figure 6.4 we present the throughput associated with our proposed test-bed model run on top of 32 simulation kernel instances, each one running on a private CPU-core of our test machine. By the results, we can see that the execution of the simulation model relying on our SSMS provides a speedup in the order of 70%. In addition, we note that there is a tangible difference between the two curves' trends. In fact, the throughput associated with the SSMS execution has a constant growth, which suggests a constant event commitment rate. On the other hand, the centralized-LP implementation's slope shows fluctuations, which are related to the large amount of events associated with variables' reads/updates which must be processed. Therefore, the number of committed events per GVT interval is not constant, due to the fact that the amount of workload

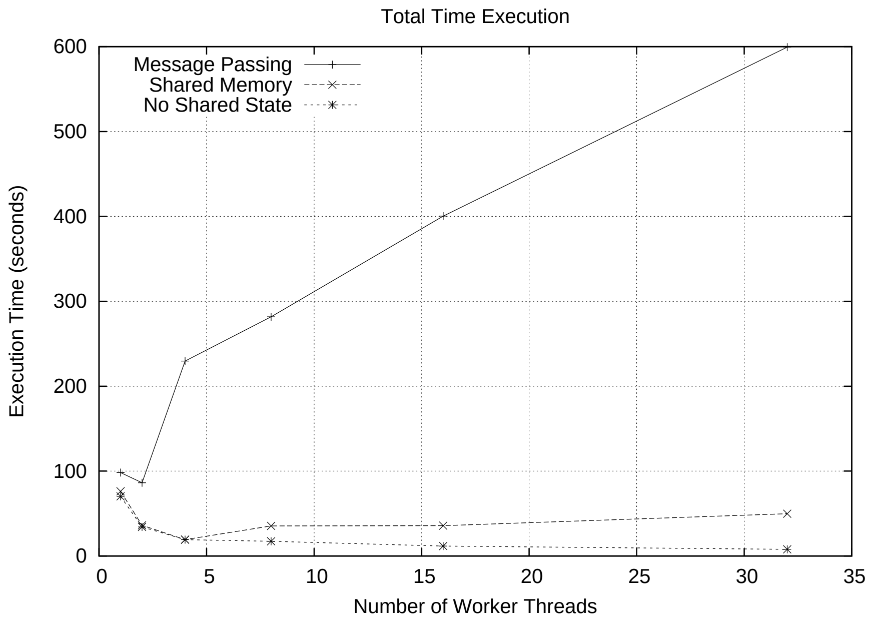


Figure 6.5: Scaling with respect to the Number of Parallel Instances

processed by differentiated LPs is totally different and that the LVT of the LP keeping the shared state diverges from the other LPs' one (this can entail a higher rollback probability), a scenario which is not present at all when relying on the multi-version lists in the shared memory version case.

At the same time, Figure 6.5 shows the total execution time of the simulation towards the number of parallel worker threads on which the model is run. In addition to the set of experiments described before, we present also the curve associated with the classical implementation of the benchmark (used as well in previous experimental assessments), where the shared attributes are kept in the disjoint LPs' simulation states and are reduced at the end of the simulation. By the results, we can see that both the SSMS and the centralized-LP implementation suffer from some form of thrashing. In fact, the centralized-LP version provides a speed-down in the order of 100% when the model is parallelized on top of 4 parallel kernel instances, while SSMS shows the same behaviour (although of a reduced magnitude) starting from 8 parallel kernel instances. The version with no shared state shows a trend which is the one expected by a parallel simulator.

We note that in this configuration, the SSMS's speedup towards the centralized LP is very large. Of course, the overhead in the centralized-LP case could be leveraged by having different LPs handle different variables, but this solution would not scale well towards the size of the shared state in the simulation model, and additionally it has a reduced degree of transparency.

Finally, we note that the simulation model used to assess the validity of our proposal is a worst case for our architecture, since at every event's execution some updates on the global variables are performed, producing a large contention on the linked lists. A simulation model which relies on shared-state for synchronization rather than for global statistics would benefit much more from the proposed architecture.

Chapter 7

Cross-Accessing Logical Processes' States

As two floating planks meet and part on the sea,
O friend! so I met and then drifted from thee.
— William R. Alger, Poetry of the Orient (1865)

In Chapter 6 we have shown how is it possible to provide runtime supports which allow to modify Equation (2.5) and in order to drop a part of the constraints on simulation states. This has allowed us to reduce it to Equation (6.1). In this Chapter, we present additional runtime supports, which allow us to drop the remaining constraints, so that there is no longer any limitation on what the simulation model writer can do on simulation states' variables. This will be done focusing on the x86_64 instruction set, and providing an innovative memory manager for the Linux kernel.

This allows us to finally re-adapt the traditional PDES paradigm, taking into account that the advent (and the large diffusion) of shared-memory parallel machines, such as multi-core and SMP machines, offers the technical possibility to directly share state information across different LPs. In this way, we support correct concurrent execution of LPs while jointly allowing the possibility to directly share data and masking synchronization (and hence actual parallelization) to the application programmer.

The final target along the path of supporting shared data across different LPs actually translates into enabling a sequential-style programming approach (augmented with the concept of "object", which allows for improving expressiveness while coding complex simulation models), characterized by full access capabilities to any valid memory location (logically belonging to the state of any involved object) upon executing whichever event, in either read or write mode.

A motivating example for the relevance of the proposed solution comes from simulation frameworks for Transactional Data-Grid systems, such as NoSQL data stores (see, e.g, [34]), where it is common to simulate distributed commit protocols by having the simulated coordinator to schedule the arrival of a *prepare request* event to the involved sites, which needs to carry information about the write set and, in some cases, also the read set of the committing transaction. These sets may entail hundreds of data-item keys, and are populated at the coordinator while simulating the execution of the transaction. These sets are therefore instantiated by the transaction-coordinator LP within its local state, and would need to be packed and transmitted via events upon starting the simulation phase of the distributed commit protocol in case of traditional PDES implementations. This poses communication overhead problems and requires simulation-model code for marshalling read/write sets as simulation event payloads.

However, sharing based exclusively on global variables limits the actual possibility to share data in size, given that the storage for global variables is statically defined at compile time. Also, it still constraints the programmer, who is not allowed to directly access arbitrary slices of (dynamically-allocated) memory destined to keep portions of the simulation model (e.g., by having them logically representing the state of a generic LP).

In this Chapter we exactly tackle the above problem, namely how to support direct access by concurrent LPs to memory locations that are dynamically allocated by any LP, and logically included within its local state via, e.g., pointer based referencing. The same pointers can be used as payloads of events so that the recipient LP can use them to directly access the local state of a different LP, in either read or write mode. This breaks disjointness in memory access at the programming level, hence enabling the support for sequential-style DES programming (where any valid memory location keeping a portion of the state of whichever LP, is accessible while processing any simulation event), and creates a new kind of dependency that we term *cross-state* dependency, which stands as complementary with respect to the classical event dependency proper of PDES. On the other hand, guaranteeing correct (e.g., causally consistent) execution of simulation events in the presence of cross-state dependency across concurrent LPs requires proper application-transparent synchronization mechanisms to be put in place, which we provide in this Chapter.

The technique has similarities with the proposal in [162], where the message exchange was supported via shared-memory segments passed via pointers. Here, we complement such a strategy, as we allow pointers to appear as the payload of traditional events.

Overall, in this Chapter we will:

1) present the design and implementation of an innovative memory management architecture, oriented to Linux systems, which allows to detect the

materialization of cross-state dependencies across LPs that are run concurrently, in an application transparent manner. The architecture requires a minimal patch to the Linux kernel, given that it is almost exclusively based on an external loadable module.

2) present a synchronization scheme, entailing speculative processing, which takes into account both event and cross-state dependencies and allows the parallel run to mimic a classical sequential one where the simulation events are processed in non-decreasing timestamp order, while jointly being allowed to access any valid memory location belonging to the state of the simulation model (namely any memory location logically belonging to the state of some LP). On the other hand, our scheme, which we name *Event and Cross-State Synchronization* (ECS), allows running simulation events destined to different LPs concurrently (again transparently to the programmer).

7.1 Event and Cross-State Synchronization

7.1.1 Cross-State Dependency Tracking

In this section we present the memory management architecture we have designed and developed in order to support cross-state dependency, and to actually track the materialization of such type of dependency across LPs that are run concurrently. Let us stress again that our architecture supports cross-state dependency in a fully transparent manner with respect to the application level software. We have specifically designed a memory management architecture allowing not to loose the benefits from multi-threading.

We explicitly target the scenario where multiple threads can take care of

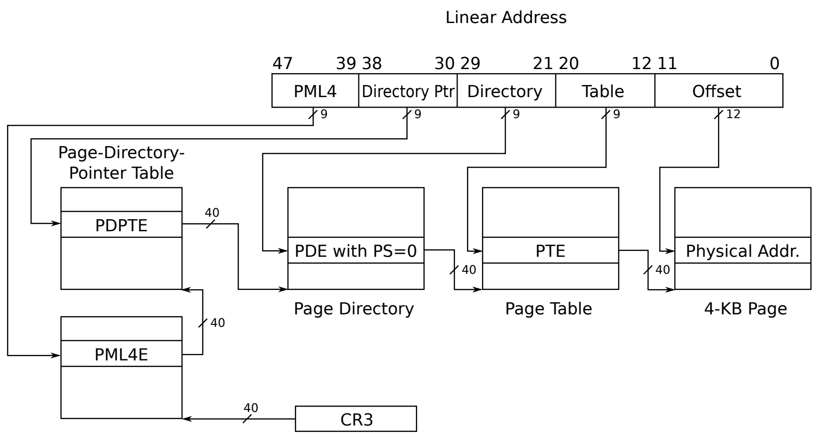


Figure 7.1: The Paging Scheme in x86_64 processors

dispatching whichever LP for execution (although we will still rely on temporary-binding schemes between LPs and worker threads in order to cope with, e.g., locality and other performance-related aspects).

In our architecture, virtual memory is destined for usage to any LP according to *stocks*. We have inserted a new layer for managing memory right under DiDyMeLoR, which was described in Chapter 4. More in detail, when the LP requests new memory buffers (which we support via the traditional `malloc` service, redirected to a proper memory allocator, as described in Chapter 4), the memory management architecture reserves an interval of page-aligned virtual memory addresses, namely the stock, which is achieved via the standard `mmap` POSIX API. We note that any page in the stock is an empty-zero page, thus being not really allocated in memory till the first read/write access to it is performed. This is the standard management performed by POSIX (e.g., Linux) systems.

To understand how we use the stock for supporting cross-state dependency tracking, let us consider the actual paging scheme offered by x86_64 architectures[1]. As shown in Figure 7.1, any 64-bit logical address has only 48 valid bits, which are used as access keys for a 4-level paging scheme, ultimately supporting pages of 4KB in size. The top level page table is called PML4 (or also PGD - Page General Directory) and keeps 512 entries. All the other page tables, operating at lower levels, have 512 entries each as well. In our design, the stock of virtual memory pages destined for allocation of memory buffers for a given LP corresponds to the set of contiguous virtual pages whose virtual-to-physical memory translation is associated with a single entry of the second-level page table, which is called PDP – Page Directory Pointer—its entries are therefore referred to as PDPTE. Note that a single stock corresponds to 512^2 pages, for a total of 1GB of virtual memory. Hence reserving a single stock for a LP allows managing an LP-state requesting up to 1GB of (dynamic) memory. On the other hand, reserving multiple stocks for a same LP will lead to manage LP states reaching multiple GB in size.

We have created a special device file, whose driver is loaded into the Linux kernel via an external module, which can be handled via proper `ioctl` commands, associated with specific logic coded within the driver. The `SET_VM_RANGE` command allows the special device to register the stocks to be reserved, and their association to the LPs (which are again distinguished via classical unique numerical identifiers). When this command is issued, the state of the device file changes so that the driver sets up a kernel-level map (accessible in constant time) where for each reserved stock, which is logically related to one entry of a PDP page-table, the identifier of the LP destined to use that stock is recorded. In Figure 7.2 we show an example where a given PDP table has its 0-th entry—hence the corresponding stock of virtual memory pages—reserved for LP_x, and

[1]Technically, this same methodological solution can be adapted to x86 architectures as well, with some modifications to the memory manager.

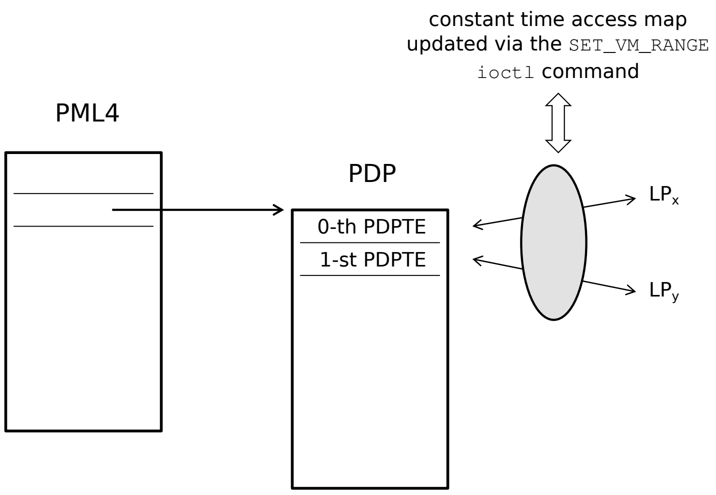

constant time access map updated via the SET_VM_RANGE ioctl command
PML4
PDP
0-th PDPTE
1-st PDPTE
LPx
LPy

Figure 7.2: Example Association between Stocks and LPs.

its 1-st entry reserved for LP_y.

By this kind of organization, if LP_x accesses any virtual address included in the stock reserved for LP_y, we know that such a memory access (which can be either in read or write mode in our execution model) is occurring outside the boundaries of its local state, and is actually involving the state of another LP. Therefore, we are experiencing a cross-state dependency. We recall again that this may occur, e.g, if LP_y scheduled a simulation event destined to LP_x, carrying as payload the pointer to some memory buffer belonging to the state of LP_y, just to indicate to LP_x where to take (and possibly update) the information requested for processing the event.

The core problem to cope with in order to exploit the stocks as the means to capture whether the generic LP_x (currently dispatched for execution along any worker tread WT_i within the PDES platform) is materializing a cross-state dependency is related to how to determine that event processing gives rise to a memory reference falling outside the boundaries of the stocks currently reserved

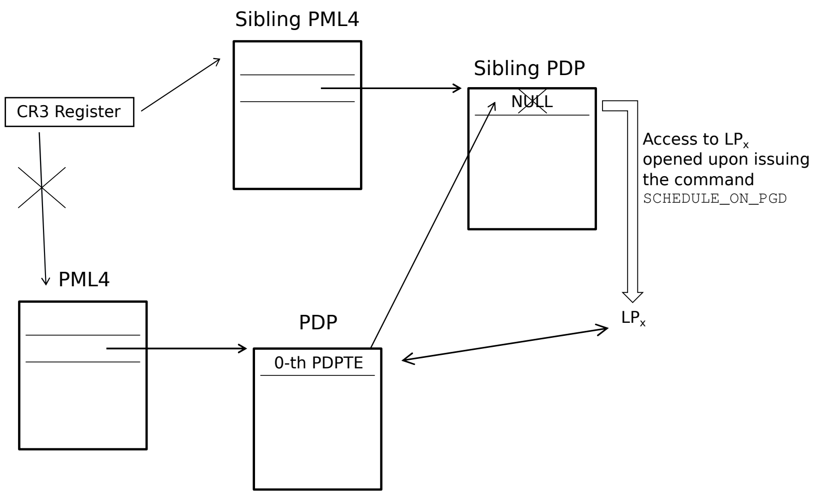

Sibling PML4
Sibling PDP
CR3 Register
NULL
Access to LP$_x$ opened upon issuing the command SCHEDULE_ON_PGD
PML4
PDP
0-th PDPTE
LP$_x$

Figure 7.3: LP_x's Memory Stock is opened for Access

for LP_x. We note that classical memory protection mechanisms supported by the operating system (and related segmentation-fault handling schemes) are not suited for our purposes. Particularly, given that we are targeting multi-threaded PDES platforms, we cannot simply a-priori protect the accesses to stocks that are reserved for LPs other than LP_x upon dispatching LP_x along any worker thread WT_i. This is because these LPs might be requested to run concurrently with respect to LP_x along other worker threads, which all share the same page table and experience the same protection rule as WT_i. Overall, closing to WT_i the access to the stocks not reserved for LP_x upon dispatching it (e.g., via the `mprotect` POSIX API) would lead to a change in the state of the page table where any other thread would not be allowed to access those stocks. This would clearly hamper concurrency, also leading to unneeded memory faults (by threads running logical processes other than LP_x) in contexts where LP_x requires no access to "remote" stocks while processing the event.

In order to cope with the above depicted core issue, we have devised a memory management architecture where any worker thread WT_i is associated with a sibling PML4 page table, whose entries point to sibling PDP page tables. The sibling page tables (both PML4 and PDP) destined for usage by a worker thread can be instantiated by relying on the `GET_PGD` command included in the special device file driver, which returns a descriptor for subsequent operations. By default, the entries of the sibling PDP page tables, which are associated with the stocks that have been destined for usage by the LPs, are all set to NULL. This means that they do not allow to reach the lower-level page tables, hence not allowing access to any already allocated stock (therefore, any attempt to access the stocks will lead to a memory fault). On the other hand, when WT_i dispatches LP_x for event execution, the entries of the PDP sibling tables that correspond to the virtual memory stocks destined for usage by LP_x are "opened" to correctly allow the retrieval of the lower-level page tables that contain the actual mapping of virtual-to-physical memory (or indications about whether the pages are not present, e.g., they are swapped-out pages). This is done by copying the corresponding entries of the original PDP tables onto the destination entries within the sibling PDP page tables (see Figure 7.3 for an example scenario where the stock associated with LP_x is again related to the 0-th entry of a given PDP page table).

In our architecture, this operation can be executed by relying on the additional `SCHEDULE_ON_PGD` command that we have included within the special device file driver, which can be issued via the `ioctl` interface. In other words, by using this command, the worker thread is allowed to switch into what we refer to as *simulation-object mode*, where the unique stock accessible is the one associated with the dispatched LP (say LP_x in the example discussion), while the other stocks are not accessible (given that their corresponding entries into the sibling PDP page tables are still set to NULL). As sketched in Figure 7.3, in our implementation this operation also leads to a change of the `CR3` register (namely, the page table pointer register in x86_64 processors), thus allowing to switch to the sibling PML4 for virtual-to-physical address resolution purposes.

Having different sibling PML4 tables, associated with the different concurrent worker threads, leads to the possibility to concurrently dispatch and execute different simulation objets (this is done by having each worker thread opening the access to the stocks associated with the LP it is currently dispatching) while still having the possibility to determine whether any of the dispatched LPs is confining its memory references within its own stocks. The assumption underlying this type of organization is that, when there is the need for opening access to a given stock, the corresponding memory management information is already present in to the corresponding PDP entry of the original page tables. This is not guaranteed by simply validating virtual memory addresses via `mmap`, which as hinted leaves memory into the empty-zero state. To overcome this problem, our architecture entails a stock allocation policy that beyond calling `mmap`, also explicitly writes a null byte into one single virtual page of the stock (the initial one). In this way, the Linux kernel traps the access to empty-zero memory and allocates the whole chain of page tables for managing the pages within the stock (although a single one of these pages is really allocated), which guarantees the existence of the PDP entry associated with the slot, to be filled into the corresponding sibling PDP entry upon dispatching the LP owning the stock.

Two additional points need to be discussed. First, having all the stocks closed for access by the worker thread, except the one(s) related to the dispatched LP, leads (as noted before) to memory faults in case of a memory access to stocks other than open one(s), namely in case of materialization of a cross-state dependency across concurrent LPs. However, these faults cannot be tracked (and handled) via classical segmentation-fault handling given that the "remote" stocks have already been validated via `mmap`, and the Linux kernel would simply lead the fault to reallocate the whole chain of page table entries for mapping the accessed virtual page in memory. This would lead the whole system to a state where for the same virtual page we would have multiple chains of page table entries representing its state (e.g., the frame used for mapping the page, which might be different along the multiple chains of page table entries) which is a discrepancy not directly manageable by the Linux kernel (except if using invasive patches). To avoid this scenario, upon installing the driver for the special device file, via loading the external module, we change the IDT table (directly accessible via the `IDT` register) in order to redirect the page-fault handler to an ECS-specific handler (rather than the original `do_page_fault` kernel function). In case the fault is not related to accesses to remote stocks within the sibling paging scheme, then the original handler is invoked. Otherwise, the ECS handler gives control back to user mode in order to let the PDES platform actuate ECS synchronization policies, exactly aimed at coping with cross-state dependencies. We also note that the expected overhead for this kind of memory-access tracking scheme is likely reduced with respect to classical tracking via segmentation fault.

