TOR VERGATA
UNIVERSITY OF ROME

Ph.D. program in

COMPUTER SCIENCE, CONTROL AND GEOINFORMATION

XXXIV cycle - A.Y. 2021/22

New methods for the effectiveness of speculative parallel discrete event simulation

Advisor

Prof. Francesco Quaglia

Co-Advisor

Dr. Alessandro Pellegrini

Candidate

Matteo Principe

Reviewers

Prof. Alfredo Garro

Prof. Kevin Jin

Acknowledgment

Before starting, I would like to convey my deepest gratitude to everyone who walked with me in the journey of this Ph.D.

First, I would express my deepest gratitude to my advisor, Prof. Francesco Quaglia, who drove me in the results achieved at first in my master thesis, and finally with this dissertation. From the very beginning he devolved me his passion for simulation, and his interest in this challenging computer science field. His patience, his sincerity and his ability to understand my weaknesses were just some of the qualities that demonstrate how much he guided me throughout these studies. I also thank my co-advisor, Alessandro Pellegrini, for all the time he dedicated to me and the huge help he provided during all this long path.

I would like to acknowledge the priceless input of the HPDCS research group in Sapienza, who initiated me to the research life and gave me the possibility to collaborate with skilled, but and the same time humble, people.

Then, I cannot forget my family. My parents, who filled with love all my fears and my sisters, who enlightened me with their strength, were the wind carrying the boat of my studies in the sea of the research. I am really convinced that without your smiles, your examples and your inheritance I couldn't overcome the tallest waves of the storms I needed to face. Thanks to my grandparents, some of them seeing this achievement from somewhere high in the sky. Without you, the kid I used to be couldn't get this far, and

I'm sure you'll keep holding my hand while taking the hardest choices the future will put in front of me.

One could think that the Ph.D. is a personal achievement. Well, while it actually is, for me it couldn't be reached without the encouragement and the kindness of my buddies, Stefano and Andrea. I really thank you, guys, for everything you donated me: your kindness, our laughs, the beers the countless times you were helping me in believing in myself... I will bring those memories with me forever.

Last but not least, I would like to thank Alessandra. Your love, and your (sometimes unbelievable) comprehension where invaluable gifts you brought in my life gratuitously. This is something I will always have impressed in my heart, and I hope I will be able, one day, to do the same for you.

Matteo Principe

We make a living by what we get.
We make a life by what we give.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Go back?... No good at all! Go sideways? Impossible! Go forward? Only thing to do! On we go!

— Bilbo Baggins, The Hobbit

The last decade has seen the rise of challenging large-scale resource-intensive scientific computations and the emergence of Cloud Computing (CC) as a prominent computing system. Historically, CC revealed itself as an excellent choice to run high-performance software at a competitive cost. The explosion of applications ready to be run on the cloud has allowed end users to break the wall of on premise computation, making their soutions available without the need of expensive computing systems. Yet, its growth didn't come at the same speed as the increase of programming paradigms and abstractions. Indeed, CC exploitation in high-performance applications still largely relies on programmers' distributed/parallel programming skills, which are prerogatives of a restricted set of experts. If, for instance, serverless computing can be considered very convenient in terms of degree of freedom for developers and in terms of price, it on the other hand presents limitation for certain high-performance computing workloads (resource restrictions, harder debugging, etc). At the same time, new virtualization (and containerization) techniques provide lightweight and fault-tolerant deployments of applications while maintaining low prices for their execution.

However, while CC is gaining access to flexibility in terms of hardware and software capabilities, it is someway increasing the programmers' obliviousness of the underlying platform. Indeed, software abstractions represent a clear tradeoff between programmability and performance, as the cost associated with the production of code handling the underlying platform is masked by the provided easiness of the deployment on cloud clusters. Therefore, it is necessary to provide a methodology accessible to the masses, which is at the same time ensuring high performance standards. For this reason academic communities are spending a lot of effort on it. As an example, the Partitioned Global Address Space (PGAS, [Sti09]) programming model gives developers the possibility to generate code that can be efficiently run on top of parallel/distributed architectures, possibly made of VMs. However, while providing significant advantages, this solution fails at preventing demand from coders high skills since it is based on particular programming languages such as Unified Parallel C, Co-Array Fortran and Titanium.