This is because in our scheme, the ECS fault handler is activated via a simple passage into kernel mode, and then a passage back to user mode. It is not required to invoke the Linux scheduler, which is ultimately responsible for triggering the activation of signal handlers to be dispatched when the application eventually returns into user mode (after being hit by some signal, e.g., the segmentation fault one). As an additional note, upon a memory fault occurring on sibling PDP entries (due to cross-state dependency materialization) the

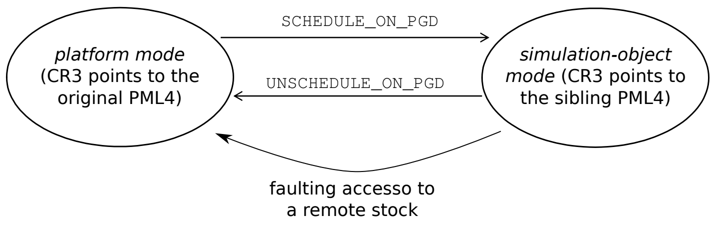

platform mode
(CR3 points to the original PML4)
SCHEDULE_ON_PGD
UNSCHEDULE_ON_PGD
simulation-object mode (CR3 points to the sibling PML4)
faulting accesso to a remote stock

Figure 7.4: State Diagram for Switch Operations between Page Tables

faulting thread is put back into what we call *platform mode*, which implies that it is switched back onto the original PML4. This is done to allow the worker thread to access any memory location required reconciling the execution of the concurrent LPs according to ECS synchronization. This aspect will be analysed in detail in Section 7.1.2. On the other hand, when the event processing activity naturally ends (due to the completion of operations executed by an event handler), the worker thread can switch back to platform mode on demand (hence gaining access to any memory location or data structure supporting the parallel execution) by using the `UNSCHEDULE_ON_PGD` command that we have implemented within the driver, which can be triggered by again exploiting the `ioctl` POSIX API. In Figure 7.4 we show the state diagram where the events causing the switch between simulation-object and platform modes are depicted.

Second, our architecture needs anyway to co-exist with the kernel scheduler, which poses issues on the side of managing the sibling PML4. Particularly, all the threads within a same Linux process share the same memory management information (the so-called memory context), including the pointer to the original page table. This pointer is used by the kernel scheduler upon re-dispatching the thread after it has been context-switched off the CPU. Particularly, this pointer is reloaded into the page-table pointer register `CR3` upon the occurrence of a context switch that gives control to the thread. However, if the thread was executing in simulation-object mode, `CR3` would need to be filled with the address of the sibling PML4 (rather than the original page table). To achieve this, a minimal patch to Linux has been adopted, which has been located right in the end of the kernel `schedule` function[2]. The patch simply checks whether the value of a special function-pointer we inserted into the kernel is not null, in which case the function pointer is invoked, which gives control to a proper `CR3` manager implemented within our external module. This manager checks whether the thread is running in simulation-object mode (which can be done by checking per-thread metadata that were setup upon the `SCHEDULE_ON_PGD` command invocation) and, in the positive case, it loads the sibling PML4 pointer into the `CR3` register (thus maintaining the simulation-object mode when running the thread). Note that the aforementioned special pointer is exported as a kernel symbol, and can be set to a value different from NULL upon inserting the external module. If this pointer is not set the Linux kernel behaves as usual, by simply restoring the `CR3` register according to the standard rules when the thread is rescheduled after a context switch.

As already mentioned, integration of Di-DyMeLoR with the currently presented architecture has been straightforward given that, rather than relying on actual `malloc` implementations for pre-reserving the segment destined to allocate the chunks for a given LP, in the integrated architecture we let Di-DyMeLoR rely on the stock allocator. Hence, the virtual memory segment managed by DyMeLoR boils down to the stock of virtual memory pages supported in the presented architecture.

As an additional note, our approach requires reloading the `CR3` register any time we switch between platform and simulation-object mode. The penalty

[2]To allow a pure-module approach, this task could be done by having the kernel module patch the `schedule` kernel function upon initialization.

incurred consists in flushing the TLB right upon loading a new value into `CR3`, which is done automatically by the firmware logic of x86_64 processors (this is anyhow required in order to make the access-rule of the target page table—original vs sibling—visible after the switch, which cannot be achieved without refilling the TLB). However, the data cache does not require to be invalidated, hence we expect that the cost for TLB renewal would look affordable as soon as a certain level of locality is exhibited while running either in platform or in simulation-object mode. As for this aspect, relying on Di-DyMeLoR would favour locality in simulation-object mode, given that Di-DyMeLoR implements policies aimed at maximizing virtual-memory contiguousness of the memory chunks delivered for usage by the LP.

Finally, we note that the on-demand switch to simulation-object mode or (back) to platform mode requires invoking the `ioctl` system-call. While the cost for system-calls has been traditionally considered an issue in high performance computing, especially when dealing with fine grain tasks, such costs are nowadays definitely reduced thanks to the `sysenter` and `sysexit` machine instructions, which are explicitly designed for low-latency system calls, by relying on operating systems with a flat memory model and no segmentation. These instructions have been optimized by reducing the number of checks and memory references so that a call or return has been shown to take less than one-fourth the number of internal clock cycles when compared to the traditional approach based on the `int` instruction, which was explicitly based on segment-gate retrieval and segmented-to-linear memory addressing translation.

Experimental data related to costs associated with TLB flushes, system-call involvement and the aforementioned management of the memory access faults via the ECS handler will be anyhow provided in Section 7.2.

7.1.2 The Event and Cross-State Synchronization Scheme

In this section we provide the core mechanisms underlying ECS synchronization. The main difference between classical event-based synchronization and ECS is that ECS-synchronization tasks let LPs process their events in non-decreasing timestamp. At the same time, any cross-state dependency is materialized at ST T to let the involved process (namely the one accessing remote stocks reserved for other LPs) observe the state snapshot that would have been observed at simulation time t in a sequential-run.

We base ECS synchronization on the following two innovations:

1) the introduction of temporary LP blocking phases, which may even lead to temporary block of the execution of an already dispatched LP (namely of an already dispatched simulation event at that LP);

2) the introduction of so called *rendez-vous* events, which are kinds of *system-level* simulation events not causing updates on the destination LP's state, but only driving block and unblock actions for processing activities of the LPs. These will be exploited to temporarily disable a LP to perform updates on its state along the ST axis, given that its state snapshot is currently involved in a cross-state dependency.

We note that point 1 leads to an event processing model where control (along any worker thread) can return to the platform layer before an already started event-processing phase actually ends. This takes place according to an interrupt-driven scheme, different in nature from event-preemption schemes that have been put in place in optimistic PDES systems, to squash the execution of events that are detected to be causally inconsistent while still being processed, for either performance or infinite-loop avoidance reasons [143, 116]. In fact, they have been typically based on polling (see, e.g., [143]) to be explicitly actuated by the event processing code, which is used to periodically query the platform layer to check whether no straggler event/antievent was delivered.

On the other hand, point 2 leads to bridge PDES execution models with Transactional Memory models, particularly by having read/write operations across different stocks serialized according to the logical time for their occurrence, thus making this an *obstruction-free* algorithm [60].

In our solution, each LP_x is associated with a cross-state dependency set that we refer to as CSD_x, which records the identifiers of all the LPs towards which LP_x has materialized a cross-state dependency while processing an event. CSD_x is initialized as empty upon dispatching LP_x for the execution of any new event, and gets possibly updated while processing the event.

ECS synchronization exploits the ad-hoc memory-fault management architecture presented in the previous section in order to detect that LP_x is accessing a remote memory stock (e.g., the stock associated with LP_y) in either read or write mode, while processing its next event (e.g., e_x). The identity of the LP towards which the cross-state dependency is being materialized (LP_y in our example discussion) is also known, given that the ECS memory-fault handler, which pushes the thread's execution back in platform mode, notifies such an identifier into the worker thread's user-mode stack.

The memory fault occurrence gives rise to the following algorithmic steps:

1) Execution of e_x is temporarily blocked, hence LP_x transits into a block state;

2) A *rendez-vous unique identifier* is generated and assigned to the event e_x, which we refer to as $rvid(e_x)$. The identifier can be easily generated according to differentiated schemes, such as by using a tuple $\langle x, count \rangle$,

where x is the identifier of the LP in charge of executing e_x and $count$ is a global unique counter value[3].

3) A special rendez-vous event e_y^{rv} is scheduled for LP_y, marked with both timestamp and rendez-vous if equal to the ones of event e_x (formally $T_{e_y^{rv}} = T_{e_x}$ and $rvid(e_y^{rv}) = rvid(e_x)$). We note that rendez-vous events are not generated by the simulation model, rather they are platform-generated events. Hence they do not have any associated processing rule at the application level, and must be therefore handled by the simulation kernel, without triggering the model's event-handler callback.

Rendez-vous events are incorporated into the event queue of the destination LP as if they were traditional events. Given that we are targeting optimistic synchronization, this means that a rendez-vous event may be a straggler event and might trigger a rollback operation. They must be therefore processed at the correct point of the destination LP's simulation trajectory, but the processing actions are platform-level ones proper of ECS, i.e. event handlers are not (and should not!) be defined for the events associated with rendez-vous management.

When LP_y is dispatched for processing a rendez-vous event e_y^{rv}, ECS performs the following algorithmic steps:

1) LP_y is put into a block state;

2) A special rendez-vous acknowledgement event e_x^{rva} is scheduled for LP_x, marked with no-timestamp but with the same rendez-vous identifier of e_y^{rv} (formally, $rvid(e_x^{rva}) = rvid(e_y^{rv})$.

On the other hand, when the rendez-vous acknowledgement event e_x^{rva} is delivered to the recipient LP_x, ECS performs the following steps:

[3]This can be again done using the Cantor pairing function.

1) it inserts the identifier of the sender LP, namely y into CSD_x

2) it puts LP_x back in the ready state (so that it could be eventually re-dispatched along some worker thread, thus resuming the execution of the originally interrupted event e_x).

At this point we know that LP_y is blocked (thus not being currently allowed to process its events), hence the snapshot of its state is available to LP_x for read/write operations, such as the operation that originally gave rise to the ECS memory fault and to the cross-state dependency being handled via the rendezvous. However, upon re-dispatching LP_x (which leads to resuming the processing of e_x), the involved worker thread cannot transit into simulation-object mode by only opening the stock(s) associated with LP_x into the sibling page tables. Rather, we also need to open access to the stock(s) associated with LP_y. In our architectural support, this can be still achieved via the `SCHEDULE_ON_PGD` command, given that it can acquire a set of stock identifiers to be opened in the sibling page tables when the worker thread transits into simulation-object mode. Particularly, upon re-dispatching LP_x, the `SCHEDULE_ON_PGD` command will be issued passing as input the set $x \cup CSD_x$, which for our example discussion, will contain the identifiers of both LP_x and LP_y.

The above algorithmic steps can be iterated in case cross state dependencies are materialized towards multiple LPs while processing the event e_x, which will lead to the scenario where LP_x can be rescheduled multiple times (while being in the processing phase of e_x) with incrementally enlarged sets of open stocks. On the other hand, once a remote memory stock (associated with a distinct LP) becomes open for access by LP_x during the processing phase of event e_x, any access to this stock by LP_x while processing this event will not cause any additional ECS memory fault.

We only need to discuss how the finalization of the processing phase of e_x is handled. Essentially, such finalization needs to generate notifications that the stocks associated with LPs towards which cross-state dependencies have been materialized are no longer locked for access by LP_x. Hence, the owner LPs can resume their normal processing activities (thus they can resume from the block-state). This is achieved via the following steps executed right after the processing of event e_x at LP_x:

1) an unblock event e_k^{ub} is sent towards any LP_k whose identifier is logged within CSD_x upon the end of the processing phase of e_x. These events are again not marked with timestamps, but with the rendez-vous identifier of the event e_x originating the cross-state dependency. Then CSD_x is reset as empty.

2) upon the delivery of e_k^{ub}, the recipient LP is simply put back as ready for being dispatched (hence exiting the block-state).

However, additional mechanisms are required in order for ECS to provide correctness and to also ensure progress of the parallel run.

Correctness

Given that ECS targets speculative processing, where LP blocking is not caused by native event dependencies, rather by the need for executing memory read-/write operations in multiple stocks as in-memory transactions, some care must be taken when handling rollback phases. Particularly, when we process an event e_x that gives rise to a rendez-vous event e_y^{rv}, we need to define rules for handling the rollback phase of either LP_x or LP_y at a ST $T' < T_{e_x}$ (or equivalently $T' < T_{e_y^{rv}}$). The peculiarity of this scenario is related to that e_x and e_y^{rv} are both causally related to each other. Particularly, if e_x is rolled back, then we need to rollback e_y^{rv} given that LP_x may have performed updates on the memory stocks destined to keep the state of LP_y while processing e_x[4]. On the other hand, the processing outcome of e_x is affected by values possibly read by LP_x from the stocks destined to LP_y at time T_{e_x}. In case these values change due to a rollback of LP_y at a ST preceding $T_{e_y^{rv}}$, the updated values should have been observed while processing e_x by LP_x.

In order to handle such mutual dependency, we devise the following scheme. When the event e_x is rolled back, we simply send an anti-event for the rendez-vous event e_y^{rv} that was scheduled while processing e_x. Given that e_y^{rv} was actually incorporated into the event list of the destination LP_y, the arrival of the anti-event gives rise to a classical annihilation that possibly rolls back LP_y to the latest processed event with timestamp less than $T_{e_y^{rv}}$. This solves the problem of rolling back LP_y due to the rollback of a rendez-vous generating event e_x on LP_x.

On the other hand, in case the rollback is originated on LP_y, and pushes this LP to a simulation time less than $T_{e_y^{rv}}$ (which leads to undo the execution of e_y^{rv}), the following actions are taken by ECS. A special rendez-vous-restart event e_x^{rvr}, marked with the original rendez-vous identifier (namely $rvid(e_x)$) is sent out towards LP_x. This special event annihilates the processing of the original instance (while not removing it from the input queue), which will lead to ultimately undoing e_y^{rv} via an anti-event). Given that when processed after

[4]In our memory management support for cross-state dependency tracking, we do not distinguish whether the dependency is originated by read or write operations (or both). In case the dependency was exclusively due to read operations, then LP_y might not be forced to rollback while rolling back event e_x (although a temporary reconstruction of the snapshot to be accessed in read mode would be requested in case the target LP_y run ahead of LP_x in simulation time). Distinguishing between read-generated and write-generated cross state dependencies will be the target of future work. Anyway, by artificially rolling back LP_y even in cases where no updates on its memory stocks were performed by LP_x while processing e_x is a conservative, safe approach.

the rollback, the event e_x will give rise to a rendez-vous marked with a different identifier (with respect to the rolled-back rendez-vous instance), no mismatch will occur in any annihilation phase for rendez-vous events associated with different incarnations of their generating event (which also avoids cycles in the annihilation process).

Also, all the other types of events used in ECS, such as acknowledgement and unblock events, are not actually incorporated into the event lists of the LPs, thus being inherently ephemeral, and not requiring particular care in the rollback scheme. These events can be simply discarded at the recipient side if the rendez-vous associated with their corresponding identifier (e.g., $rvid(e_x)$ in case of the acknowledgement event sent to LP_x upon the rendez-vous) is no more in place.

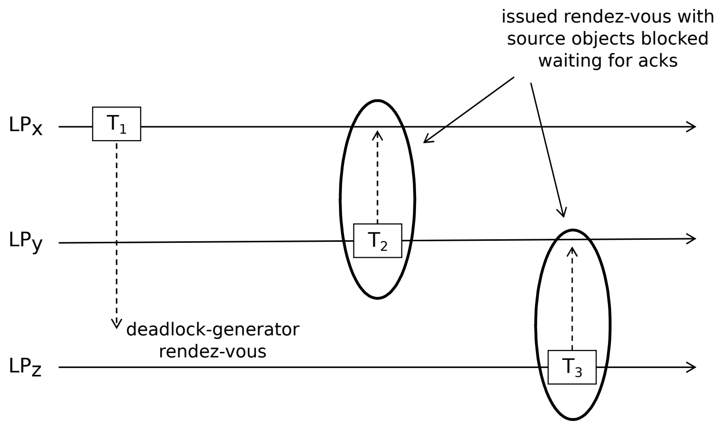
Progress

A bit more complex to deal with is the guarantee of progress in ECS. Specifically, care must be taken to avoid deadlocks and live-locks, and the domino effect in the rollback scheme. Let us first consider the deadlock/live-lock issue. A deadlock may arise in case of rendez-vous events involve a set of LPs in a cycle, where the rendez-vous associated with the minimum timestamp along the cycle leads the LP generating this rendez-vous to wait for the rollback of a different LP that is in turn in the block-state due to a different rendez-vous it issued, which needs to be completed. An example situation of this type is shown in Figure 7.5, where LP_x issues at ST T_1 a rendez-vous towards LP_z, which is in its turn waiting for LP_y to reach ST T_3 for a rendez-vous between LP_z and LP_y. On the other hand, LP_y is waiting for LP_x to reach ST T_2 for a rendez-vous with it. To avoid this deadlock scenarios, we can simply adopt the rule that, in case a rollback needs to be executed by LP_x which is currently blocked due to a rendez-vous it generated while processing and event e_x, this LP is simply resumed from the block-state by also squashing the finalization of the rendez-vous (this will lead to manage the rollback of the rendez-vous as explained above, e.g., by issuing the anti-event for the already sent out rendez-vous event).

We note that this implies that the current stack seen by the LP also needs to be refilled with correct information (since, upon resuming, its context will no more be the processing context for the rendez-vous generating event). We note that this is a problem similar to the one of restoring the correct stack for the LP upon resuming the processing of a rendez-vous generating event that lead it into the block-state (so that the worker thread currently executing this LP passes control to a different LP, which needs to operate on a proper stack image). Details on how we handled this issue in our implementation, where the cross-state dependency tracking architecture and ECS have been integrated into ROOT-Sim, will be discussed in Section 7.2.

We also note that annihilating the rendez-vous event via the corresponding anti-event is safe even in case the destination LP is currently blocked waiting for the finalization of the rendez-vous. In fact, it can be simply resumed from the block-state (again with proper stack image manipulation) and can be rolled back. This might require to alter its state image, which can be again safely done given that it is no longer going to be accessed by a different LP in a rendez-vous phase.

We note however that unblocking the LP generating a rendez-vous so as to prevent deadlock in case a rollback is required may, in its turn, lead to live-lock. Specifically, live-lock may in principle arise in case of simultaneous events materializing circular cross-state dependencies across multiple LPs. Each LP_x


issued rendez-vous with source objects blocked waiting for acks
LP_X
T_1
LP_Y
T_2
deadlock-generator rendez-vous
LP_Z
T_3
Figure 7.5: Deadlock Originated by a Rendez-Vous Generating Event

along the cycle, executing an event e_x at simulation time T_{e_x}, is hit by another object due to a cross-state memory faulting access at the same simulation time, which may lead to request the rollback of the events generating the rendez-vous circularly. This is known to possibly lead the rollback cycle to reappear indefinitely [92]. To overcome this problem, we need a priority management scheme for simultaneous events, that needs to be reflected also on the management of rendez-vous events. Particularly, if we have two events e_x and e_y such that $T_{e_x} = T_{e_y}$, and we have a priority scheme telling that $e_x \rightarrow e_y$ (namely, e_y is identified as causally dependent on e_x), then we need to enforce that any rendez-vous event e_y^{rv} generated by e_x is also causally related to e_y according to $e_y^{rv} \rightarrow e_y$. This way, the rendez-vous that are caused by events having the same timestamp are anyway sequentialized according to the priority scheme. We note however, that the guarantee of progress in optimistic PDES systems in the presence of simultaneous events is a more general problem, with respect to what we might experience in ECS, and has been extensively studied in literature [78]. Hence different literature solutions for tie-breaking simultaneous events (see, e.g., [106]) can be exploited for integration with ECS according to

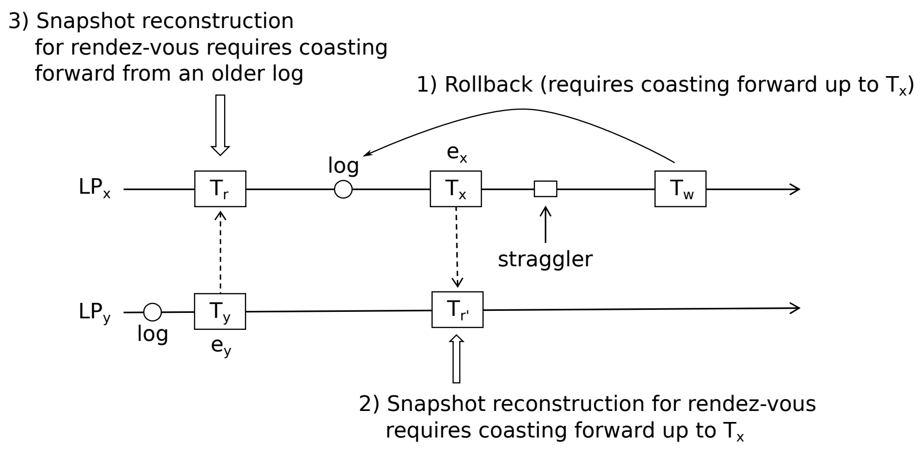

3) Snapshot reconstruction for rendez-vous requires coasting forward from an older log
1) Rollback (requires coasting forward up to T_x)
LP_x
T_r
log
e_x
T_x
T_w
straggler
LP_y
log
T_y
e_y
$T_{r'}$
2) Snapshot reconstruction for rendez-vous requires coasting forward up to T_x

Figure 7.6: Domino-Effect due to a Rollback

the scheme suggested above.

The final issue to cope with is domino effect in the rollback scheme. Particularly, by the ample literature on log/restore in optimistic PDES systems (see Section 3), we know that SSS, which avoids logging the LP state after the processing of each event, allows for optimizing the performance tradeoff between logging cost and restore cost. However, state restore at ST T requires the LP to be rolled back to the latest state log with time less than or equal to T, and then to fictitiously reprocess intermediate event up to T in a silent mode, namely with no interactions with other LPs—the coasting-forward phase. In ECS this is no longer possible since a coasting forward event might be a rendez-vous generating event. Hence, in order for this to be re-processed, the LP originally hit by the rendez-vous also needs to rollback at the time of the rendez-vous, so to provide its state snapshot for correct access by the LP performing coasting-forward. It is easy to show that this may lead the originally rolling-back LP to rollback further back along simulation time, according to the domino-effect.

An example is shown in Figure 7.6, where in order to execute the coasting forward involving event e_x at LP_x, we need to reconstruct the snapshot of LP_y at time T_{e_x}, But this leads to the need for processing e_y in a coasting forward, which in turn leads LP_x to restore to a time less than T_{e_y}. To avoid the execution of coasting-forward phases leading to rollback interactions with other LPs, our approach is based on complementing the selected SSS algorithm by forcing the log of the LP's state right after the processing of a rendez-vous generating event. This will lead to the scenario where no rendez-vous generating event will ever be included in the sequence of events between two subsequent snapshots of the same LP. Hence, no rendez-vous will ever be re-processed in any coasting forward phase. Additionally, rendez-vous-generated events must also be excluded by any coasting-forward phase, since for these events the rendez-vous source LP may have performed updates into the state of the involved LP. To avoid a rendez-vous-generated event to be included in any coasting forward, we can again force a state log of the involved LP right after the event is processed.

7.2 Experimental Evaluation

7.2.1 Implementation within the ROOT-Sim Platform

A few relevant modifications to the ROOT-Sim simulation platform have been made to integrate the presented ECS Scheme. Most relevantly, we have created stack-separation across the different LPs, by locating the stack of each LP in the initial part of a stock of memory destined for LP usage[5]. Also, execution resume in the different stacks, by also providing the correct processor and stack image, has been supported via `sigjump` and `longjump` POSIX API functions. They have also been used as the support for, e.g. squashing the stack image

[5]This can be done at simulation startup phase by requesting a per-LP memory buffer (which will per used as LP stack) to the stock allocator to which Di-DyMeLoR requests memory buffers.

in case a rollback occurs while the LP is in the block state (which eventually leads the LP to resume execution with a different context). As for the (temporary) binding of LPs to threads, we still relied on the already supported load sharing policies presented in [169, 170, 171]. Also, the LPs currently bound to a given worker thread are still dispatched according to the Lowest-Timestamp-First policy. However, LPs that are in the block state are not considered in the dispatching process (thus being again eligible for dispatching only after exiting the block state).

7.2.2 Results

This section is divided in two parts. Initially we provide data for an evaluation of the overhead (and hence of the feasibility) by the core memory management support underlying ECS. To this end we use PCS, natively entailing disjointness of memory accesses by the different LPs. After, we present data related to the assessment of the whole ECS architecture, by also comparing the run-time behaviour of models coded in such a way to be run on top of ECS (hence coded in sequential-style with no disjointness of memory accesses across different LPs) with respect to the counterpart exclusively based on traditional PDES programming (only relying on event-dependencies via message passing). In this part of the study we rely on NoSQL data grid model.

Overhead Assessment

We evaluated the overhead introduced by the presented memory management architecture by relying on a PCS model with 1024 wireless hexagonal cells covering a square region, each one managing up to 1000 wireless channels. Two specific aspects are relevant for this study:

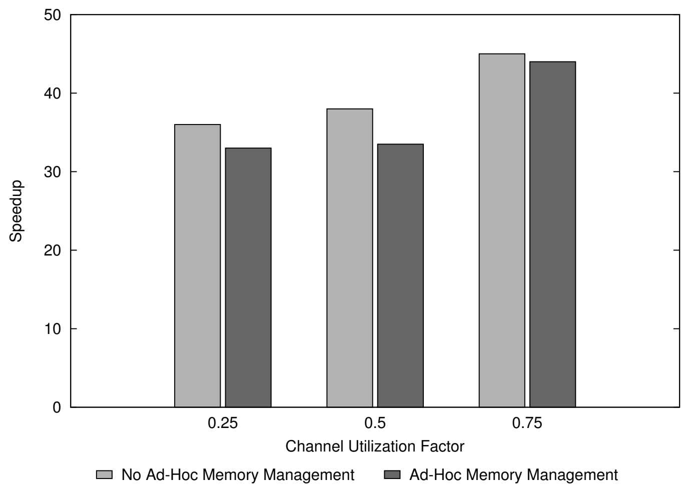


Figure 7.7: Relative Speedup: Memory Management vs Sequential Run

1) each LP models an individual cell, and the interactions between LPs exclusively take place via handoff events of mobile devices across different cells (hence memory accesses by the different LPs are intrinsically disjoint);

2) the average granularity (CPU requirement) of the events is directly proportional to the wireless channel utilization factor, since the more channels are busy, the more complex is the calculation of interference and SIR while simulating power regulation.

We initially run this model with three different settings for the channel utilization factor as defined by Equation (2.6), namely 25%, 50% and 75% (that gave rise to average granularity of the simulation events ranging from the order of 30 to 100 microseconds). Also, we considered three different execution modes: a classical sequential execution (relying on a calendar queue scheduler), a parallel execution where no ad-hoc memory management facility is activated, and a parallel execution where we rely on the innovative memory management architecture. Note that the latter execution mode entails switching between

LP-mode and platform-mode (with refill of the CR3 register and implicit squash of the TLB) when changing the actual mode. Hence, such a mode allows us to assess the overhead for mode-switch operated by the support for ECS. In Figure 7.7 we show the variation of the speedup (vs the sequential run) we observed for simulating at least 1 million (committed) events in the different parallel execution modes (each sample is the average over 10 runs based on different random-generation seeds). By the data we see how the maximal loss in performance by the ad-hoc memory management architecture entailing switch between platform and LP modes is on the order of 9% and is observed for finer grain of the simulation events (namely, 25% utilization factor). Such a performance penalty almost disappears when moving to the coarser grain configuration. Also, as expected, the speedup by the parallel runs over the sequential one tend to increase vs the average event granularity.

Successively we modified the PCS model in order to generate fictitious rendez-vous periodically. When one fictitious rendez-vous occurs, the executing LP simply performs a dummy read operation into the state of an adjacent cell. However, we do not really enable ECS synchronization (in fact, in this experiment it does not matter whether the dummy read access is not processed in timestamp order on the hit LP), rather we only trap the access and open the stock associated with the LP hit by the read operation. In this way we are able to assess the overhead by ECS support when also including the management of memory faults and the activation of the ECS handler. For this experiment, we considered the configuration with wireless-channel utilization factor set to 50%, and we varied the frequency of occurrence of the fictitious rendez-vous between 1% and 10% of the total number of events processed. In Figure 7.8 we show the relative speedup achieved by the configuration with ad-hoc memory

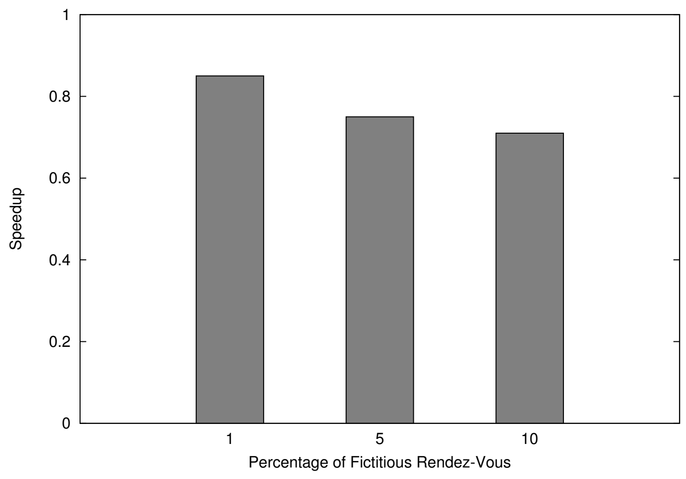


Figure 7.8: Relative Speedup: Memory Management vs Classical Parallel Run

management and fault handling upon the occurrence of fictitious rendez-vous vs the configuration with no ad-hoc memory management. By the data, we see how the ad-hoc architecture induces a speed-down that increases vs the frequency of fictitious rendez-vous. Note however, that the speed-down is not only caused by the overhead for handling the memory faults. Rather it is also due to mode switch between platform and object modes, which is mandatory in order to create the per-thread memory view needed to trap the access to the state of other LPs. On the other hand, the speed-down is quite limited for relatively infrequent fictitious rendez-vous, and becomes non-negligible only when moving towards scenarios with relatively frequent rendez-vous occurrences (say 10%).

On the other hand, the whole ad-hoc memory management architecture has been thought and realized to provide a unique innovative support for handling cross-state dependencies in presence of concurrent LPs transparently to the application code (which, as hinted, enables sequential-style programming on top of PDES platforms). Hence the loss in performance in contexts where the model to be executed exhibits intrinsically disjoint accesses across different LP-states (such as for the configurations in Figure 7.7) is the unavoidable price to be paid for the achievement of a run-time environment offering the above-mentioned level of transparency.

Effectiveness Assessment

For the NoSQL model we consider two different implementations, one not relying on ECS, which explicitly transmits the write set as the payload of the *prepare request* event[6], and another one based on ECS, where such write sets are directly accessed via pointers by the involved simulated nodes (hence the *prepare request* event only needs to carry the pointer, indicating to the target LP where to find in memory the information related to the simulated 2PC phase). This model also entails a special LP which is a global statistics collector. For the case of non-reliance on ECS, the updates of the state of this LP takes place by explicitly scheduling *update-statistics* events towards it. On the other hand, for the case of ECS synchronization, we have that each simulated node can directly access the state of the statistics collector LPs in order to perform updates.

We simulated a NoSQL data-grid system with 64 nodes (with degree of replication 2 of each $\langle key, value\rangle$ pair in the data store), with closed-system configuration in terms of number of clients (and hence number of transactions) running within the system. Particularly, we set the number of active concurrent clients continuously issuing transactions to the value 64. This configuration resembles scenarios where the 64 clients operate as front end servers (co-located with the data-platform nodes) with respect to end-client applications. Also, we set the amount of keys touched in write mode by transactions to 10 and 100, which

[6]For this configuration we have the programmer explicitly coding the marshalling operations of the write set, which impacts both programming and event scheduling costs.

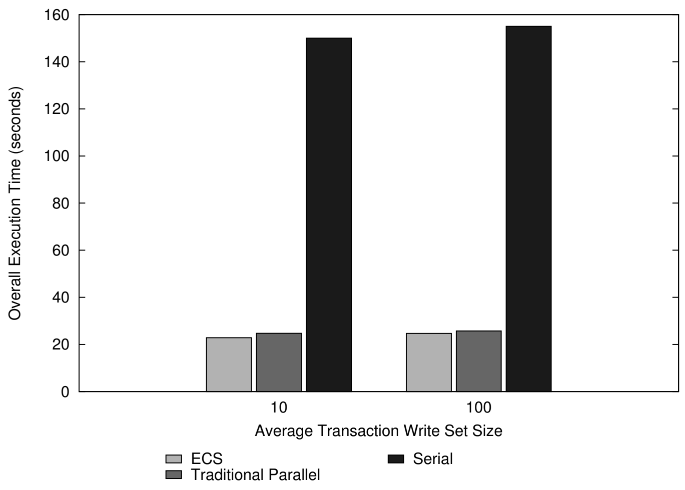


Figure 7.9: Execution times for the NoSQL data store models

gives rise to different dynamics/costs in terms of marshalling operations, message buffering, and transmission for the case of non-ECS based synchronization, thus allowing us to study configurations with different performance tradeoffs.

In Figure 7.9 we report the execution time for simulating a predetermined simulation time interval for the operation of the NoSQL data-grid system. By the data we see how both ECS and the traditional message-passing-based parallel approach provide performance improvements over the sequential run. More important, the performance delivered by ECS is even slightly better than the one by the traditional parallel approach with disjoint accesses to local state of the LPs. Overall, transparency of speculative parallelization with cross-state dependencies is achieved by also delivering performance comparable to the traditional parallel approach, which however does not mask message passing to the programmer, at least in relation to marshalling data in event payloads. Also, the traditional approach does not support direct memory writes into, e.g., the statistic collector LP, thus requiring more complex coding schemes, aimed at realizing the updated via simulation events. On the contrary, with ECS we support such a direct memory update operation, hence offering to the programmer the possibility to code his model more simply, exactly according to sequential programming style.

Chapter 8

Effects on Timeliness and Accuracy

Speed is n subsittute fo accurancy

— Anonymous

After having discussed technical and performance issues related to our solutions, in this Chapter we start reasoning about the effects of the proposed architecture to enhance the degree of transparency that the user experiences.

In particular, by relying on the TCAR agent-based simulation model, we will see how a simulation architecture supporting full transparency has the additional benefit of providing the simulation model writer with more accuracy with respect to traditional PDES systems. We start this discussion by analysing the effects of optimism on simulation results, and the we provide the same results obtained by relying on the Output Management subsystem described in Chapter 5, to show how they better match the output from the sequential execution of the model.

We rely on an agent-based model for this study because they have an intrinsically expressive power, and they constitutes a proven solution to study complex real-world scenarios. In these models, agents exhibit individual or collective interactions, which have been shown to reliably express interactions between different objects/entities in real world phenomena such as disaster rescue [163], computational sociology [102], logistics [81], biomedical applications [101], and economic analysis [120].

They are therefore widely used not only to study steady state or equilibrium properties of a system, rather to determine the exact simulated-time when a given predicate becomes true (which must be done very efficiently as well in case of, e.g., time-critical decision making [83]). High precision in the determination of such a time instant would require frequent inspection of the state of the simulation model (ideally at each state transition, namely after the execution of each simulation event). This may result feasible in traditional sequential simulation, where the unique running thread may quickly retrieve the information for predicate evaluation from the data structures (representing the simulation model state) within its address space. On the other hand, when employing parallel/distributed simulation techniques, such a fine grain inspection along the simulation-time axis may result unviable due to the overhead for process coordination, which may hamper the achievable speedup. Also, for high performance optimistic parallel simulation, where the computation performed might be subject to rollback due to violations event-causality caused by speculative processing, the inspection on the simulation model trajectory may be explicitly delayed to the time instant when a given portion of the computation becomes committed (namely no rollback can even occur in a given simulated-time interval while further executing the simulation model). As a consequence, a shift may appear between the simulated time when the parallel run detects that the predicate holds, and the corresponding simulated time when the sequential run tracks the holding of the same predicate.

8.1 Effects of Optimism on Simulation Results

We have implemented the TCAR model[1] described in Section 2.5.3 in order to be run on top of ROOT-Sim. Interesting for this discussion is ROOT-Sim's peculiar service that, once a new GVT value is available, transparently rebuilds a Committed and Consistent Global Snapshot (CCGS), formed by a collection of individual LPs' states [31]. This occurs via update operations applied to local committed checkpoints of individual LPs so to eliminate mutual dependencies among the final-achieved state values. Once the CCGS is built, each LP gains control via an additional callback within the API, referred to as `OnGVT`, by also having access to the copy of its state image belonging to the CCGS. Such a service can support, e.g., termination detection schemes based on global stable predicates evaluated on a committed and consistent global snapshot. This is a relevant alternative to typical optimistic PDES engines where the run is assumed to be completed only when overstepping a given GVT value.

However, the evaluation of the global predicate on the CCGS has a frequency which is bounded by the frequency of GVT calculation, thus a temporal shift can occur before the simulation application layered on top of the ROOT-Sim platform can track the holding of a global predicate while simulation time advances. This is the core point we address in our study, which is targeted at evaluating the effects of variations of the frequency of GVT calculation (which may impact the speedup of the parallel run, given that it contributes to the overhead for

[1]The source code of our implementation is available at http://www.dis.uniroma1.it/∼hpdcs/ROOT-Sim/tcar.tbz.

distributed coordination) on the estimation of the final time for area-coverage within the TCAR model.

8.1.1 Results

The simulation model has been run until reaching a 100% coverage of the whole region, with a visit factor of 20 (i.e., every cell must be visited at least 20 times before the simulation can complete). This has been done in order to avoid interference in the performance data due to the initial I/O (to access configuration files) and setup operations executed in the early phase of the simulation run. We have set ingress cells for robot unleash to 4, mimicking a situation where the rescue terrain is accessed by a limited number of rescue teams (which can be significant in real disasters). Each rescue team is able to unleash a variable number of ant robots, in the range $[4, 32]$, thus having a number of agents in the simulation in between 8 and 1024. In our implementation, the model is able to simulate a square region (12 Km^2) divided into 4900 hexagonal cells (rather than square cells, as in the original model). According to the original specification of the model [85], each ant robot moves from one cell to another in a time interval of variable length (provided that the destination is reachable, i.e. no obstacle is in between the current and the destination cell). In our configuration of the model, we have made two important choices: i) there are no obstacles in the terrain; ii) time interval is drawn according to an exponential distribution with mean value 100 seconds (corresponding to the typical speed of an ant robot of 50 cm per second), which models the set of variables that can potentially impact the robot move. As for choice i), this allows us to study a *lower bound* of complete region-coverage time. In fact, inserting any obstacle will prevent agents to freely move around, not allowing certain actions, and

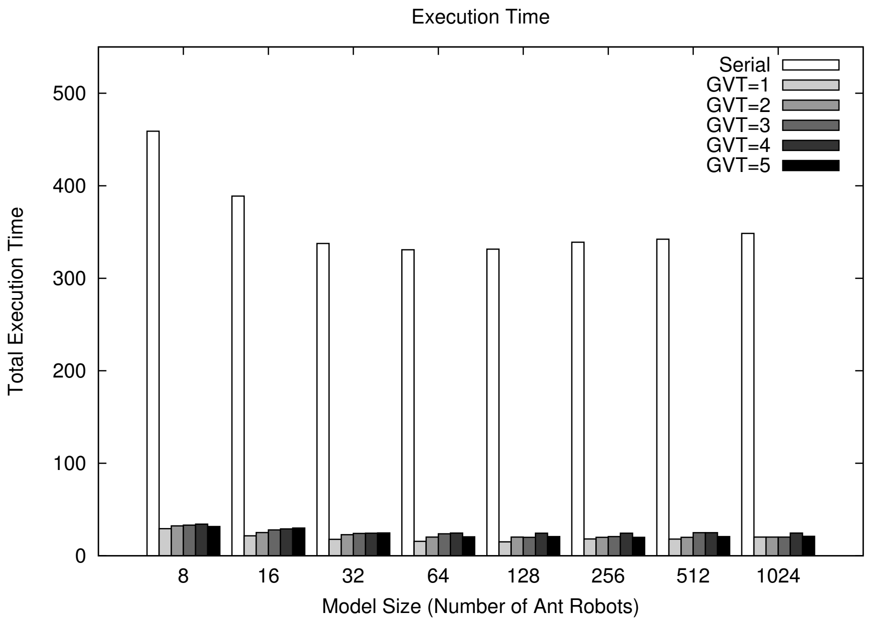


Figure 8.1: Global Execution Times

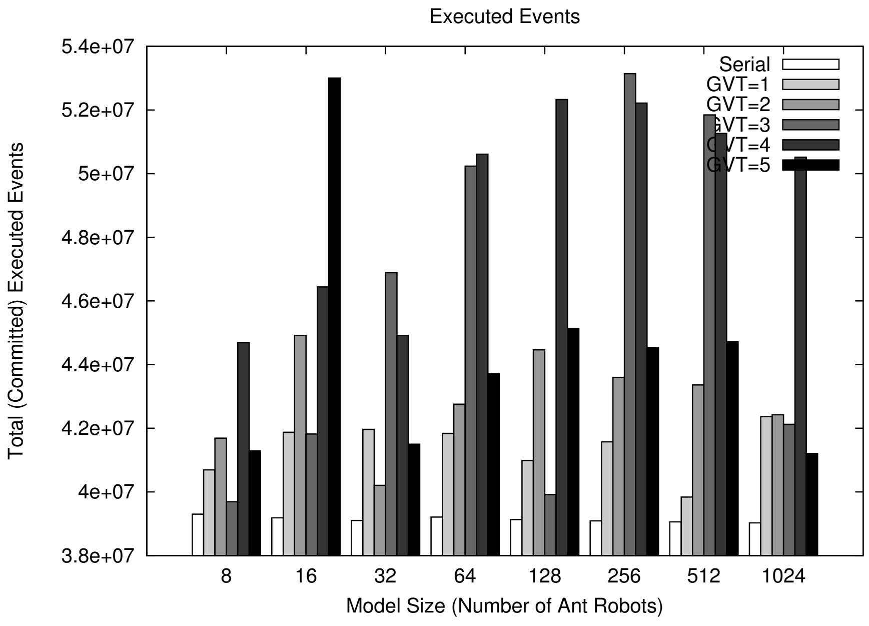


Figure 8.2: Committed Events

increasing the exploration time. Choice ii), instead, allows us to study how much time is required by a group of agents to explore a small area with a very high detail, due to the intrinsic speed limitations (one meter per second [161]).

We have compared the optimistic parallel results with sequential ones, i.e. where events are executed sequentially relying on a competitive scheduler based on Calendar Queues [17]. The GVT computation period in ROOT-Sim has been varied in between 1 and 5 seconds (in wall-clock-time) to check how the confidence interval in the simulation results changes. Each plot is averaged over 20 different runs, both in the parallel and in the sequential case. The initial random seed has been varied across the different runs, in order to account for variations in the execution dynamics, but the set of seeds in the sequential and parallel executions is the same, to really compare similar execution patterns.