As far as programmability is concerned, the increased complexity of new (possibly heterogeneous) architectures has brought the IT industry and research to develop efficient *Domain-Specific Languages* (DSL) to provide ad-hoc primitives for certain areas of interest, typically scientific. Indeed, a DSL targeting a specific field allows experts of the domain to focus only on relevant aspects of the problem of interest rather than handling issues related to the architecture the software needs to run on. While the approach looks promising, unfortunately, in the context of more compute-intensive domains, there is still a significant lack of well-accepted DSLs.

One of the main purposes of this thesis is to provide the reader with solutions to deploy complex compute-intensive applications implemented with simplified programming models on top of arbitrarily complex parallel/distributed infrastructures. In fact, a core aspect tackled in this thesis

is transparency, in the sense that tools and methodologies are provided to the final developer with no (or slight) modifications to the original code, allowing them to continue using the original (sequential) code on top of multi-core or clusters of machines. By avoiding the need for fine-tuning the code to cope with architecture-specific characteristics, the productivity of programmers is enhanced, resulting in *i)* faster development, *ii)* reduced debugging time, *iii)* no high skill requirements for problems to be run on top of supercomputers, *iv)* increased testability. On the other hand, the results of this thesis also provide solutions that require the end-user to intervene concretely. In fact, there are situations in which having a high degree of freedom in customizing runtime environments can provide much better performance or more accurate results. Moreover, it is often the case that the programmers are aware of some insight the system can be fed with to impact the overall execution. Therefore, the results of this thesis are comprehensive of the tradeoff between programmability and final performance level, providing tools to abstract the low-level difficulties of parallel/distributed computing platforms and, at the same time, the possibility to deliberately tune computing environments to their needs.

The research context that has been chosen for the realization of the goals mentioned earlier is *simulation*, and this choice has been made for several reasons. First of all, this is a hot field in the scientific world since many researchers demand timely production of simulated outputs in critical scenarios of decision making, for example, in symbiotic systems or what-if analysis ([ATCL09], and [AST07], respectively). Moreover, the last two years have proven that the request for simulation of real-world scenarios is really crucial: the COVID-19 pandemics had impacts on many fields, spanning from supply chain to healthcare systems, and trying to predict what the consequences of the adopted measures to stop the spreading of the

disease could be is considered more and more a concern. This is confirmed by the number of publications that moved at a fast pace towards this direction: examples of such research results can be found in [Iva20], and [WCDC+20].

Secondly, the simulation field often deals with problems which domain inherently requires a high amount of parallelism. This happens because simulation runs need to process models that are constantly growing in their size, in terms of parameters, actors playing in the environment, data to be processed for statistics purposes, dimension of datasets to be analyzed, etc.

Finally, being a vast field involving many scientific areas, it is often the case that scientists defining the models to be simulated are not computer science specialists. Rather, they belong to a plethora of different areas, including, for instance, meteorology, biology, economy, finance, neuroscience, etc. Therefore, since their area of expertise does not necessarily involve computer science concepts, taking care of concerns related to system architectures or performance is a requirement of the tools exploited to carry on the simulations of interest. For all these reasons, simulation can be considered a well-suited context to implement the aforementioned transparent and performance-oriented tools to reduce the burden of complex parallel programming concerns to final developers (i.e., scientists writing their models). The results presented in this thesis appear in the following publications:

1. Matteo Principe, Tommaso Tocci, Alessandro Pellegrini, Francesco Quaglia: Porting Event & Cross-State Synchronization to the Cloud. SIGSIM-PADS 2018: 177-188

2. Matteo Principe, Tommaso Tocci, Pierangelo di Sanzo, Francesco Quaglia, Alessandro Pellegrini: A Distributed Shared Memory Middleware for Speculative Parallel Discrete Event Simulation. ACM Trans. Model. Comput. Simul. 30(2): 11:1-11:26 (2020)

3. Matteo Principe, Andrea Piccione, Alessandro Pellegrini, Francesco Quaglia: Approximated Rollbacks. SIGSIM-PADS 2020: 23-33

4. Andrea Piccione, Matteo Principe, Alessandro Pellegrini, Francesco Quaglia: An Agent-Based Simulation API for Speculative PDES Runtime Environments. SIGSIM-PADS 2019: 83-94

As a last note, all the research spent in the years of my doctorate also produced contributions to the scientific community of interest. Particularly, review processes led to the publication of the following papers:

- Matteo Principe: Reproducibility Report for the Paper: Probing the Performance of the Edinburgh Bike Sharing System using SSTL. SIGSIM-PADS 2020: 153-155

- Emilio Incerto, Matteo Principe: Reproducibility Report for the Paper: "Differentiable Agent-Based Simulation for Gradient-Guided Simulation-Based Optimization". SIGSIM-PADS 2021: 39-43

CHAPTER 2

Research background and attained results

All we have to decide is what to do with the time that is given us.