In Figure 8.1 we present the global execution times for both the sequential and the parallel execution (on top of ROOT-Sim) of the TCAR simulation model. By the results, we see that the parallel execution provides a speedup in between 15 and 22. Different settings for the GVT computation interval provide a different completion time. In particular, the higher the GVT, the higher the simulation wall-clock-time. This is not related to a loss of precision in the simulation, it is rather due to the way the simulation termination condition is evaluated. As mentioned, the GVT reduction protocol is a periodic computation. If the termination condition is verified immediately after the GVT reduction, then a non-negligible amount of wall-clock time will pass before it will be checked again (leading simulation time to advance). On the other hand, for significantly increased values of the GVT period, the opposite behaviour is noted, since the delayed GVT computation allows to catch the termination condition right after it holds. This situation can be seen as well in Figure 8.2,

Configuration		**Sequential**	**GVT=1**	**GVT=2**	**GVT=3**	**GVT=4**	**GVT=5**
16 Robots	Mean	211.86	216.31	218.27	218.69	234.99	221.81
	Std. Dev.	1.56	15.11	13.28	11.07	12.46	15.64
128 Robots	Mean	26.56	27.37	28.41	28.29	32.61	29.24
	Std. Dev.	0.16	1.08	1.37	3.25	1.83	1.01

Table 8.1: Mean Completion Simulation Time (in Simulated Hours)

where the total number of (committed) events executed by the simulation are reported. As it can be clearly seen, the parallel runs commit a higher number of events than the sequential. Also, the peak values for the parallel runs are not always noted for larger GVT period, just for the reasons explained above.

In order to assess the results' reliability, in Table 8.1 we present the LVT values at which two intermediate configurations of the simulation were stopped. These are the ones with 16 and 128 robots. The reported information is useful to model writers, as it is part of the outcome of the simulation itself. In particular, this tells how much time (in simulated hours) the ant robots would need to cover the entire region. From the data (presented in form of mean time and standard deviation), it can be clearly seen that the sequential simulations offer the most stable result[2]. Interestingly, the parallel executions show a completion simulated time which is always higher that the sequential. This is related, again, to the way the termination condition is checked upon GVT reduction. This result is very important, showing that the outcome of a parallel (optimistic) simulation does not give the most precise result, rather it places a (correct) upper bound on the real values of simulation. Of course, it is up to the simulation model writer to decide how much this divergence from the real simulation results can affect her simulation, and how much benefit she can gain from the increased performance.

[2]We recall that all the results are averaged over 20 different runs, with different initial random seeds.

Configuration		Sequential	Output Management
16 Robots	Mean	211.86	212.01
	Std. Dev.	1.56	1.67
128 Robots	Mean	26.56	26.49
	Std. Dev.	0.16	0.19

Table 8.2: Mean Completion Simulation Time with Output Manager (in Simulated Hours)

8.2 Simulation Completion Detection via Output Management

We have run the same set of experiments relying on the Output Management subsystem presented in Chapter 5. Specifically, by the nature of the subsystem, we still rely on the `OnGVT` callback to check for simulation termination. Yet, concurrently, we have augmented the event handlers' logic to generate an output message (associated with the current LVT value) whenever a LP (i.e., a cell in our simulation model) detects that the its correct coverage has been reached.

In particular, each LP stores in its simulation state a flag that tells whether he has already reported completion time for the simulation on the console output. Upon the execution of each `REGION_IN` event, the simulation model checks whether the coverage condition is met. In the positive case, it checks (by relying on this additional flag) whether the condition was already met in the past or not. In the negative case, it reports on the standard output, by relying on a `printf` call, the ST at which it was met for the first time.

We have collected these values (again, averaged over 20 different runs), and we report in Table 8.2 the maximum one among the cells (which actually tells that the simulation was completed at all the LPs), along with the ones extracted from the sequential run.

We note that the results obtained by relying on the Output Management subsystem are much closer to the sequential ones, and show a confidence interval which allows us to tell that the actual difference is only due to the different initialization random seeds according to which the simulation has been run.

This means that, although the simulation execution still benefits from the execution speedup shown in Figure 8.1, the reliability of the results is comparable to the one shown by the sequential run.

Chapter 9

Effects on Programmability

Frustra fit per plura, quod fieri potest per pauciora.
Pluralitas non est ponenda sine necessitate.
(It is pointless to do with more what can be done with fewer.
Plurality must never be posited without necessity.)
— William of Ockham, Summa Totius Logicæ, ca. 1323

In this Chapter we make use of the PCS benchmark to show what are the implications of the solutions that we have presented in this dissertation on programmability. Specifically, we make a comparison between the PCS implementation to be run on top of ROOT-Sim and the one to be run on top of ROSS[1].

ROSS (the *Rensselaer's Optimistic Simulation System*) is a simulation framework relying on both conservative and optimistic synchronization, the latter being supported by reverse computation. It has been extensively used in literature [111, 54], and recently a multi-threaded version targeted at multicore systems has been proposed [74]. Additionally, it has been proven to scale very well on

[1]We will analyse only some code snippets of the whole simulation models. As of this writing, the full source code for the ROOT-Sim's version is available at `http://svn.dis.uniroma1.it/svn/hpdcs/root_sim/trunk/examples/pcs` [22], while the one for ROSS is available at `https://github.com/carothersc/ROSS/tree/master/ross/models/pcs`

massive multi-core systems, like the IBM BlueGene [7, 9].

As in any simulation model, at simulation startup PCS must initialize its internal state. In ROSS this process is carried out in two steps, the first of which is performed into the `main` entry function, which is coded within the simulation model. The second initialization step, on the contrary, is performed by the `Cell_StartUp` function, which is a particular event handler called for simulation startup. The first step entails model-wide initialization operations, while the second one performs per-LP ones. We present below some data structures and the most important operations related to PCS initialization in ROSS (i.e., we have removed some portions of the code which are just redundant for our discussion):

Listing 9.1: ROSS Initialization

```
struct State { // This is the per−LP simulation state
        double Const_State_1;
        int Const_State_2;
        int Normal_Channels;
        int Reserve_Channels;
        int Portables_In;
        int Portables_Out;
        int Call_Attempts;
        int Channel_Blocks;
        int Busy_Lines;
        int Handoff_Blocks;
        int CellLocationX;
        int CellLocationY;
};

int main(int argc, char **argv) {
        tw_init(&argc, &argv);

        nlp_per_pe = (NUM_CELLS_X * NUM_CELLS_Y) / (tw_nnodes() * g_tw_npe);
        additional_memory_buffers = 2 * g_tw_mblock * g_tw_gvt_interval;
        g_tw_events_per_pe = (nlp_per_pe * (unsigned int)BIG_N) + additional_memory_buffers;
        num_cells_per_kp = (NUM_CELLS_X * NUM_CELLS_Y) / (NUM_VP_X * NUM_VP_Y);
        vp_per_proc = (NUM_VP_X * NUM_VP_Y) / ((tw_nnodes() * g_tw_npe)) ;
        g_vp_per_proc = vp_per_proc;
        g_tw_nlp = nlp_per_pe;
        g_tw_nkp = vp_per_proc;
        g_tw_mapping = CUSTOM;
        g_tw_custom_initial_mapping = &pcs_grid_mapping;
        g_tw_custom_lp_global_to_local_map = &CellMapping_to_lp;

        if( tw_ismaster() ) {
                printf("Print here application configuration\n");
                fflush(stdout);
        }
        tw_define_lps(nlp_per_pe, sizeof(struct Msg_Data), 0);

        tw_run();
        if(tw_ismaster()) {
                CellStatistics_Compute(&TWAppStats);
                CellStatistics_Print(&TWAppStats);
        }
        tw_end();
        return 0;
}

tw_lptype mylps[] = { // NULL-Terminated vector of (per-LP-type) event-handler pointers
    {
      (init_f) Cell_StartUp,
      (event_f) Cell_EventHandler,
      (revent_f) RC_Cell_EventHandler,
      (final_f) CellStatistics_CollectStats,
      (map_f) CellMapping_lp_to_pe,
      sizeof(struct State)
    },
    {0},
};

void Cell_StartUp(struct State *SV, tw_lp * lp) {
        SV->Normal_Channels = MAX_NORMAL_CHANNELS;
        SV->Reserve_Channels = MAX_RESERVE_CHANNELS;
        SV->Portables_In = 0;
        SV->Portables_Out = 0;
        SV->Call_Attempts = 0;
        SV->Channel_Blocks = 0;
        SV->Handoff_Blocks = 0;
        SV->Busy_Lines = 0;
        SV->Handoff_Blocks = 0;
        SV->CellLocationX = lp->id % NUM_CELLS_X;
        SV->CellLocationY = lp->id / NUM_CELLS_X;

        if (SV->CellLocationX >= NUM_CELLS_X || SV->CellLocationY >= NUM_CELLS_Y) {
                tw_error(TW_LOC, "Cell_StartUp: Bad CellLocations %d %d \n", SV->
                    CellLocationX, SV->CellLocationY);
        }
        SV->Portables_In = GenInitPortables(lp);

        for (i = 0; i < SV->Portables_In; i++) {
                TMsg.CompletionCallTS = HUGE_VAL;
                TMsg.MoveCallTS = tw_rand_exponential(lp->rng, MOVE_CALL_MEAN);
                TMsg.NextCallTS = tw_rand_exponential(lp->rng, NEXT_CALL_MEAN);
                switch (Cell_MinTS(&TMsg)) {
                        case COMPLETECALL:
                                tw_error(TW_LOC, "APP_ERROR(StartUp): CompletionCallTS(%lf) 
                                    Is Min \n", TMsg.CompletionCallTS);
                                break;

                        case NEXTCALL:
                                ts = max(0.0, TMsg.NextCallTS - tw_now(lp));
                                CurEvent = tw_event_new(lp->gid, ts, lp);
                                TWMsg = (struct Msg_Data *) tw_event_data(CurEvent);
                                TWMsg->CompletionCallTS = TMsg.CompletionCallTS;
                                TWMsg->MoveCallTS = TMsg.MoveCallTS;
                                TWMsg->NextCallTS = TMsg.NextCallTS;
                                TWMsg->MethodName = NEXTCALL_METHOD;
                                tw_event_send(CurEvent);
                                break;

                    case MOVECALL:
                        // [case removed for brevity]
                }
        }
}
```

We want to highlight some aspects of the above code snippet. First, we can see that the simulation state of the LP is predefined (listing 9.1, lines 1–14). The application model developer must pre-define a contiguous memory buffer which will serve as simulation state for the LP. On the other hand in the ROOT-Sim version initialization code (which is shown below), LP's state is defined as a struct as well, but with a set of pointers which make the simulation state grow or shrink, depending on the actual runtime dynamics of the application (listing 9.2, lines 21–22). In fact, during the execution of the initialization procedure (which in ROOT-Sim is just an event, not a `main` function, listing 9.2, lines 27–28) a couple of malloc calls are issued to shape the initial simulation state (listing 9.2, lines 29, 48). The first one will become automatically the per-LP main state pointer, as described in Chapter 4.

A significant difference arises from the way initialization is called. ROOT-Sim asks the model writer to define the logic for the `INIT` event, which will be then transparently scheduled to all the LPs in the simulation upon start-up. ROSS asks the model writer to explicitly call the `tw_init` function, which initializes the internal simulation kernel. This is a difference which tells how ROOT-Sim is explicitly oriented at the simulation-model writer, which is requested to know almost nothing about the underlying simulation kernel to implement its code, while in ROSS it is the model writer who has to explicitly start-up the kernel.

Then we see (listing 9.1, lines 19–29) that it is the model's duty to setup internal simulation variables which represent, e.g., the mapping between LPs and processing units, the global-to-local mapping, the number of memory buffers to store messages. ROOT-Sim, targeting transparent development of simulation models, does not require the model developer to specify anything about LP mapping or memory management. Instead, LPs to CPU (i.e., worker thread) binding is done transparently, and can be recomputed periodically in order to support load sharing (see [169, 170, 171]). Rather, via the `me` parameter to the `ProcessEvent` callback function, it tells the LP being scheduled its global identifier, while the `n_prc_tot` global variable is set automatically to the total number of LPs in the simulation, in case the model developer needs this information.

Both ROOT-Sim's and ROSS' versions of the PCS model have the need to check whether a specific instance of the kernel is running, to avoid printing multiple copies of the same buffer. ROSS relies on the `tw_ismaster()` function, which tells whether the current instance is the master kernel or not (listing 9.1, line 31). ROOT-Sim does not expose the notion of master kernel (or master thread) to the simulation model writer, which can therefore rely on the more generic `me` id to decide whether to print some configuration or not (listing 9.2, line 42).

An additional call in ROSS is required, namely `tw_define_lps` (listing 9.1, line 35). This function accepts the total number of LPs, the size of messages which will be exchanged across LPs, and a pointer to the seed for this LP (if required). If the seed is not present (like in the above example), then it is set to a default value. In the ROOT-Sim counterpart, there is no need to pre-specify the size of (future) message exchange, as it is able to deal with variable size messages, the size of which is specified upon each schedule operation.

This is explicitly done at listing 9.2, line 54, where each LP sends to itself a

START_CALL event which actually starts the simulation's execution. Since this event has no payload information, the last parameters (payload pointer and size) are NULL and 0, respectively.

ROOT-Sim configuration is now done, and events are already flowing due to the first START_CALL event being injected within the INIT handler. As for ROSS, the structure defined at listing 9.1, lines 46–56 tells (for each LP type) what are the function pointer that must be called for specific actions. Specifically, after the main function explicitly calls tw_run to tell that the simulation must start (listing 9.1, line 37), the init_f function is called for each LP. The function Cell_StartUp (registered as initialization callback), then, completes the initialization of the simulation model.

This function is essentially a specific (initialization) event handler, which accepts a pointer to the LP's state, as the ROOT-Sim's event handler does. After having setup the initial conditions for the cell (as the INIT handler did in the ROOT-Sim version), Cell_StartUp injects GenInitPortables(lp) events in the system. As in ROOT-Sim, the timestamps (compare listing 9.1, line 78 to listing 9.2, line 53) are derived according to an exponential distribution using internal (rollbackable) random number generators, respectively the Expent and tw_rand_exponential functions.

In ROSS, an error condition is handled by a call to tw_error (listing 9.1, line 82), while in ROOT-Sim the same functionality can be implemented by relying on a couple of printf/exit calls, as supported by the I/O Management subsystem (listing 9.2, lines 58–59).

To conclude initialization's description, in ROSS a different action is taken upon cell initialization (on a per-portable basis) depending on the value of the Cell_MinTS, which just tells whether a call will be handed off to a neighbour cell or not. This is to start the simulation in an already steady-state, differently from what we do in ROOT-Sim.

The code for ROOT-Sim's PCS initialization is provided below:

Listing 9.2: ROOT-Sim Initialization

```
typedef struct _sir_data_per_cell{
    double fading;
    double power;
} sir_data_per_cell;

typedef struct _channel{
        int channel_id;
        sir_data_per_cell *sir_data;
        struct _channel *next;
        struct _channel *prev;
} channel;

typedef struct _lp_state_type{
        unsigned int channel_counter;
        unsigned int arriving_calls;
        unsigned int complete_calls;
        unsigned int blocked_on_setup;
        unsigned int blocked_on_handoff;
        unsigned int leaving_handoffs;
        unsigned int arriving_handoffs;
        unsigned int *channel_state;
        channel *channels;
} lp_state_type;

void ProcessEvent(unsigned int me, simtime_t now, int event_type, event_content_type *
      event_content, unsigned int size, void *ptr) {
        [...]
        switch(event_type) {
                case INIT:
                        state = (lp_state_type *)malloc(sizeof(lp_state_type));
                        if (state == NULL){
                                printf("Out of memory!\n");
                exit(EXIT_FAILURE);
            }
            bzero(state, sizeof(lp_state_type));
            ta = TA;
            ta_duration = TA_DURATION;
            ta_change = TA_CHANGE;
            channels_per_cell = CHANNELS_PER_CELL;
            complete_calls = COMPLETE_CALLS;

            // Show current configuration, only once
            if(me == 0) {
                printf("Print configuration here\n");
                fflush(stdout);
            }

            state->channel_counter = channels_per_cell;
            state->channel_state = malloc(sizeof(unsigned int) * 2 * (
                CHANNELS_PER_CELL / BITS + 1));
            for (w = 0; w < state->channel_counter / (sizeof(int) * 8) + 1; w++) {
                state->channel_state[w] = 0;
            }

            timestamp = (simtime_t)(Expent(ta));
            ScheduleNewEvent(me, timestamp, START_CALL, NULL, 0);
        break;

        default:
            fprintf(stderr, "PCS: Unknown event type! (me = %d - event type = %d)\n",
                me, event_type);
            exit(EXIT_FAILURE);
    }
}
```

Both ROOT-Sim and ROSS rely on event handlers for managing the logic associated with events. As mentioned, in ROOT-Sim they are implemented as cases of the only application callback, namely ProcessEvent. In ROSS, multiple callback functions are defined, among which the `event_f` one implements the forward execution of events, and `revent_f` one implements the reverse execution (we recall that ROSS supports rollback by reverse computation). In this comparison, we will only discuss the code for starting a new call in a cell. Below, the ROSS code for supporting forward and backwards execution of a call start-up is provided:

Listing 9.3: ROSS Call Startup

```
void Cell_NextCall(struct State *SV, tw_bf * CV, struct Msg_Data *M, tw_lp * lp) {
        int done, dest_index = 0;
        int currentcell = 0, newcell = 0;
        tw_stime ts;
        struct Msg_Data TMsg;
        struct Msg_Data *TWMsg;
        tw_event *CurEvent;
        double result;

        TMsg.MethodName = M->MethodName;
        TMsg.ChannelType = M->ChannelType;
        TMsg.CompletionCallTS = M->CompletionCallTS;
        TMsg.NextCallTS = M->NextCallTS;
        TMsg.MoveCallTS = M->MoveCallTS;

        SV->Call_Attempts++;

        if ((CV->c1 = NORM_CH_BUSY)) {
                SV->Channel_Blocks++;
                result = tw_rand_exponential(lp->rng, NEXT_CALL_MEAN);
                TMsg.NextCallTS += result;

                switch (Cell_MinTS(&TMsg)) {
                        case COMPLETECALL:
                                tw_error(TW_LOC, "APP_ERROR(NextCall):␣CompletionCallTS(%lf)␣
                                      Is␣Min␣\n", TMsg.CompletionCallTS);
                                break;
```

```

                              case NEXTCALL:
                                        // [case removed for brevity]

                              case MOVECALL:
                                        newcell = lp−>gid;
                                        while (TMsg.MoveCallTS < TMsg.NextCallTS) {
                                                  M−>RC.wl1++;
                                                  currentcell = newcell;
                                                  dest_index = tw_rand_integer(lp−>rng, 0, 3);
                                                  newcell = Cell_ComputeMove( currentcell, dest_index ); //
                                                         Neighbours[currentcell][dest_ index];
                                                  result = tw_rand_exponential(lp−>rng, MOVE_CALL_MEAN)
                                                         ;
                                                  TMsg.MoveCallTS += result;
                                        }

                                        ts = max(0.0, TMsg.NextCallTS − tw_now(lp));
                                        CurEvent = tw_event_new((currentcell), ts, lp);
                                        TWMsg = (struct Msg_Data ∗)tw_event_data(CurEvent);
                                        TWMsg−>MethodName = TMsg.MethodName;
                                        TWMsg−>ChannelType = TMsg.ChannelType;
                                        TWMsg−>CompletionCallTS = TMsg.CompletionCallTS;
                                        TWMsg−>NextCallTS = TMsg.NextCallTS;
                                        TWMsg−>MoveCallTS = TMsg.MoveCallTS;
                                        TWMsg−>MethodName = NEXTCALL_METHOD;
                                        tw_event_send(CurEvent);
                                        break;
                    }
     } else {
          // [branch removed for brevity]
     }
}

void RC_Cell_NextCall(struct State ∗SV, tw_bf ∗ CV, struct Msg_Data ∗M, tw_lp ∗ lp) {
          int i;
          SV−>Call_Attempts−−;
          if (CV−>c1) {
```

```
63                SV->Channel_Blocks--;
64                tw_rand_reverse_unif(lp->rng);
65                for (i = 0; i < M->RC.wl1; i++) {
66                        tw_rand_reverse_unif(lp->rng);
67                        tw_rand_reverse_unif(lp->rng);
68                }
69        } else {
70                // [branch removed for brevity]
71        }
72 }
```

For the sake of brevity, we will discuss only the essential differences from the ROOT-Sim's version of the model, which will be shown later in this Chapter.

It is interesting to note that in the `if` statement (listing 9.3, line 18)) there is an assignment as well. In this example, `CV->cl` is a field of a `tw_bf`, a *bit field*, which (as discussed earlier in Section 2.3.1) allows to support reverse computation (as it is indeed done on listing 9.3, line 62) by telling which branch was taken. ROOT-Sim on the other hand does not rely on reverse computation, and therefore there is no need to manipulate platform-related data structures during the execution of any event.

On listing 9.3, lines 33–40, there is a `while` loop. Within this loop, the counter `M->RC.wl1` is increased upon each iteration. It is interesting to note that `M` is the message data, so the event handler is directly modifying data stored into the event queue of the simulation kernel. This counter is increased to allow reverse computation of the loop (listing 9.3, line 65), by allowing an exact number of reverse iterations to be executed.

It is interesting to note that, as explained in Section 2.3.1, random-library calls must be rolled back as well. Since ROOT-Sim is based on state save & restore, this is transparently done upon a state restore (by restoring a previous random seed). In case of reverse computation, this must be done explicitly in the reverse event, by calling reverse version of the random generator (`tw_rand_reverse_unif(lp->rng)`, listing 9.3, line 64, 66–67), exposing as well (in this example) the internal random seed to the simulation model developer.

In the ROOT-Sim case, upon call start-up the function `allocation` is called (listing 9.4, line 7), which takes care of performing the initial power allocation and SIR regulation. It is interesting to note that (at line 45) new memory buffers are allocated on an uncommitted portion of the simulation trajectory (and then linked to the main simulation state, at line 52), relying on the facilities offered by Di-DyMeLoR.

Event generation in ROSS is supported by a couple of calls, i.e. `tw_event_new` to generate a new event buffer, and `tw_event_send` to actually schedule it (listing 9.3, lines 43, 51). In ROOT-Sim an event can be just any structure, thus the API function `ScheduleNewEvent` (listing 9.4, line 15) simply accepts a pointer to any memory buffer and the size of the new event being inserted in the system.

Listing 9.4: ROOT-Sim Call Startup

```
1  case START_CALL:
2          state->arriving_calls++;
3          if (state->channel_counter == 0) {
4                  state->blocked_on_setup++;
5          } else {
6                  state->channel_counter--;
7                  new_event_content.channel = allocation(state);
8                  new_event_content.from = me;
9                  new_event_content.sent_at = now;
10
11                 new_event_content.call_term_time = now+(simtime_t)(Expent(state->ta_duration));
12                 handoff_time = now + (simtime_t)(Expent(state->ta_change));
13
14                 if(new_event_content.call_term_time <= handoff_time) {
15                         ScheduleNewEvent(me, new_event_content.call_term_time, END_CALL, &
                                   new_event_content, sizeof(new_event_content));
```

```
            } else {
                new_event_content.cell = FindReceiver(TOPOLOGY_HEXAGON);
                ScheduleNewEvent(me, handoff_time, HANDOFF_LEAVE, &
                    new_event_content, sizeof(new_event_content));
            }
        }

        timestamp = now + (simtime_t)(Expent(state->ta));
        ScheduleNewEvent(me, timestamp, START_CALL, NULL, 0);

break;

/*******************/

int allocation(lp_state_type *pointer) {
        int i;
        int index = -1;
        double summ;
        channel *c, *ch;

        for(i = 0; i < pointer->channels_per_cell; i++){
            if(!CHECK_CHANNEL(pointer,i)){
                index = i;
                break;
            }
        }

        if(index != -1) {
            SET_CHANNEL(pointer, index);

            c = (channel*)malloc(sizeof(channel));
            if(c == NULL){
                printf("malloc error: unable to allocate channel!\n");
                exit(EXIT_FAILURE);
            }

            c->next = NULL;
            c->prev = pointer->channels;
```

```
                c->marked = false;
                c->channel_id = index;
                c->sir_data = (sir_data_per_cell*)malloc(sizeof(sir_data_per_cell));
                if(c->sir_data == NULL){
                        fprintf(stderr, "malloc error: unable to allocate SIR data!\n");
                        exit(EXIT_FAILURE);
                }

                if(pointer->channels != NULL)
                        pointer->channels->next = c;
                pointer->channels = c;

                // [recompute SIR and fading here]
        } else {
                fprintf(stderr, "Unable to allocate channel\n");
                exit(EXIT_FAILURE);
        }
        return index;
}
```

By the above examples, it is clear that the programming model supported by ROOT-Sim is definitely more targeted at the simulation model programmer. In fact, all the above ROOT-Sim examples are completely written in ANSI-C, and do not manipulate any internal platform data structure. Additionally, in the event handlers there is no notion of parallelism, thus allowing a fully sequential programming style. These were both clear goals of this dissertation.

Additionally, ROSS requires the user to manually write reverse events (although effort is being put in their automatic generation [65, 66, 173]). This inevitably harasses transparency, as the model writer must be aware of the synchronization scheme which is supporting her model's execution, and must manually change the values of some internal parameters.

The proposed code for ROOT-Sim, on the other hand, can be executed with any DES simulation kernel which respect the main-loop specification from Algorithm 2.1, thus showing backwards compatibility with any traditional DES model, which can be therefore deployed on a concurrent environment without substantial modification.

Chapter 10

Conclusions and Future Work

Now this is not the end. It is not even the beginning of the end.
But it is, perhaps, the end of the beginning.
— Winston Churchill, The End of the Beginning, 1942

In this dissertation we have tackled the issue of transparency in PDES simulation kernels supported by optimistic synchronization. In our voyage, we have always tried to finely determine what is the best tradeoff between the degree of transparency that the simulation model writer can see, and the overall performance of the simulation.

At the end of this work, the user can rely on several innovative methodologies and runtime supports which allow her to rely on the standard ANSI-C programming language to implement any simulation model based on the event-driven programming paradigm, which is proper of the traditional DES paradigm.

In particular, by the results in Chapter 4, the user is able to transparently rely on malloc calls—invoked in an uncommitted portion of the simulation trajectory—to make the per-LP simulation states grow or shrink on demand, depending on the current execution dynamics. In turn, by the innovative services provided, which are based on transparent (compile-time) static software instrumentation, the simulation kernel is able to determine (with no intervention by the user) what are the portions of the simulation state(s) which have been updated since the last log operation. In this way, the simulation kernel is able to take incremental snapshots, which have been proven by the experimental results to provide a significant performance benefit in scenarios where the simulation states are large, but scarcely updated. Additionally, the instrumentation mechanisms required for supporting transparent incremental state saving have been shown to be of negligible cost in scenarios where the simulations states are very large.

The second breakthrough towards transparency in simulation model development has been made in Chapter 5, where an innovative runtime support based on the notion of an *output daemon* (i.e., a process separated from the main simulation kernel, which communicates with it via shared memory) has been presented. The advantages provided by the output daemon are numerous. First, it allows the model writer to issue output commands within event handlers, i.e. during a portion of the simulation's execution which is not committed. Nevertheless, the output is materialized only when the associated event gets committed, thus avoiding disruptive (i.e., non-rollbackable) effects on the outside world. Second, it provides a (geographical-scale) ordering on the produced output, which is non-trivial (as it would otherwise require an additional stage of post processing) and allows the technique to be used even with conservative synchronization approaches. Finally, it is based on autonomic runtime monitoring which tries to balance the delay experienced by the user in output commitment with respect to the overall simulation throughput, avoiding (not-wanted) performance degradation.

In Chapters 6 and 7 we have dealt with shared state access in simulation models, i.e., we have shown how it is possible to drop constraints expressed by Equation (2.5), which were the foundation upon which traditional PDES was built. In particular, Chapter 6 has shown us a runtime support, based on binary instrumentation, which allows the user to access in a concurrent way global variables, by relying on an innovative set of non-blocking algorithms. By the results, we have shown that using multi-version lists reduces the incidence of rollback operations when read/write operations are not really causally dependent. Along with a performance study, a correctness proof has been provided as well.

On the other hand, in Chapter 7, we have proposed a twofold innovation. In the beginning, we have presented an innovative memory-management system, targeted at Linux operating systems, which allows, by relying on a loadable kernel module, to have multiple threads (the simulation-kernel worker threads, in our case) share the same portions of the memory page directory with different access privileges. This has allowed us, in the second part of the Chapter, to build an innovative synchronization scheme, which is able to transparently materialize cross-LP state dependencies. Via the exploitation of the rollback operation, this synchronization scheme allows to align two LPs simulation trajectories whenever one tries to access (in either read or write mode) the private simulation state of the other. Both a correctness and a progress proof have been shown, together with a performance evaluation.

Each of the proposed solutions has been implemented within the ROOT-Sim open source optimistic synchronization platform, and a performance evaluation has been presented. In Chapters 8 and 9 we have shown which are the impacts on model development of our solutions, highlighting how a negligible overhead provides the end user with an improved programming experience, allowing her to write purely sequential-style code (according to the event-driven programming paradigm proper of traditional DES) which is in turn run concurrently in an optimized fashion on top of the ROOT-Sim simulator.

The unavoidable question is: *where do we go now*? The results presented in this thesis pave the way to further research directions, which can be followed in the near future by interested researchers to improve even more the experience of programmers, in order to help distributing the power of parallel computing architectures to the masses.

Many of the proposed solutions work on a parallel environment, but cannot be instantly moved to distributed environments. For example, dealing with global variables' access on a distributed architecture inevitably requires some sort of coordination among the nodes, which cannot abstract from a distributed consensus algorithm, based on message passing. Developing such a strategy would allow the immediate deployment of a sequential program on top of a network-interconnected architecture.

Dealing with input is another key issue. Interactive simulation can provide significant benefits to the end user, as it would allow her to see how a system would behave upon different (sudden) solicitations. In the context of optimistic synchronization, where the fossil collection operation actually "destroys" the notion of the remote past, this can be a challenging problem. Techniques which keep some simulation snapshots even before the last GVT value could be a viable solution to facing this issue.

Most programmers rely on many third party libraries. In our solutions, only stdlib has been addressed. Given the incredible (ever-changing) vastness of libraries out there, some automatic approach should be adopted to cope with them. Possible solutions could entail relying on dynamic techniques like the ones which are the ground of the ltrace tool, which is nevertheless limited in the number of arguments which can be handled when the original headers are not available. An analysis of the assembly call graph could be useful to determine, by the registers/stack regions used before calling a library function which (and what) are the parameters passed, although this solution might be prone to errors in case of only few calls are issued to a library function. Source code could be analysed as well, although this solution is less portable (in fact, although targeting C code, the ones that we have proposed, since they rely on static binary analysis, could be re-adapted to Fortran, C++, or other programming languages, which have already been used to implement a plethora of simulation models).

In the context of cross-state accesses, an interesting enhancement could entail making a distinction between read-generated and write-generated cross-state dependencies. In fact, in case of a read-state dependency, the target LP could be prevented from rollbacking. Rather, its state could be artificially reconstructed at that specific simulation time, thus avoiding to waste a (possibly correct) portion of the performed work.

An additional aspect which is interesting to study is what happens when the simulation model writer is actually *aware* of the parallel architecture that is lying under his software. For example, what does it means for a programmer to call, e.g., a `pthread_create()` function in his code? Technically, it would create an additional worker thread, which the platform is not aware of, and would therefore likely cause the whole simulation to crash abruptly. So, calls to pthread-family functions should be handled explicitly. Yet, creating a new thread, does it mean that the user is willing to create a new LP or that she is trying to parallelize some activities within one (or multiple) events? Is she creating a parallel execution trajectory with respect the current LP? A semantic investigation should be required, to effectively define what this awareness would entail from a simulation perspective.

The most interesting (and complex) extension to this work entails re-adapting the proposed solutions to any kind of code, not only simulation one. This could be done by dropping the “constraint” that the user has to implement events. In fact, respecting the event-driven programming paradigm is the only requirement to the developer to have the proposed solutions fully working. If there were the possibility to automatically detect what is an “event” in any program, than this would extend the proposed solution to any software, thus achieving full automatic code parallelization. Although in principle this is a very complex extension, preliminary ideas tells us that it could be either done by (costly) static analysis of the binary program, or speculatively at runtime by trying to define “on demand” *transactions* on the code. Or, of course, a mixture of the two approaches.

In any case, this thesis has opened the way to many new improvements at the bleeding edge of compile-time/runtime supports to application development which, in the future, will eventually allow any inexperienced programmer to fully benefit from the incredible computing power which the industry has provided us with, in the recent years.

Appendix A

Hijacker

Inde consultat ex duobus itineribus breve et solitum
sequatur an inpeditius et intemptatum
(He then deliberated whether he should pursue the short and
ordinary route, or that which was more difficult and unexplored)
— Cornelius Tacitus, The Annals

Hijacker [123] is a binary instrumentation technique which we have developed as a support tool for our research, and which has been extensively used throughout this thesis.

Binary Instrumentation is a technique which modifies a binary program by inserting additional instructions or by changing existing ones at compile time (*static instrumentation*) or at runtime (*dynamic instrumentation*), in order to observe or modify the execution's flow, without altering the overall program's semantic. This technique has been successfully exploited in several fields, such as behaviour monitoring [36], performance analysis [150], attack detection [114], or to alter code for supporting transactional memories [118]. Instrumentation tools like Pin [130], Dyninst [38], Valgrind [167], and DynamoRIO [37] have been extensively used in order to analyse various applications and observe their behaviour.

In static instrumentation, the executable is analysed at a certain stage during its compiling process (i.e., before or after the final linking stage), while in dynamic instrumentation some logic is inserted into the application so that, when the process is launched, the control is taken by a runtime module which alters the code during the actual program's execution. While the latter could be regarded as a more powerful technique, due to the fact that at runtime more information is available to the runtime disassembling module (e.g., any conditional branch can be successfully evaluated), the former technique allows for the creation of more efficient executables, due to the fact that only the strictly-necessary instrumentation code is inserted in the application binary. Considering that we explicitly address High Performance Computing scenarios, Hijacker necessarily falls in the former category, although, as it will be discussed later, few additional runtime tasks are performed.

Static binary instrumentation has nevertheless some disadvantages. In fact, the instrumentation process cannot target third-party libraries, especially shared libraries. In fact, if a static instrumentation tool were to target such libraries, the resulting instrumented library would affect every executable in the system which is relying on it. Considering that instrumentation could significantly affect the behaviour of the code (depending on the user needs), this approach would be non-viable. On the other hand, Hijacker offers a set of tools to redirect specific (third-party libraries) functions calls to user-defined stubs, or to efficiently wrap specific ones, thus giving the user the freedom to wisely use third-party code under controlled execution flow.

The process of instrumenting an executable poses two challenges: on the one hand, since instrumentation works on machine-level code, this process is intrinsically instruction-set-dependent. On the other hand, in order for the user to correctly modify the application's flow (without altering its semantic), she has to manually provide the tool with the additional code to be injected, which can be a non-trivial task, since it may depend on the actual compiler, architecture and calling conventions specified by the current ABI. In order to hide this complexity away, we have specifically designed Hijacker in order to provide three main features:

- the instrumentation process is *rule-driven*, i.e. the operations performed by Hijacker are specified via an xml file, which instructs the tool on the specific tasks to be performed in the process;
- the most common tasks can be performed *transparently*, since Hijacker comes bundled with a set of *instrumentation features* (e.g., target memory address reconstruction) which can be inserted into the original binary;
- the process of instrumenting the code is instruction-set and executable independent, i.e. Hijacker performs its tasks on an internal binary representation, decoupling the specific details of the underlying architectures and therefore allowing the tool to instrument the same original high-level code compiled for different architectures.

There are several works in literature and several tools which address the problem of instrumentation. The earliest implementations of binary instrumentation toolkits are ATOM [158] and EEL [40]. They both instrument code at compile time, avoiding as much as possible runtime overhead. ATOM is targeted at alpha machines only, while EEL tries to hide the complexities of the underlying instruction sets providing abstract C++ interfaces for altering the code. On the contrary, our tool drives the instrumentation process by relying on rules provided in a xml file.

BIRD [113] is a binary rewriting platform for Windows/x86 only. This tool basically relies on the insertion of a 5-byte branch instruction in order to give control to instrumentation code, or relies on interrupts when 5 bytes are not available. This technique is similar to the one presented by PEBIL [90], although the latter tool is targeted at Linux boxes and uses function relocation allowing the tool to rely on the 5-byte branch at any instrumentation point. The main difference from our proposal is that we completely rebuild the final executable, and thus there are no limits in the amount of code which can be inserted in-place (i.e., without relying on calls to other portions of the executable)

Dyninst [38] is a tool for static and dynamic instrumentation. It can either create a modified version of the binary at compile time, or can operate at runtime. The most notable feature of this tool is the ability to perform liveness analysis on registers' values and on flag register's bits.

Some tools like Pin [130] rely on just-in-time instrumentation. In particular, Pin can be seen as a middleware which places itself under the original application, and at applications interrupt points it instruments the upcoming parts of the original code. Just-in-time instrumentation is used as well by other tools like DynamoRIO [37] and Valgrind [167]. The latter tool offers a very large set of functionalities, ranging from memory-management errors detection to cache utilization analysis. Nevertheless, while incredibly useful, techniques used by this tools produce a heavyweight overhead, making it not suitable for deployed HPC applications.

A.1 Design and Implementation

In the compiling process, Hijacker lays just before the final linking stage. In fact, we have explicitly decided to work on a relocatable representation of the executable because we can rely on the additional linking metadata in order to perform our application analysis and build our internal binary representation.

Hijacker's architecture is divided in a front-end module—which provides several compatibility layers with different executable formats and assembly languages, and is able to parse xml rule files—and a back-end module—which performs the actual instrumentation operations on an intermediate (machine-independent) representation of the executable. Hijacker is open source[1], and is available as one of the tools released by the HPDCS research group[2]. The overall architecture is depicted in Figure A.1.

A.1.1 Rule Specification

As mentioned before, Hijacker is a rule-based instrumentation tool. To support the instrumentation process and to leverage the user from many technical details, we allow her to instruct Hijacker via a simple xml file, an example of which is provided in Figure A.2. As it can be seen, the user can instruct Hijacker to perform several actions on the executable on a per-function basis or on a global-basis.

In particular, the configuration file gives the freedom to insert manually-written portions of code (by relying on the `<Inject>` tag). The code should be written in the target machine's assembly language, or in a higher language for which a compiler is available on the machine, and is automatically compiled

[1]http://www.dis.uniroma1.it/∼pellegrini/?p=hijacker
[2]http://www.dis.uniroma1.it/∼hpdcs

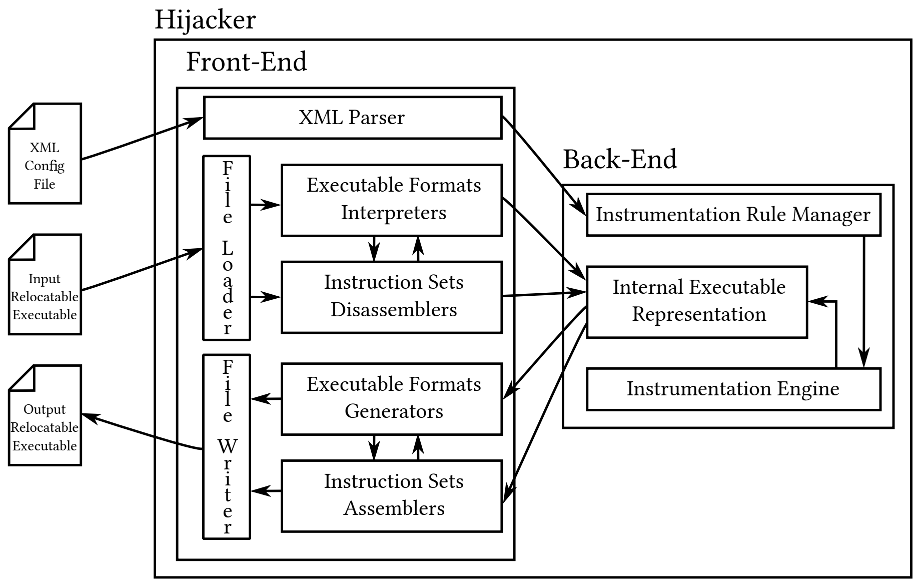

Hijacker
Front-End
XML Parser
XML Config File
Input Relocatable Executable
Output Relocatable Executable
File Loader
File Writer
Executable Formats Interpreters
Instruction Sets Disassemblers
Executable Formats Generators
Instruction Sets Assemblers
Back-End
Instrumentation Rule Manager
Internal Executable Representation
Instrumentation Engine

Figure A.1: General Hijacker's Architecture

```
<HijackerRules>
  <Inject file="memorycopy.c"/>
  <Function name="foo">
    <Instruction instruction="I_MEMWR|I_MEMRD" injectBefore="memcount.S">
     <AddCall where="before" function="monitor" arguments="target" convention="stack"/>
    </Instruction>
  </Function>
  <Instruction instruction="I_JUMP|I_CONDITIONAL" injectAfter="jumpcount.S">
    <AddCall where="before" function="monitor" arguments="register" convention="stdcall"/>
  </Instruction>
  <Instruction instruction="movs" replace="">
    <AddCall where="after" function="memcopy" arguments="target" convention="registers"/>
  </Instruction>
</HijackerRules>
```

Figure A.2: Example configuration file

by Hijacker. Instructions to be altered are specified using the `<Instruction>` tag, which supports several attributes: `instruction`, specifies either a single (target-architecture) assembly instruction, or a (machine-independent) family of instructions, as it will be clearly discussed in Section A.1.2; `injectBefore`, `injectAfter`, and `replace` specify either an assembly code file or a specific instruction to be placed (respectively) before, after or in place of the related instruction. The `<AddCall>` tag allows the user to insert specific calls to original or injected functions in the code. Several attributes are allowed: `where` determines whether the call is placed before or after the target instruction; `function` specifies which function must be called; `arguments` specifies which arguments should be passed to the callee (as it will be explained in Section A.1.4); `convention` determines if the arguments are passed either by stack or by registers (respecting the target architecture's calling convention). The user can specify which operations should be performed on a specific function by enclosing the relevant tags in the `<Function>` tag, where the function name is specified in the `name` attribute. We note that if `<AddCall>` is used within a `<Function>` tag, the meaning of the `where` attribute specifies whether the function call is performed after the function specified in `<Function>` is called (i.e., the call is injected in the function's code) or before any call to it in the executable.