— Gandalf, The Lord Of The Rings

The area into which the results presented in this thesis reside is simulation, a discipline that has been used since ancient times and that during the computing era arose again as a promising technique to advance research in different scientific fields. The term simulation refers to the act of mimicking real-world scenarios in order to reproduce a physical (or even synthetic) system. Such an operation is typically performed through abstractions aimed at imitating what the natural system would have done. Specifying a set of rules, requirements, and responses to actions in a model, we can provide a reasonable estimation of what would have happened in the actual (i.e., the modeled) situation. Note, however, that this could be very hard (sometimes nearly impossible) to achieve in practice, as, for instance, physical systems can involve an unbearable amount of computation, both in terms of time and costs. Indeed, representing such a dynamic in a computer program can be challenging, as the magnitude of variables involved in a physical phenomenon and the interactions with other systems could be unpredictable or unbearable to program. For this reason, starting from the late 70s, re-

search in computer simulation began to grow increasingly and continues to gain more and more appeal: the rising of new computing architectures has unlocked new possibilities to tackle open issues in this field.

Two main aspects distinguish simulation systems from each other: time and randomization. Therefore, the simulation taxonomy depends on how critical they are to the considered system and how they are handled during simulation. Indeed, there are situations in which time is not an interesting quantity, for example, when the objective is to evaluate or approximate a mathematical problem that is not easy to solve analytically and involves a high number of variables. This is the case of numerical methods, used when analytical or symbolic approaches to solving math problems are computationally complex (e.g., Stochastic Differential Equation solution evaluation). Consider, for instance, the case of the approximation of the value of a finite integral of a function. It is decisive to pick as many randomly distributed points falling in the integration interval as possible in order to sum all the values the function gets in those points and divide them by the total number of points. Monte Carlo simulations typically exploit this technique and repeat the process multiple times in order to converge to a particular result within the considered approximation window. On the other hand, there are plenty of scenarios in which time is the essential variable to consider. Physical systems are strictly related to time as things evolve according to it, and depending on how time is modeled, we span from continuous to discrete simulation. The first one best suits the case of a natural physical system and the progress is guaranteed by the estimation of mathematical functions that describe the behavior of the system itself. For instance, heat diffusion in environments are ruled by a set of differential equations that must be evaluated to reproduce the area's state over time accurately. Although this approach faithfully provides results comparable to real systems,

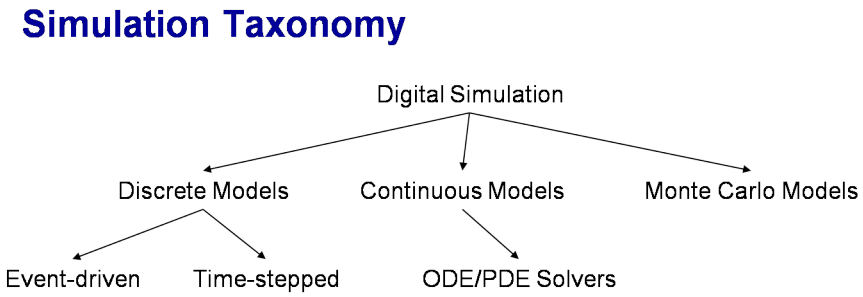


FIGURE 2.1: *Simulation taxonomy*

the computation associated with it could become hard.

On the other hand, discrete simulation better handles the situation in which the time interval between evolutions of the model can vary, spanning seconds, hours, days, and so on. This is the situation, for instance, of traffic jams or demographic growths, in which a single-step evolution of the system can be seen as an event that discretely takes place. Figure 2.1 exposes a comprehensive scheme of the simulation taxonomy, where a narrower distinction among discrete simulations is made. Discretization could be carried out via either the advancement of fixed time steps, which makes the execution progress depending on the rules based on that time progression, or the execution of discrete events which define the evolution of the system based on their own time of occurrence and rules associated with their processing. The latter one is the one of interest of this theses and will be thoroughly discussed in the following sections.