When Hijacker is launched, the front-end's XML parser module loads the configuration file and instructs the back-end's instrumentation rules manager about the operations which will be performed during the instrumentation process.

A.1.2 Application Analysis and Internal Binary Representation

In order to perform the instrumentation tasks, the original (relocatable) executable must be processed first. The front-end's file loader module performs a sequence of tests on the executable in order to determine which executable format is used to represent the program, and triggers the corresponding executable format interpreter (among the ones registered at Hijacker's compile time) in order to start loading the program. The first step undertaken by the executable interpreter is to check which assembly language is used to represent instructions in the executable, and this information is reported back to the file loader manager, which searches among the available disassembler engines for one able to interpret the code. If a suitable disassembler is found, this information is reported to the executable interpreter.

This modular approach allows any combination of assembly languages and executable formats for the representation of a program, and allows for high extendibility of the tool, decoupling the process of adding supports for new/additional formats and languages from the instrumentation process itself. At the time of this writing, Hijacker is bundled with an ELF format interpreter, an x86 disassembler and an x86_64 disassembler, while a PE interpreter and an ARM disassembler are under development.

The executable interpreter then starts analysing the program and builds an intermediate representation of it which we refer to as *program map*. This program map is structured in sections, whose type can be code, data, or raw. The latter section describes any type of section which is not involved in the instrumentation process, and is therefore straight recreated in the altered program. The data section keeps track of global data used by the program, in terms of name of the variables (if any), their size in byte (if available), and their initial value.

The code section actually contains an intermediate representation of the instructions. In particular, the executable interpreter gives control to the suitable disassembler in order to start a linear scan of the assembly code. Each assembly instruction gets stored into a data structure that keeps the original bytes of the instruction, along with several attributes describing the instruction itself. Each data structure keeps a set of pointers to target adjacent instructions in the function, data (if any), and/or other instructions (if any, in case of branches or function calls). Depending on the actual assembly language used by the program (and on the compiler which generated the code), this process can present more or less difficulties. As an example, some compilers emit data within the code in order to support efficient execution of, e.g., indirect branches deriving from the compilation of switch cases. The disassembler module is able to communicate with the executable interpreter if, during the instrumentation process, some data segments are discovered within the instructions[3]. In this case, the executable interpreter adds to the data section the additional data structures, enforcing the logical separation between data and code used to build the program map. Of course, this action slightly changes the "shape" of the altered program which will be produced as output of the instrumentation process, but does not change its operational flow.

During the linear scan of the assembly code, Hijacker's disassemblers cross-check the information retrieved from symbol and relocation tables from the original relocatable program. In this way, the disassembler is able to decode

[3]The methodologies used by disassembler modules for discovering such portions of data are different, depending on the architecture and its ABI, and discussing them is out of the scope of this appendix. Nevertheless, to give the reader an idea, they mostly rely on decompilation error-retry algorithms, coupled with address/register-value evaluation.

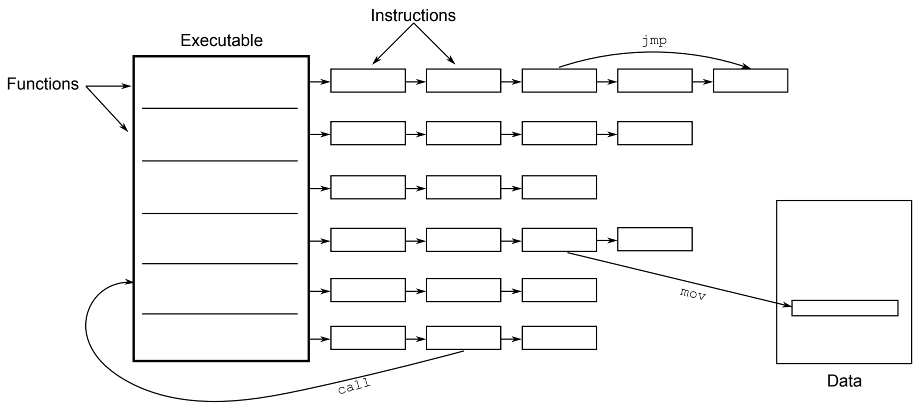


Figure A.3: Hijackers Internal Representation of Executables

instructions and organize their internal representation in functions, preserving the connection among instructions and the data, in order to produce an intermediate representation of code as depicted in Figure A.3. We note that the connections between instruction-instruction (due to branches in the code), instruction-function (due to function calls), and instruction-data (due to data movement) are realized in the intermediate representation as memory pointers, rather than offsets as most assembly languages do. Considering that functions are represented as linked lists of instructions (the elements' order in such lists is their appearance in the code, which does not necessarily correspond to the program's execution flow), inserting or removing any instruction at any point of code does not alter the linking between objects in the intermediate representation.

Whenever an instruction is interpreted by the disassembler, it gets marked using special flag values which describe the actual *family* of the instruction. The available flags and their meanings are:

`I_MEMRD`: The instruction reads from memory

`I_MEMWR`: The instruction writes to memory

`I_CTRL`: The instruction performs checks on data

`I_JUMP`: The instruction alters the execution flow

`I_CALL`: The instruction calls a different function

`I_RET`: The instruction returns from a callee

`I_CONDITIONAL`: The instruction is executed only if a condition is met

`I_STRING`: The instruction operates on large amount of data

`I_ALU`: The instruction does some logical/arithmetic operation

`I_FPU`: The instruction does some floating-point operation

`I_STACK`: The instruction works on stack

`I_INDIRECT`: The instruction behaviour might depend on some runtime value

or any or'ed combination of them. For example, an instruction marked as `I_MEMWR|I_MEMRD` reads from and writes to memory, while an instruction marked as `I_JUMP|I_CONDITIONAL` is an indirect branch. These combinations of flags can be used in the `instruction` attribute described in Section A.1.1 to specify instrumentation rules for groups (families) of instructions which perform a certain action.

A.1.3 Code and Data Instrumentation

Once the program map is completely built, the execution control is given to Hijacker's back-end, in particular to the instrumentation rules manager. This module starts applying the rules parsed from the xml configuration file, by triggering (for each rule) the corresponding operation in the instrumentation engine.

The instrumentation engine operates on the internal intermediate representation of the executable. Whenever some rule requires a modification in a particular function, the corresponding entry is found in the functions array and the instructions' data structures are linearly scanned in order to identify the ones which must be instrumented. If the rule involves adding some instructions before or after the target one, some new nodes are simply inserted in the list of instructions. For the generation of machine-level code, the instrumentation engine asks the instruction-set assembler to produce the machine-level representation of the instruction. If the insertion of code entails the compilation of some assembly file, the default compiler installed on the host machine (`cc`) is invoked automatically. When Hijacker is launched, a custom compiler can be specified, which will override this setting.

As hinted before, whenever some rule is applied, the connections between instructions, functions and data are preserved since they are realized as memory pointers between structures. This is true even in the most complex cases: if instructions are referred from the data section (e.g., in the aforementioned branch table case) then the approach described in Section A.1.2 had notified executable interpreter about the presence of data segments within the code. If the disassembler module was able, at runtime, to discover the presence of references to instructions, then they are replaced with memory pointers targeting the destination instructions. This approach has been shown to cover most of the code generated by standard compilers. Nevertheless, there are some cases which involve the generation of instruction addresses completely at runtime. If such a case is found, then Hijacker is not currently able to correctly instrument the executable, although we are working on making the technique discussed in [168] more efficient and suitable for HPC.

A.1.4 Bundled Instrumentation Features

During the instrumentation process, disassemblers populate the data structures used for representing instructions in the intermediate representation with all the information which can be gathered from their binary representation during the process. This information can be stored in the executable and therefore used at runtime by, e.g., user-injected functions. This is exactly the goal of the `arguments` attribute described in Section A.1.1, and the one of the *bundled instrumentation features*: provide the end user with some cached disassembly information in order to allow some sort of dynamic monitoring without having to rely on any kind of dynamic instrumentation.

At the time of this writing, Hijacker provides the end user with two bundled instrumentation features, namely *target address reconstruction* and *indirect register detection*. The former is a feature which allows, whenever an `I_MEMRD` or `I_MEMWR` instruction is being instrumented, to insert some additional code which evaluates by software the destination address of such instruction, and the size of the access. This information can be passed as argument of any (user-injected or not) function in the binary.

The branch register instrumentation feature is similar in spirit to target address reconstruction. This feature allows any function in the code to be called passing two arguments: an integer code representing the register which is used to execute an indirect memory access (either for reading/writing or for altering the control flow with an indirect branch), and its actual runtime value. This can be particularly useful to trace at runtime the actual execution patterns of an application, when this information cannot be detected at compile time.

A.1.5 Binary Multi-versioning

Hijacker can make multiple (differently-instrumented) versions of the same executable coexist in the same image, in case more `<Executable>` tags in the xml configuration file are found. This facility has been successfully exploited in [168] to create differently-instrumented versions of the same original application-level program which coexist and the execution flow is passed from one version to the other.

In order to support such a scenario, Hijacker allows to specify a new entry point for the program. This facility can be used to insert, e.g., some logic in the instrumented application which dynamically selects one of the two versions of the application binary depending on the current execution dynamics. This approach can be regarded as a means for efficiently creating executables which, according to the autonomic paradigm [64], are able to efficiently react to changes in the execution dynamics by, e.g., selecting a different operating mode. The different operating modes could be realized as differently-instrumented versions of the same executable, sparing the programmer the implementation of slightly different versions of the same program, a process which is inherently long and error-prone.

From a technological point of view, whenever two or more versions of the executable must be created, the rule manager asks the instrumentation engine to create multiple copies of the program map as the first action. Each copy will have its internal symbols renamed adding a progressive number, so that if the user wants to inject code which calls either instrumented version, she will be able to do so consistently. The data section is not duplicated, so to allow a consistent sharing of data among the versions. Each program map version will be then instrumented applying the related rules.

A.1.6 Binary Recreation

The last step in the instrumentation process is the recreation of a (relocatable) executable which can be later passed to the linker to complete the compiling process. When the rules manager has finished applying modifications to the program map(s), the control is returned to Hijacker's front-end.

The proper executable formats generator is thus triggered in order to recreate a new executable on disk. This process entails repeatedly accessing the program map(s) in order to build all the data structures which are required by the format itself. As it can be clearly seen, this process is highly format-dependent, and the description of the operations involved would be in the scope of specifically-targeted papers. Nevertheless, to give the reader an idea, they are similar in what actual compilers do whenever they are generating a program from a set of source files.

A.1.7 Third-party libraries

The possibility to rely on third-party libraries depends on the actual behaviour of used functions. In fact, as an example, if the user is instrumenting the executable to track memory updates and then relies, e.g., on standard `memset` library function, then memory updates performed by `memset` would not be tracked.

As mentioned before, there are several solutions to this problem, each one having different drawbacks. In order to provide efficient solutions to this issue, we have explicitly avoided to rely on costly runtime analysis approaches, while on the contrary we have given the user the possibility to wrap third-party library calls, by relying on a specific `<Library>` tag. In this way, the user can specify which are the external library calls which she wants to wrap, and therefore any call in the executable to them will be actually redirected to user-specified functions, in a way similar to what standard linkers allow to do during the compilation process.

A.2 Experimental Evaluation

In order to evaluate the instrumentation overhead induced by the proposed architecture, we have conducted experiments on a family of configurations of PCS ROOT-Sim. Each cell sustains the same workload of incoming calls, whose interarrival time is exponentially distributed, and whose average duration is set to 2 minutes. The call interarrival frequency to each cell has been varied in the interval between 1 and 6.25 calls per simulation time unit, thus providing increasing values of the channel utilization factor, as of Equation (2.6), in between 12% and 75%, and hence increasing values of the expected length of the PCS list of in-use records (we recall the complete description in Section 2.5.3. The residence time of an active device within a cell has a mean value of 5 minutes and follows the exponential distribution. This has the effect of performing an increasing number of memory updates whenever the climatic model starts scanning the allocated channels for recomputing the optimal power allocation values. For the above scenario, we have run experiments with 1024 wireless cells, modelled as hexagons covering a square region, each one managing 1000 wireless channels.

In Figure A.4 we present the results for the various PCS configurations (i.e., interarrival frequencies) in two cases: the first represents the execution of PCS instrumented using Hijacker, where every memory-write access (i.e., `I_MEMWR` instructions) have been instrumented by placing before each of them a call to a hand-written module which relies on the target address reconstruction bundled instrumentation feature for creating a memory access map (similarly to what has been done in Chapter 4); the second represents an execution of the original

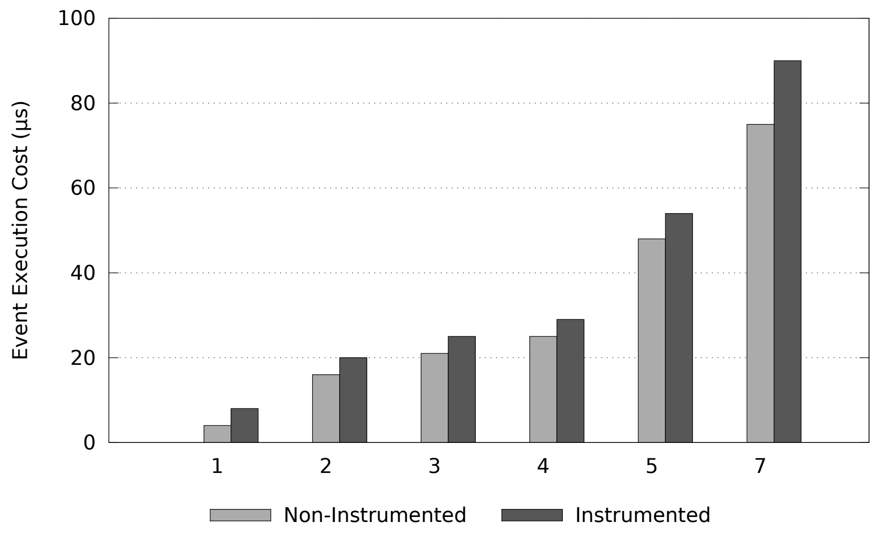


Figure A.4: Overheads Associated with Different Workloads

benchmark, with no instrumentation. The instrumented scenario is non trivial, since it additionally requires the execution of a runtime module to compute the actual memory addresses. Figure A.4 shows average execution time of PCS events in both cases. As it can be clearly seen, the overhead added by the modifications of the code by Hijacker are negligible, while by relying on the instrumented code the user is able to reduce actual management costs by using incremental logging facilities like the ones described in Chapter 4.

Appendix B
ROOT-Sim

"A slow sort of country!" said the Queen.
"Now, here, you see, it takes all the running you can do, to keep in the same place.
If you want to get somewhere else, you must run at least twice as fast as that!"
— Lewis Carroll, Through the Looking-Glass, 1871

ROOT-Sim [67] is a PDES simulation kernel relying on the optimistic synchronization paradigm. It comes as a static library which can be linked to executables implementing simulation models using the ANSI-C programming standard [84], as if they were completely sequential.

In particular, the user can organize the code in as many functions/files as needed, can perform any I/O operation during the simulation (keeping in mind that I/O operations can degrade performance) as described in Chapter 5, can use dynamically-allocated memory to build the simulation state as described in Chapter 4, and so on and so forth. The only exception is that the `volatile` qualifier becomes meaningless (i.e., there is no possibility for a variable to be modified outside the simulation platform). In addition, the model lives in user space only, so no *NIX system calls should be used. No regular entry point is required for the application-level code (i.e., no `main()` function must be implemented), as entry points for the application code are specified by ad-hoc APIs, which will be discussed in the next section.

The actual simulation is based on events: each LP processes events, and its advancement in the LVT is connected to their execution. LPs communicate via messages, via global variables (see Chapter 6), or via direct pointer passing (see Chapter 7). ROOT-Sim, as the Time Warp protocol stands, enforces a logical identity between events and messages. This means that a message envelopes an event to be scheduled to some destination LP. Each event (and thus message backing it) is identified by a numerical code, which is defined by the application-level logic. Each type of message is fixed-size, but more than one type of message can be used. In particular, the application-level code must provide a definition of a `struct` for each event type, where the content of a message (an event) must be specified. Furthermore, events must be of known types, i.e., each event must be associated with a unique ID number which must be specified whenever a message is sent.

Each LP has its own execution context and its own stack. They are both implemented as user-level threads, with a practical implementation similar to the one proposed in [42]. Each stack lives in the LP's stock (as described in Chapter 7), so we enforce a complete separation between the simulation-kernel-level and the LP-level data structures. Context switches are executed relying on POSIX `setjmp`/`longjmp` API functions.

Each logical process has its data structures enclosed in a `LP_struct` data structure (resembling what the Linux kernel does for processes), thus enforcing modularity end easiness to extension. All the execution context information is maintained into this structure.

Concurrent execution is based on the notion of worker thread. At simulation start-up a certain number of worker thread is spawned, so that they can take control of some (or all) the computing resources available in the underlying architecture. Therefore, a single machine in the distributed simulation context can host one or more simulation kernel instances. LPs are evenly distributed across simulation kernel instances at simulation start-up, adopting either a *block policy* or a *round policy*. Nevertheless, LPs are *bound* to worker threads on a periodic basis. Specifically, according to the proposals in [169, 170, 171], the worker threads periodically redistribute the (locally hosted) LPs in order to maximize the overall performance by adopting a load sharing policy.

At simulation start-up, ROOT-Sim delivers to each LP a special `INIT` event (identified by the reserved code 0) which allows the simulation model to set up its initial configuration. In particular, the model can define its simulation state relying on a sequence of `malloc()` calls, and the initial values can be retrieved by command-line arguments which are delivered as `INIT`-event's payload, resembling the standard ANSI-C `char **argv` vector. During the execution of the `INIT` event, other events can be scheduled at any LP in the system, therefore allowing the actual simulation to start. According to the ROOT-Sim programming model, the first `malloc()` call issued by each LP during the execution of the `INIT` event is considered as the initial part of the LP's simulation state, and will be later passed via a specific pointer to allow the execution of additional events. This can be overridden by a specific API provided by the platform, as it will be discussed later. Nevertheless, this approach allows LPs' states to arbitrarily grow/shrink during the simulation's execution, just relying on additional `malloc()`/`free()` calls. This unique feature means that the user can produce standard code to design the simulation model, with the only requirement that every additional `malloc`'d memory region must be referred via a pointer in the

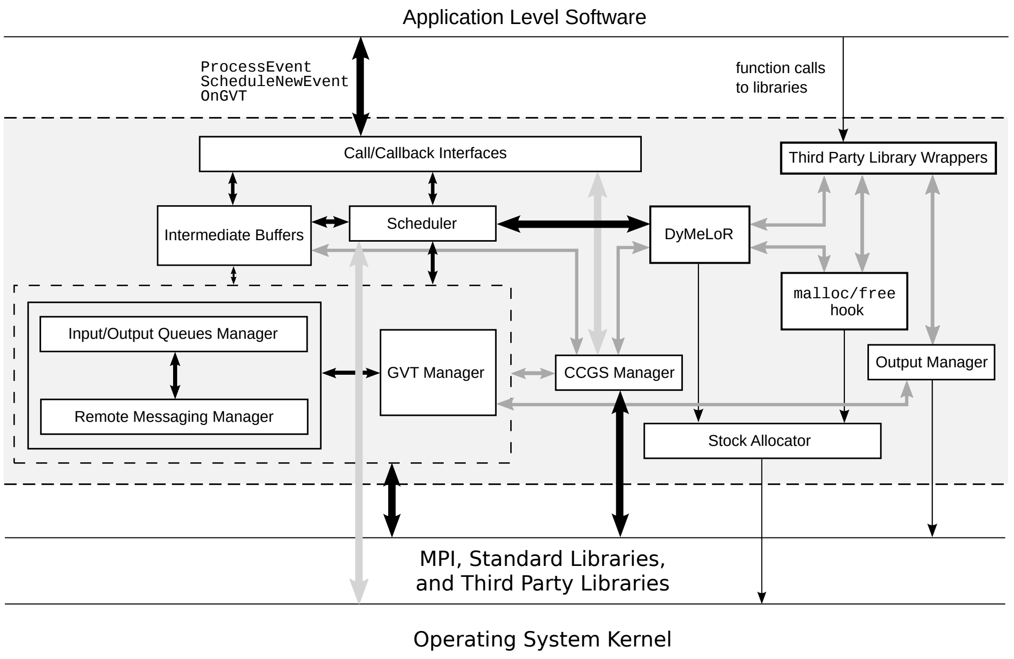


Figure B.1: ROOT-Sim Architecture

first `malloc`'d structure for the simulation platform to be able to correctly rollback previous simulation states.

ROOT-Sim is open source[1], and is available as one of the tools released by the HPDCS research group[2].

The general architecture of the current version of ROOT-Sim is shown in Figure B.1.

B.1 Supported APIs

The core API to allow communication between application-level code and simulation kernel is very simple. It consists of one call function, `ScheduleNewEvent()`,

[1]http://www.dis.uniroma1.it/∼pellegrini/?p=hijacker
[2]http://www.dis.uniroma1.it/∼hpdcs

and two callback functions, `ProcessEvent()` and `OnGVT()`. The callbacks must be necessarily implemented in the simulation model to be compliant with the library. Then, the rest of the code can be implemented in any way, albeit respecting the ANSI-C standard. These functions have the following signature/purpose.

`void ProcessEvent(int me, time_type now, int event_type, void *event_content, void *state)` is the callback that supports the actual processing of simulation events, and it is used by the kernel to give control to the application layer. `me` is the ID of the LP being scheduled, `now` is the current value for the local clock, `event_type` is the numerical code for the event to be processed, `event_content` is the information regarding the event itself, and `state` is the current LP's state. Inside of `ProcessEvent()` the execution is fully speculative, i.e. the events that are executed might be eventually undone. The programmer, nevertheless, is completely unaware of this issue, and can simply implement state transitions within this callback. ROOT-Sim will transparently undo (in case of a detected inconsistency) or commit speculative events (whenever a new GVT value is computed and the commitment horizon is moved forward). The only issue concerning `ProcessEvent()` is the execution of non-rollbackable actions. In fact, if the programmer, e.g., prints some text on the screen during the execution of an event that will be eventually rolled back, the output generated will not be reverted. This is a non-trivial problem associated with speculative execution, even more if transparency is enforced and the programmer is given the freedom to implement its model by relying on standard ANSI-C. ROOT-Sim offers a facility which tries to address this issue (at the cost of some delay in the materialization of the actual output), which will be presented in Section B.2.

`void ScheduleNewEvent(int receiver, time_type timestamp, int event_type, void *event_content, int event_size)` is a function that allows injecting a new simulation event within the system, to be destined to whichever simulation object. `receiver` denotes the ID of the destination LP, `timestamp` is the LVT associated with the event to be processed, `event_type`, `event_content`, and `event_size` allow the correct identification and delivery of the actual event. For efficiency reasons, the invocation of this function does not immediately involve the actual deliver of the associated event to the destination LP. Instead, events are buffered and asynchronously delivered when the execution of the current one is completed. This allows to pack together more events if the destination LP is the same, and prevents delays in the current event's execution. We note that this asynchronous deliver does not affect the correctness of the execution, as ROOT-Sim will order events in the input queue before scheduling the next event to the destination LP. In case the delay created by this internal buffering generates an out-of-order execution at some LP, then the rollback procedure will restore consistency.

`bool OnGVT(void *snapshot, int gid)` is a callback that gives control to the application layer by also providing a committed snapshot of the simulation object. The execution of `OnGVT()` is therefore not speculative, i.e. any action taken within this function will never be undone. This means that, e.g., any I/O operation within this function is perfectly safe, and therefore it can be used to gather statistics on the ongoing simulation, if the user is aware of the synchronization strategy and does not want to rely on the facilities described in Chapter 5. We note that, since the timestamp associated with `snapshot` refers to the committed portion of the computation, it is forbidden to call `ScheduleNewEvent()` within `OnGVT()`, because this might induce a rollback operation of already committed events. In case the user calls `ScheduleNewEvent()` in this callback, a runtime error will be generated. `OnGVT()` additionally implements a distributed termination control: since `snapshot` is a portion S_i of the Committed and Consistent Global State (CCGS) S, according to [104] a global predicate can be locally evaluated on S_i. If the model determines that the simulation is completed for that particular LP, `OnGVT()` can return the `true` value. ROOT-Sim will collect all return values, and in case all the LPs agree, the simulation will stop.

In addition to the core API, ROOT-Sim has a set of additional facilities. First, `void SetState(void *new_state)` allows the LP to manually specify which is its simulation state. It is not mandatory in a simulation-model's implementation, as explained in Section B.3, but gives the programmer more freedom.

ROOT-Sim comes bundled with a fully-featured numerical library, which gives the modeller the possibility to generate random numbers drawn from several distributions. So far, the `Random()`, `Expent()` (exponential), `Normal()`, `Gamma()`, `Poisson()`, and `Zipf()` distributions are implemented. This library is crucial in the development of simulation models (and should be used in place of other libraries) because it adheres to the Piece Wise Deterministic paradigm (see [41]), i.e. the same sequence of numbers will be deterministically produced if a rollback operation is performed. The numerical library maintains one seed per each LP, pseudo-randomly drawn by an initial master seed which can be either randomly computed at simulation start-up, or manually specified by the user. This gives full control on the model's execution, giving the possibility to re-study the same configuration (determined by the initial master seed) which will give the same final outcome, independently of the actual events' execution pattern.

In addition, ROOT-Sim offers the library function `int FindReceiver(int topology)` to draw an LP's neighbour according to some geometric topology uniformly at random. Valid values for `topology` are `RING`, `BIDRING`, `LINEAR`, `HEXAGON`, `SQUARE`, `STAR`, `MESH`, which describe respectively a ring, a bidirectional ring, a linear vector, a square region divided in several hexagonal cells, a square region divided in several square cells, a star topology, and a mesh. This library function can be used to simplify the implementation of communication among LPs, and the returned id can be directly used as a parameter of `ScheduleNewEvent()`.

B.2 Internal Features

The simulation platform offers several facilities to support model's simulation in the most effective way, and to control the simulation's execution. Configuration can be specified either at command line, or by relying on an interactive shell which supports execution scripts as well. The main simulation loop (concurrently executed by each worker thread) is shown in Algorithm 2.8.

The GVT computation is carried out according to the shared-memory based proposal [50]. Its interval can be tuned at simulation start-up. If the user selects a higher value, the overhead associated with this operation gets reduced, but since the fossil collection operation (see Chapter 1) is thus executed less often, the overall simulation likely requires more memory to maintain, e.g., older checkpoints. By tweaking this parameter, the user can manually balance the tradeoff between performance and memory consumption.

As mentioned before, at simulation start-up ROOT-Sim initializes the internal numerical library starting from a master seed. The user can ask ROOT-Sim to randomly draw this master seed or can specify it manually. In case the seed

Algorithm 2.8 Implementation of ROOT-Sim's Simulation Loop

```
procedure Simulation-Loop
    while $End$ == false do
        Receive remote messages
        Process Bottom Halves
        if GVT interval has passed then
            start GVT computation
        end if
        $e_{next} \leftarrow$ events associated with timestamp $T_{min}$ among the bound LPs
        if  then$e_{next}$ is a straggler
            rollback LP in charge of processing $e_{next}$ to $T_{e_{next}}$
        else
            switch context to LP in charge of processing $e_{next}$
        end if
        if CCGS tells simulation is complete then
            $End \leftarrow$ true
        end if
        process outgoing messages
        if GVT computation complete then
            execute Fossil Collection
        end if
    end while
end procedure
```

is randomly chosen, the user can configure ROOT-Sim to store it as the master seed for future runs. In this way, a set of experiments can be conducted on the same events' pattern, therefore comparing the very same model's configuration.

Concerning initialization, ROOT-Sim allows to run the simulation model on a distributed environment, by ultimately relying on MPI for message delivery. Using a description of the machines which can be used to run the simulation (i.e. address for reaching them, and number of CPU-cores), ROOT-Sim transparently sets up the distributed simulation environment. For efficiency reasons, two different mappings between LPs and simulation kernels are available: *block distribution* evenly divides the available LPs across the kernel instances, while *circular* assigns one LP per kernel, in a circular fashion. Depending on the inter-LP interaction, the one that limits at most the number of inter-kernel interactions due to event exchange can be selected, so to reduce the impact on performance.

As for state saving, ROOT-Sim offers a large set of facilities which address both transparency and execution efficiency at the same time. In particular, as mentioned, the simulation state can be scattered across different segments of allocated memory, and log/restore operations can be carried out either in non-incremental [166] or incremental way (see Chapter 4). Additionally, ROOT-Sim can be configured to switch between these two differentiated modes autonomically and to optimize the checkpointing interval, either by relying on an analytic model [168], or by relying on a genetic algorithm [126].

ROOT-Sim offers the user the possibility to select either the traditional LTF scheduler for event processing selection, or an optimized $O(1)$ variant [147], which selects the next event to be scheduled in a (probabilistic) constant time.

Third-party libraries are almost fully supported. The user is allowed to rely on any third-party library if it is stateless. Support for stateful third-party libraries, as stated in Chapter 10, is subject of future work.

A branch of the ROOT-Sim kernel supports migration of LPs among different kernel instances depending on the current workload [128].

B.3 A Code Example

We present here some code snippets implementing a ROOT-Sim application which models a set of N nodes connected as a mesh, sending packets randomly to each other. The first important thing is to define the possible events handled by the model, the content of an event message, and the structure of the state:

```
#include <ROOT−Sim.h>
#define PACKET 1 // Event definition
#define DELAY 120
#define PACKETS 1000000 // Termination condition

typedef struct _event_content_t {
    time_type sent_at;
} event_content_t;
typedef struct _lp_state_t{
    int packet_count;
} lp_state_t;
```

In this model we allow just one application-defined event, `PACKET`, which identifies the transit of a packet in the mesh. Then, we must specify the actual events' logic. `ProcessEvent()` is the only entry point for speculative event processing, so we rely on a `switch` construct to demultiplex them:

```
void ProcessEvent(unsigned int me, time_type now, unsigned int event, event_t *content,\\
                        unsigned int size, lp_state_t *ptr) {
    event_t new_event;
    time_type timestamp;

    switch(event) {

        case INIT: // must be ALWAYS implemented
            state = (lp_state_t *)malloc(sizeof(lp_state_t));
            state−>packet_count = 0;
            timestamp = (time_type)(20 * Random());
            ScheduleNewEvent(me, timestamp, PACKET, NULL, 0);
            break;

        case PACKET: {
            pointer−>packet_count++;
            new_event_content.sent_at = now;
            int recv = FindReceiver(MESH);
            timestamp = now + Expent(DELAY);
            ScheduleNewEvent(recv, timestamp, PACKET, &new_event, sizeof(new_event));
        }
    }
}
```

The code logic is fairly simple: upon `INIT` event, the LP's state is `malloc`'d and initialized, and an initial packet is sent to the LP itself. Whenever a `PACKET` event is received, a local counter is increased, and a packet is sent back to a random LP in the simulation environment. Timestamps are computed according to an exponential distribution, exploiting the internal `Expent()` function.

`OnGVT()` is the second callback to be implemented, which performs a local check on the LP's state. If the number of packets passed through the LP is smaller than `PACKETS`, then the simulation cannot be halted yet:

```
bool OnGVT(lp_state_t *snapshot, int gid) {
    if (snapshot->packet_count < PACKETS)
        return false;
    return true;
}
```

B.4 Runtime Data

Beyond the statistics that the user can programmatically obtain from the execution of the `OnGVT()` callback, ROOT-Sim has three levels of statistics generation which can be configured at start-up. The first (default) level produces a file containing average values for the most common performance metrics on a system-wide scale. An example file is presented in Figure B.2. With it, the user can gather information about general execution and overall performance, i.e. the efficiency of the simulation, or the number of events executed.

If the user is interested in the same statistics but at a per-kernel granularity

```
TOTAL KERNELS............... : 32
TOTAL PROCESSES............. : 1024
TOTAL EVENTS EXECUTED........ : 169923760
TOTAL COMMITTED EVENTS....... : 96932768
TOTAL ROLLBACKS EXECUTED..... : 2357476
TOTAL ANTIMESSAGES.......... : 72208424
AVERAGE ROLLBACK FREQUENCY... : 1.4 %
AVERAGE ROLLBACK LENGTH...... : 30.62 events
EFFICIENCY.................. : 58.11 %
AVERAGE EVENT COST.......... : 11.48 us
AVERAGE CHECKPOINT COST...... : 28.327 us
AVERAGE RECOVERY COST........ : 20.110 us
Simulation started at........ : Thu Mar 21 23:23:07 2013
Simulation finished at....... : Thu Mar 21 23:27:49 2013
Computation time............ : 282 seconds
Last GVT value.............. : 50828.568372
Average Committed Event-Rate. : 343733.22 events/second
```

Figure B.2: General Statistics Output File

(i.e. not a system-wide average), the second level of statistics generates a text file like the previous one for each kernel instance. This is useful, e.g., to check if the workload of the model is evenly distributed across simulation kernel instances. At the same time, this level produces punctual data during the simulation's execution in the form ⟨GVT phase number, GVT value, committed events in this phase, cumulated committed events⟩ (one tuple per line). This adds some overhead, which is nevertheless minimal because this operation is performed periodically only during the GVT calculation. This is a very useful information to track performance variations during the execution of the simulation model.

The last (more intrusive) level of statistics generation prints on a per-kernel separate file additional information for each GVT phase, in the form ⟨total events, committed events, rollbacks, average event cost, average checkpoint cost⟩.

Bibliography

[1] A. Adya and B. Liskov. Lazy consistency using loosely synchronized clocks. In *PODC*, pages 73–82, 1997. (cited on pages 190, 191.)

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS, pages 483–485. ACM, 1967. (cited on page 8.)

[3] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 1994 Real-Time Systems Symposium*, pages 172–181. IEEE Computer Society, Dec 1994. (cited on page 9.)

[4] R. Ayani. A parallel simulation scheme based on the distance between objects. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 21, pages 113–118, 1989. (cited on page 41.)

[5] H. Aydt, S. J. Turner, W. Cai, and M. Y. H. Low. Research issues in symbiotic simulation. In *Proceedings of the 2009 Winter Simulation Conference*, WSC, pages 1213–1222. Society for Computer Simulation, Dec. 2009. (cited on page 11.)

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, May 2000. (cited on page 102.)

[7] P. D. Barnes, Jr, C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp speed: Executing time warp on 1,966,080 cores. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS, pages 327–336. ACM, 2013. (cited on page 240.)

[8] D. W. Bauer, G. Yaun, C. D. Carothers, M. Yuksel, and S. Kalyanaraman. Seven-o'clock: A new distributed gvt algorithm using network atomic operations. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation*, pages 39–48. IEEE Comp. Soc., 2005. (cited on pages 48, 50.)

[9] 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, pages 35–44. IEEE Computer Society, 2009. (cited on page 240.)

[10] 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*. IEEE Computer Society, Mar 1993. (cited on pages 86, 87, 88, 90.)

[11] G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *Computer*, 39(1):110–112, Jan. 2006. (cited on page 7.)

[12] S. Bellenot. Global virtual time algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 122–127, Jan. 1990. (cited on page 48.)

[13] S. Bellenot. State skipping performance with the Time Warp operating system. In *Proceedings of the 6^{th} Workshop on Parallel and Distributed Simulation*, PADS, pages 53–64. Society for Computer Simulation International, 1992. (cited on pages 79, 90.)

[14] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., 1986. (cited on pages 190, 191.)

[15] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991. (cited on page 9.)

[16] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. (cited on page 149.)

[17] R. Brown. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31:1220–1227, October 1988. (cited on pages 30, 142, 148, 149, 168, 234.)

[18] D. Bruce. The treatment of state in optimistic systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 40–49. IEEE Comp. Soc., June 1995. (cited on page 95.)

[19] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical Report MIT/LCS/TR-188, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977. (cited on pages 39, 40.)

[20] J. Burns and N. A. Lynch. Mutual exclusion using invisible reads and writes. In *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, 1980. (cited on page 187.)

[21] W. Cai and S. J. Turner. An algorithm for distributed discrte-event simulation—the "carrier null message" approach. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 3–8. Society for Computer Simulation International, Jan. 1990. (cited on page 41.)

[22] 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*, pages 53–60. IEEE Comp. Soc., May 2000. (cited on pages 56, 239.)

[23] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, july 1999. (cited on page 50.)

[24] R. P. Case and A. Padegs. Architecture of the IBM system/370. *Communications of the ACM*, 21:73–96, Jan. 1978. (cited on pages 99, 183.)

[25] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE–S5(5):440–452, Sept. 1979. (cited on pages 33, 39, 40, 55.)

[26] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–206, Apr. 1981. (cited on page 39.)

[27] K. M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*. Society for Computer Simulation International, Mar. 1989. (cited on page 41.)

[28] K. M. Chandy and R. Sherman. Space-time and simulation. Technical Report No. 238, University of Southern California, Information Sciences Institute, 1989. (cited on page 96.)

[29] 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, pages 1–14. USENIX Association, 1999. (cited on page 43.)

[30] L.-l. Chen, Y.-s. Lu, Y.-P. Yao, S.-l. Peng, and L.-d. Wu. A well-balanced time warp system on multi-core environments. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 1–9. IEEE Comp. Soc., 2011. (cited on pages 36, 97.)

[31] D. Cucuzzo, S. D'Alessio, F. Quaglia, and P. Romano. A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. *Distributed Simulation and Real Time Applications, IEEE/ACM International Symposium on*, 0:227–234, 2007. (cited on page 231.)

[32] S. R. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: a Time Warp system for shared memory multiprocessors. In *WSC '94: Proceedings of the 26th conference on Winter simulation*, pages 1332–1339. Society for Computer Simulation International, 1994. (cited on pages 56, 94.)

[33] R. H. Dennard, F. H. Gaensslen, Y. Hwa-Nien, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct. 1974. (cited on page 6.)

[34] P. Di Sanzo, F. Antonacci, B. Ciciani, R. Palmieri, A. Pellegrini, S. Peluso, F. Quaglia, D. Rughetti, and R. Vitali. A framework for high performance simulation of transactional data grid platforms. In *Proceedings of the 6th ICST*

Conference of Simulation Tools and Techniques, SIMUTools. ICST, Mar. 2013. (cited on pages 73, 75, 198.)

[35] 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*, SIGSIM-PADS, pages 103–114. ACM, 2013. (cited on page 30.)

[36] W. Drewry and T. Ormandy. Flayer: exposing application internals. In *Proceedings of the first USENIX workshop on Offensive Technologies*, WOOT, pages 1:1–1:9. USENIX Association, 2007. (cited on page 261.)

[37] DynamoRIO. http://www.dynamorio.org/. (cited on pages 261, 264.)

[38] DynInst. http://www.dyninst.org/. (cited on pages 261, 264.)

[39] W. B. Easton. Process synchronization without long-term interlock. *SIGOPS Operating Systems Review*, 6(1/2):95–100, June 1972. (cited on pages 94, 98.)

[40] EEL. http://pages.cs.wisc.edu/∼larus/eel.html. (cited on page 263.)

[41] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, sept 2002. (cited on pages 81, 91, 92, 137, 285.)

[42] R. S. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC, pages 20–20. USENIX Association, 2000. (cited on page 280.)

[43] A. Fabbri and L. Donatiello. Sqtw: a mechanism for state-dependent parallel simulation. description and experimental study. In *Parallel and Distributed Simulation, 1997., Proceedings., 11th Workshop on*, pages 82–89. IEEE Computer Society, Jun 1997. (cited on page 96.)

[44] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985. (cited on pages 95, 99.)

[45] J. Fleischmann and P. A. Wilsey. Comparative analysis of periodic state saving techniques in time warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 50–58. IEEE Comp. Soc., June 1995. (cited on pages 83, 90.)

[46] F. S. Foundation. GDB: The GNU Project Debugger. `http://www.gnu.org/software/gdb/`. (cited on page 102.)

[47] C. Fu, D. Wen, X. Wang, and X. Yang. Hardware transactional memory: A high performance parallel programming model. *Journal of Systems Architecture*, 56(8):384–391, Aug. 2010. (cited on page 11.)

[48] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990. (cited on pages 33, 39.)

[49] R. M. Fujimoto. Feature article–parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 5(3):213–230, 1993. (cited on page 173.)

[50] R. M. Fujimoto and M. Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 7(4):425–446, Oct. 1997. (cited on pages 48, 286.)

[51] R. M. Fujimoto and D. M. Nicol. State of the art in parallel simulation. In *Proceedings of the 24th conference on Winter simulation*, WSC, pages 246–254. ACM Press, 1992. (cited on page 40.)

[52] K. Ghosh and R. M. Fujimoto. Parallel discrete event simulation using space-time memory. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 201–208. CRC Press Inc., 1991. (cited on pages 96, 97.)

[53] J. B. Gilmer, Jr. An assessment of "Time Warp" parallel discrete event simulation algorithm performance. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 45–49. Society for Computer Simulation International, 1988. (cited on page 35.)

[54] E. Gonsiorowski, C. Carothers, and C. Tropper. Modeling large scale circuits using massively parallel discrete-event simulation. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 127–133. IEEE Computer Society, 2012. (cited on page 239.)

[55] B. Grošelj and C. Tropper. Pseudosimulation: An algorithm for distributed simulation with limited memory. *International Journal of Parallel Programming*, 15(5):413–456, 1986. (cited on page 41.)

[56] B. Grošelj and C. Tropper. The time of the next event algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation.* Society for Computer Simulation International, July 1988. (cited on page 41.)

[57] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP, pages 175–184. ACM, 2008. (cited on page 190.)

[58] T. Harris. A pragmatic implementation of non-blocking linked-lists. In J. Welch, editor, *Distributed Computing*, volume 2180, pages 300–314. Springer Berlin / Heidelberg, 2001. (cited on pages 99, 184.)

[59] 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, pages 409–416. ACM, 1986. (cited on page 30.)

[60] M. Herlihy and N. Shavit. On the nature of progress. In A. F. Anta, G. Lipari, and M. Roy, editors, *Proceedings of the* 15^{th} *International Conference on Principles of Distributed Systems*, volume 7109 of *OPODIS*. Springer Verlag, 2011. (cited on pages 175, 212.)

[61] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 276–290. ACM, 1988. (cited on page 10.)

[62] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, January 1991. (cited on pages 10, 60, 94, 98.)

[63] M. P. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News*, 21:289–300, May 1993. (cited on page 10.)

[64] P. Horn. Autonomic computing: IBM's perspective on the state of information technology. *White Paper*, 15:1–39, 2001. (cited on page 274.)