2.1 Discrete Event Simulation

Discrete event simulation (DES) maps the operation of a (physical) system to a discrete sequence of subsequent events over time. Any of these events happen at a precise instant in time and modify the state of the system. Indeed, events are considered discrete if they mark a change in the sim-

ulations' state in an impulsive manner. The next-event time progression ensures that no change in the system is assumed to occur between consecutive events. Therefore, the simulation time can directly shift to the time of occurrence of the next event to be scheduled.

The entities carrying out the simulation are called *Logical Processes* (LPs), which update their state by executing events and step forward in their virtual simulation times. Moreover, logical processes bring on their own simulation trajectory by interacting with other LPs via message exchange, producing events targeting either themselves or other entities, and executing those events independently. Formally, according to the *Discrete Event Systems Specification* (DEVS, [Zei19]), a discrete event simulation model can be represented as the following tuple:

$$D =< E_i, E_o, S, f_a, \delta_e, \delta_i, \sigma >, \tag{2.1}$$

where:

- $E_{in} \in E$ is the set of input events that the model can handle.
- $E_{out} \in E$ is the set of output events that the model can handle.
- $S \subset Q$ is the set of states the simulation can be at a specific instant of time. Model writers usually enhance simulation state definitions with many parameters, such as the maximum time spent when no external events are received before triggering internal logic. S_0 represents the initial state of the simulation.
- f_a is the time advance function, useful to describe how much time to persist in a particular state, after which internal state transitions might be triggered.
- δ_e is the external transition function defining the way an input event

$E_i \subset E_{in}$ marks a change in the state of the simulation. This function can be defined as $Q \times E_{in} \to S$ where $Q = \{\,(s_j, t_e) \;:\; s_j \in S, t_e \in [0, f_a(s_j)]\,\}$.

- δ_i represents the internal transition function, determining how the simulation updates its state internally when no external event occurs after the expiration of its lifetime.

- σ is the output function, which generates external output via the production of "unrecognized" events, i.e., events belonging to the set $E_u = \{(e \in E) : e \notin E_{out} \wedge e \notin E_{in}\}$.

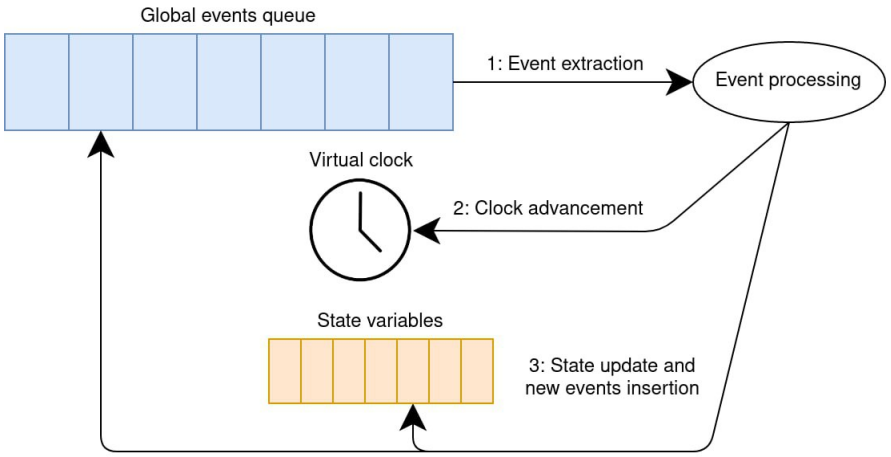


FIGURE 2.2: *DES basic kernel schema*

To summarize, discrete event simulation is carried out by a set of N logical processes $L_0, \ldots, L_{N-1}$, which represent the leading simulation entities, evolving in time due to state transitions managed by model's events, and keeping a state S_k, $k \in [0, N-1]$.

When dealing with the software implementation of DES simulators, a clear distinction between the simulation model and the simulation kernel is usually enforced. A software simulation model mirrors a phenomenon

by designing data structures, functions, and algorithms to recreate the intended situation. The simulation kernel's goal is to coherently and correctly drive the execution of events spawned by the model on top of computing platforms, ensuring consistency and possibly performance.