[65] C. Hou, D. Quinlan, D. Jefferson, R. Fujimoto, and R. Vuduc. Synthesizing loops for program inversion. In R. Glück and T. Yokoyama, editors, *Reversible Computation*, volume 7581 of *Lecture Notes in Computer Science*, pages 72–84. Springer Berlin Heidelberg, 2013. (cited on page 253.)

[66] C. Hou, G. Vulov, D. Quinlan, D. Jefferson, R. Fujimoto, and R. Vuduc. A new method for program inversion. In M. O'Boyle, editor, *Compiler Construction*, volume 7210 of *Lecture Notes in Computer Science*, pages 81–100. Springer Berlin Heidelberg, 2012. (cited on page 253.)

[67] HPDCS Research Group. ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. `http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/`, Oct. 2012. (cited on pages 56, 61, 63, 138, 279.)

[68] M. H. Hwang. Tutorial: Verification of real-time system based on schedule-preserved DEVS. In *Proceedings of the 2005 DEVS Symposium*. Society for Computer Simulation International, Apr. 2005. (cited on page 25.)

[69] M. H. Hwang. Qualitative verification of finite and real-time DEVS networks. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, pages 43:1–43:8. Society for Computer Simulation International, 2012. (cited on page 25.)

[70] M. H. Hwang, S. K. Cho, B. P. Zeigler, and F. Lin. Processing time bounds of schedule-preserving DEVS. Technical Report H1, AICM, July 2007. (cited on page 25.)

[71] M. H. Hwang and B. P. Zeigler. Reachability graph of finite and deterministic devs networks. *Automation Science and Engineering, IEEE Transactions on*, 6(3):468–478, July 2009. (cited on page 25.)

[72] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M.* (cited on pages 66, 114.)

[73] Intel Corporation. *IA-32 Intel(R) Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z.* (cited on pages 66, 114.)

[74] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. Optimization of parallel discrete event simulator for multi-core systems. In *Proceedings of the 2012 International Parallel and Distributed Processing Symposium*, IPDPS, pages 520–531. IEEE Computer Society, 2012. (cited on pages 36, 239.)

[75] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. *SIGOPS Operating Systems Review*, 21(5):77–93, Nov. 1987. (cited on pages 94, 138, 160.)

[76] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, July 1985. (cited on pages 39, 43, 47, 49, 64, 71, 78, 90, 139, 187.)

[77] D. R. Jefferson. Virtual Time II: storage management in conservative and optimistic systems. In *Proceedings of the 9th annual ACM symposium on Principles of distributed computing*, pages 75–89. ACM, 1990. (cited on pages 49, 90.)

[78] V. Jha and R. Bagrodia. Simultaneous events and lookahead in simulation protocols. *ACM Trans. Model. Comput. Simul.*, 10(3):241–267, July 2000. (cited on page 219.)

[79] D. B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 86–95. IEEE Computer Society, Oct. 1993. (cited on page 92.)

[80] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, Apr. 1986. (cited on page 30.)

[81] L. Junli. Agent-based logistics simulation system design and implementation. In *Proceedings of the 2^{nd} IEEE International Conference on Computer Science and Information Technology*, ICCSIT, pages 602–606. IEEE Computer Society, 2009. (cited on page 230.)

[82] S. Kandukuri and S. Boyd. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications*, 1(1):46–55, 2002. (cited on page 68.)

[83] T. Karmakharm and P. Richmond. Large scale pedestrian multi-simulation for a decision support tool. In *Proceedings of the 2012 Conference on Theory and Practice of Computer Graphics*, TPCG, pages 41–44. European Association for Computer Graphics, 2012. (cited on page 230.)

[84] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall Professional Technical Reference, 2nd edition, 1988. (cited on page 279.)

[85] S. Koenig and Y. Liu. Terrain coverage with ant robots: a simulation study. In *Proceedings of the fifth international conference on Autonomous agents*, AGENTS, pages 600–607. ACM, 2001. (cited on pages 69, 232.)

[86] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981. (cited on page 43.)

[87] Y. Labiche and G. Wainer. Towards the verification and validation of DEVS models. In *Proceedings of 1st Open International Conference on Modeling & Simulation*, pages 295–305, 2005. (cited on page 24.)

[88] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, Nov. 1977. (cited on pages 94, 98.)

[89] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. (cited on page 92.)

[90] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *IEEE International Symposium on Performance Analysis of Systems Software*, ISPASS, pages 175–183, Mar. 2010. (cited on page 264.)

[91] D. Lea. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`, 1996. (cited on page 109.)

[92] J. I. Leivent and R. J. Watro. Mathematical foundations of time warp systems. *ACM Transactions on Programming Languages and Systems*, 15(5):771–794, Nov. 1993. (cited on pages 47, 219.)

[93] X. Li, W. Mao, D. Zeng, and F.-Y. Wang. Agent-based social simulation and modeling in social computing. In C. C. Yang, H. Chen, M. Chau, K. Chang, S.-D. Lang, P. S. Chen, R. Hsieh, D. Zeng, F.-Y. Wang, K. Carley, W. Mao, and J. Zhan, editors, *Intelligence and Security Informatics*, volume 5075 of *Lecture Notes in Computer Science*, pages 401–412. Springer Berlin Heidelberg, 2008. (cited on page 31.)

[94] Y.-B. Lin and E. D. Lazowska. Determining the global virtual time in a distributed simulation. In B. W. Wah, editor, *Proceedings of the 19th International Conference on Parallel Processing*, ICPP, pages 201–209. Pennsylvania State University Press, 1990. (cited on page 48.)

[95] 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, Feb. 1990. (cited on page 79.)

[96] Y.-B. Lin and E. D. Lazowska. Processor scheduling for Time Warp parallel simulation. In *Proceedings of the 23rd SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 11–14. IEEE Comp. Soc., Jan. 1991. (cited on pages 57, 58.)

[97] M. Livny. A study of parallelism in distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 94–98. Society for Computer Simulation International, 1985. (cited on page 11.)

[98] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123, Jan. 1989. (cited on page 41.)

[99] E. W. Lynch and G. F. Riley. Hardware supported time synchronization in multi-core architectures. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 88–94. IEEE Computer Society, 2009. (cited on page 48.)

[100] E. W. Lynch and G. F. Riley. A sensitivity analysis of a new hardware-supported global synchronization unit. In *Proceedings of the 2009 IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, pages 1–4. IEEE Computer Society, Sept. 2009. (cited on page 48.)

[101] C. M. Macal and M. J. North. Tutorial on agent-based modeling and simulation part 2: How to model with agents. In *Proceedings of the 2006 Winter Simulation Conference*, WSC, pages 73–83. Society for Computer Simulation, 2006. (cited on page 230.)

[102] M. W. Macy and R. Willer. From factors to actors: Computational sociology and agent-based modeling. *Annual Review of Sociology*, 28(1):143–166, 2002. (cited on page 230.)

[103] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. WARPED: A Time Warp simulation kernel for analysis and application development. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, page 383. IEEE Comp. Soc., 1996. (cited on page 138.)

[104] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel Distributed Computing*, 18(4):423–434, 1993. (cited on pages 48, 285.)

[105] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. *System V Application Binary Interface AMD64 Architecture Processor Supplement*, Dec. 2007. (cited on pages 66, 111.)

[106] H. Mehl. A deterministic tie-breaking scheme for sequential and distributed simulation. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, PADS. ACM Press, 1992. (cited on page 219.)

[107] H. Mehl and S. Hammes. How to integrate shared variables in distributed simulation. *ACM SIGSIM Simulation Digest*, 25(2):14–41, Sept. 1995. (cited on pages 96, 173.)

[108] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004. (cited on page 99.)

[109] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, Apr. 1965. (cited on page 5.)

[110] MPI Forum. Message passing interface. `http://www.mpi-forum.org/`, 1994. (cited on page 36.)

[111] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns. Modeling a million-node dragonfly network using massively parallel discrete-event simulation. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC, pages 366–376. IEEE Computer Society, 2012. (cited on page 239.)

[112] R. Murphy, T. Sterling, and C. Dekate. Advanced architectures and execution models to support green computing. *Computing in Science Engineering*, 12(6):38–47, Nov. 2010. (cited on page 7.)

[113] S. Nanda, W. Li, L.-C. Lam, and T.-C. Chiueh. BIRD: Binary interpretation using runtime disassembly. In *International Symposium on Code Generation and Optimization*, CGO, Mar. 2006. (cited on page 264.)

[114] J. Newsome, D. Brumley, and D. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the 13th Symposium on Network and Distributed System Security*, NDSS, 2005. (cited on page 261.)

[115] D. M. Nicol. Parallel discrete-event simulation of fcfs stochastic queueing networks. *SIGPLAN Not.*, 23(9):124–137, Jan. 1988. (cited on page 42.)

[116] D. M. Nicol and X. Liu. The dark side of risk (what your mother never told you about time warp). In *Proceedings of the* 11^{th} *Workshop on Parallel and Distributed Simulation*, PADS, pages 188–195. IEEE Computer Society, 1997. (cited on pages 190, 211.)

[117] D. M. Nicol and P. F. Reynolds, Jr. Problem oriented protocol design. *SIGSIM Simul. Dig.*, 16(2):27–30, Apr. 1985. (cited on page 41.)

[118] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT, pages 365–375. IEEE Computer Society, 2007. (cited on page 261.)

[119] E. H. Page and R. Smith. Introduction to military training simulation: A guide for discrete event simulationists. In *Proceedings of the 30th Conference on Winter Simulation*, WSC, pages 53–60. IEEE Computer Society Press, 1998. (cited on page 31.)

[120] S. E. Page. Agent-based models. In S. N. Durlauf and L. E. Blume, editors, *The New Palgrave Dictionary of Economics*. Palgrave Macmillan, 2008. (cited on page 230.)

[121] A. C. Palaniswamy and P. A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and distributed simulation*, pages 127–134. ACM, 1993. (cited on pages 81, 90.)

[122] J. K. Peacock, J. Wong, and E. G. Manning. Distributed simulation using a network of processors. *Computer Networks*, 3, 1979. (cited on page 41.)

[123] A. Pellegrini. Hijacker: Efficient static software instrumentation with applications in high performance computing (poster paper). In *Proceedings of the 2013 International Conference on High Performance Computing & Simulation*, HPCS, pages 650–655. IEEE Computer Society, July 2013. Candidate for (but not winner of) the Outstanding Poster Paper Award. (cited on pages 62, 65, 111, 261.)

[124] A. Pellegrini and F. Quaglia. The ROme OpTimistic Simulator: A tutorial (invited tutorial). In *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*, PADABS. LNCS, Springer-Verlag, Aug. 2013. (cited on page 63.)

[125] 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 of the 23rd Workshop on Principles of Advanced and Distributed Simulation*. IEEE Comp. Soc., 2009. (cited on pages 89, 90.)

[126] A. Pellegrini, R. Vitali, and F. Quaglia. An evolutionary algorithm to optimize log/restore operations within optimistic simulation platforms. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. ICST, 2011. (cited on page 288.)

[127] 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. (cited on page 63.)

[128] S. Peluso, D. Didona, and F. Quaglia. Application transparent migration of simulation objects with generic memory layout. In *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation*, pages 169–177. IEEE Comp. Soc., june 2011. (cited on pages 37, 288.)

[129] K. S. Perumalla. μsik – a micro-kernel for parallel/distributed simulation systems. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 59–68. IEEE Computer Society, 2005. (cited on page 56.)

[130] Pin. http://www.pintool.org/. (cited on pages 261, 264.)

[131] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990. (cited on page 30.)

[132] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, pages 135–148. IEEE Computer Society, 2006. (cited on page 102.)

[133] F. Quaglia. Event history based sparse state saving in time warp. *SIGSIM Simulation Digest*, 28(1):72–79, 1998. (cited on pages 84, 90.)

[134] F. Quaglia. A cost model for selecting checkpoint positions in Time Warp parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):346–362, Feb. 2001. (cited on pages 84, 90.)

[135] F. Quaglia. On the construction of committed consistent global states in optimistic simulation. *International Journal of Simulation and Process Modelling*, 5(2):172–181, 2009. (cited on page 93.)

[136] H. Rajaei, R. Ayani, and L.-E. Thorelli. The local time warp approach to parallel simulation. *SIGSIM Simulation Digest*, 23(1):119–126, July 1993. (cited on page 53.)

[137] P. F. Reynolds, Jr. A shared resource algorithm for distributed simulation. *ACM SIGARCH Computer Architecture News*, 10(3):259–266, Apr. 1982. (cited on page 41.)

[138] P. F. Reynolds, Jr. A spectrum of options for parallel simulation. In *Proceedings of 1988 Winter Simulation Conference*, pages 325–332. Society for Computer Simulation, Dec. 1988. (cited on page 39.)

[139] S. Robinson. *Simulation: The Practice of Model Development and Use.* John Wiley & Sons, 2004. (cited on page 17.)

[140] R. Rönngren and R. Ayani. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 110–117. Society for Computer Simulation, July 1994. (cited on pages 82, 90, 129.)

[141] 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*, pages 70–77. IEEE Comp. Soc., May 1996. (cited on pages 87, 90.)

[142] A. Saltelli, M. Ratto, T. Andres, F. Campolongo, J. Cariboni, D. Gatelli, M. Saisana, and S. Tarantola. *Global Sensitivity Analysis: The Primer.* Wiley, 2008. (cited on page 11.)

[143] A. Santoro and F. Quaglia. Software supports for event preemptive rollback in optimistic parallel simulation on myrinet clusters. *Journal of Interconnection Networks*, 6(4):435–457, 2005. (cited on pages 211, 212.)

[144] A. Santoro and F. Quaglia. Transparent state management for optimistic synchronization in the High Level Architecture. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 171–180. IEEE Comp. Soc., June 2005. (cited on pages 85, 86.)

[145] A. Santoro and F. Quaglia. A version of MASM portable across different UNIX systems and different hardware architectures. In *Proceedings of the 9th International Symposium on Distributed Simulation and Real Time Applications*. IEEE Comp. Soc., Oct. 2005. (cited on pages 85, 86, 90.)

[146] A. Santoro and F. Quaglia. Transparent optimistic synchronization in the high-level architecture via time-management conversion. *ACM Transactions on Modeling and Computer Simulation*, 22(4):21:1–21:26, Nov. 2012. (cited on page 94.)

[147] T. Santoro and F. Quaglia. A low-overhead constant-time LTF scheduler for optimistic simulation systems. In *Proceedings of the IEEE Symposium on Computers and Communications*, pages 948–953, 2010. (cited on pages 58, 288.)

[148] R. G. Sargent. Verification and validation of simulation models. In *Proceedings of the 2010 Winter Simulation Conference*, pages 166–183. Society for Computer Simulation International, Dec. 2010. (cited on page 24.)

[149] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, Aug. 1995. (cited on pages 11, 96.)

[150] S. S. Shende and A. D. Malony. The tau parallel performance system. *International Journal on High Performance Computing Applications*, 20(2):287–311, May 2006. (cited on page 261.)

[151] R. Simmonds, R. Bradford, and B. Unger. Applying parallel discrete event simulation to network emulation. In *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, PADS, pages 15–22. IEEE Computer Society, 2000. (cited on page 35.)

[152] S. Skold and R. Rönngren. Event sensitive state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 1996 Winter Simulation Conference*, pages 653–660. Society for Computer Simulation, december 1996. (cited on pages 82, 90.)

[153] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985. (cited on page 30.)

[154] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148. IEEE Computer Society Press, 1981. (cited on page 43.)

[155] H. Soliman and A. Elmaghraby. An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):947–951, october 1998. (cited on page 39.)

[156] SPEEDES. `http://www.speedes.com`, 2005. (cited on pages 86, 138.)

[157] S. Srinivasan and P. F. Reynolds, Jr. Elastic time. *ACM Transactions on Modeling and Computer Simulation*, 8(2):103–139, Apr. 1998. (cited on pages 39, 54.)

[158] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Languages and Design Implementation*, pages 196–205. ACM, 1994. (cited on page 263.)

[159] J. S. Steinman, C. A. Lee, L. F. Wilson, and D. M. Nicol. Global Virtual Time and distributed synchronization. *SIGSIM Simulation Digest*, 25:139–148, July 1995. (cited on page 48.)

[160] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. (cited on pages 6, 7.)

[161] J. Svennebring and S. Koenig. Building terrain-covering ant robots: A feasibility study. *Autonomous Robots*, 16(3):313–332, May 2004. (cited on pages 70, 71, 234.)

[162] 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*, pages 44–52, 2012. (cited on page 199.)

[163] T. Takahashi, S. Tadokoro, M. Ohta, and N. Ito. Agent based approach in disaster rescue simulation - from test-bed of multiagent system to practical application. In *RoboCup 2001: Robot Soccer World Cup V*, pages 102–111. Springer-Verlag, 2002. (cited on page 230.)

[164] The SCO Group, Inc. *System V Application Binary Interface*, fourth edition, Mar. 1997. (cited on page 118.)

[165] The SCO Group, Inc. *System V Application Binary Interface, Intel386 Architecture Processor Supplement*, fourth edition, Mar. 1997. (cited on pages 66, 111.)

[166] R. Toccaceli and F. Quaglia. DyMeLoR: Dynamic Memory Logger and Restorer library for optimistic simulation objects with generic memory layout. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172. IEEE Comp. Soc., 2008. (cited on pages 85, 86, 102, 129, 288.)

[167] Valgrind. http://valgrind.org/. (cited on pages 261, 264.)

[168] R. Vitali, A. Pellegrini, and F. Quaglia. Autonomic log/restore for advanced optimistic simulation systems. In *Proceedings of the Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 319–327. IEEE Comp. Soc., 2010. (cited on pages 89, 90, 121, 272, 274, 288.)

[169] R. Vitali, A. Pellegrini, and F. Quaglia. Assessing load sharing within optimistic simulation platforms (invited paper). In *Proceedings of the 2012 Winter Simulation Conference*, WSC. Society for Computer Simulation, Dec. 2012. (cited on pages 37, 222, 244, 281.)

[170] R. Vitali, A. Pellegrini, and F. Quaglia. A load sharing architecture for optimistic simulations on multi-core machines. In *Proceedings of the 19th International Conference on High Performance Computing*, HiPC. IEEE Comp. Soc., Dec. 2012. (cited on pages 37, 222, 244, 281.)

[171] R. Vitali, A. Pellegrini, and F. Quaglia. Load sharing for optimistic parallel simulations on multi core machines. *SIGMETRICS Performance Evaluation Review*, 40(3):2–11, Aug. 2012. (cited on pages 37, 222, 244, 281.)

[172] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th International Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 211–220. IEEE Comp. Soc., Aug. 2012. (cited on pages 36, 105, 159.)

[173] G. Vulov, C. Hou, R. Vuduc, R. M. Fujimoto, D. Quinlan, and D. Jefferson. The backstroke framework for source level reverse computation applied to parallel discrete event simulation. In *Proceedings of the 2011 Winter Simulation Conference*, WSC, pages 2960–2974. IEEE Computer Society, Dec. 2011. (cited on page 253.)

[174] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 1–12, 1993. (cited on pages 103, 112.)

[175] G. Wainer. CD++: A toolkit to develop DEVS models. *Software—Practice & Experience*, 32(13):1261–1306, Nov. 2002. (cited on page 25.)

[176] G. A. Wainer. *Discrete-Event Modeling and Simulation: A Practitioner's Approach.* Computational Analysis, Synthesis, and Design of Dynamic Systems. Taylor & Francis, 2008. (cited on page 28.)

[177] J. Wang and C. Tropper. Selecting GVT interval for Time-Warp-based distributed simulation using reinforcement learning technique. In *Proceedings of the 2009 Spring Simulation Multiconference*, SpringSim, pages 49:1–49:7. Society for Computer Simulation International, 2009. (cited on pages 48, 54.)

[178] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 285–296. ACM, 2008. (cited on page 92.)

[179] D. West and K. Panesar. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 78–85. IEEE Comp. Soc., May 1996. (cited on pages 52, 88, 90, 102.)

[180] N. H. Weste and K. Eshraghian. *Principles of CMOS VLSI design: a systems perspective.* VLSI systems series. Addison-Wesley Pub. Co., 1993. (cited on page 6.)

[181] B. Wester, P. M. Chen, and J. Flinn. Operating system support for application-specific speculation. In *Proceedings of the 6th Conference on Computer Systems*, EuroSys, pages 229–242. ACM, 2011. (cited on page 93.)

[182] F. Wieland, L. Hawley, A. Feinberg, M. D. Loreto, L. Blume, J. Ruffles, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson. The performance of a distributed combat simulation with the time warp operating system. *Concurrency: Practice and Experience*, 1(1):35–50, 1989. (cited on page 35.)

[183] B. P. Zeigler. *On the Feedback Complexity of Automata.* PhD thesis, University of Michigan, Computer and Communication Sciences Department, 1968. (cited on page 18.)

[184] B. P. Zeigler. *Theory of Modelling and Simulation.* A Wiley-Interscience Publication. John Wiley, 1976. (cited on page 18.)

[185] B. P. Zeigler. *Multifaceted modelling and Discrete Event Simulation.* Academic Press Professional, Inc., 1984. (cited on pages 19, 21.)

[186] B. P. Zeigler. Hierarchical, modular discrete-event modelling in an object-oriented environment. *Simulation*, 49(5):219–230, Nov. 1987. (cited on page 19.)

[187] B. P. Zeigler. DEVS representation of dynamical systems: event-based intelligent control. *Proceedings of the IEEE*, 77(1):72–80, Jan. 1989. (cited on page 26.)

[188] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC/ETAPS, pages 147–162. Springer-Verlag, 2008. (cited on page 102.)