A classical implementation is based on the concepts of the event-driven programming paradigm, which alternates execution between two phases: *i)* event detection/scheduling and *ii)* event consuming. The first one consists in capturing simulation events resembling hardware system behaviors upon interrupts arrival. Practically, as depicted in Figure 2.2, the first point is reflected in the process of fetching the next event (according to a predefined rule, e.g., smallest timestamp first) and posting new events to the event queue. The latter point deals with the actual execution of actions specified by the event, consequently modifying the simulation state. Indeed, event consuming is represented by the tasks related to simulation time advancement, event execution, statistics update and termination condition evaluation.

Briefly, a (single-threaded) discrete event simulator can be schematized as in Figure 2.2, where:

- A **global event-queue** takes care of memorizing all the events that are required to be executed in order to complete the simulation. This queue is filled by logical processes that schedule events based on the logic in the model carried out during the simulation.

- A **shared virtual clock** is maintained among all entities to coordinate them throughout the whole simulation. Being discrete, this clock keeps track of the simulation time by jumping from timestamp T_e of an event e to T_{e_1} of a subsequent event e_1.

- A **state variables structure** tracks the changes that every simula-

tion object (namely, an LP) performs while executing events. Maintaining this information could be critical as it impacts the execution of new events and determines the final outcome of the whole simulation.

- The **event processing submodule** is taking care of executing the instructions enclosed in an event's logic, which can modify the simulation state, trigger other events, or both of them.

Although these can be considered the building blocks of a discrete event simulator, other relevant components turn out to be of genuine interest when simulating a phenomenon. First off, *Random-Number Generators* play an essential role as being real-world phenomena strictly dependable on random incidents, simulators need to recreate this unpredictability.

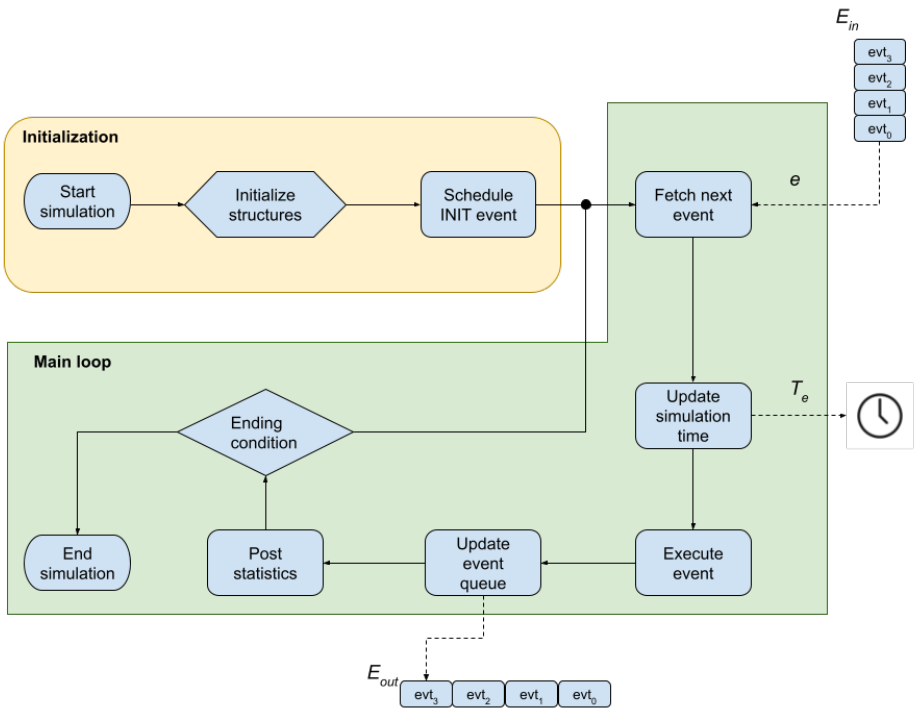


FIGURE 2.3: *DES flow chart*

Also, it is usually the case that simulations are run to predict a specific situation or perform a so-called *what-if* analysis. In such cases, *Statistics*

Submodules are designed to collect and retrieve information about how the system is evolving, both during execution and after completion, allowing the modeler to understand the modeled phenomenon better or even achieve better performance.

Finally, each simulator needs to have an *Ending Condition* which dictates when a simulation run should end. Depending on the kind of phenomenon being simulated, this condition can vary and be very model-specific. Classical implementations rely on time constraints or on the number of processed events. The main phases of a simulation kernel are depicted in Figure 2.3.

At first, the simulation kernel needs to initialize the resources which are necessary to prepare the simulation. This not only includes, for instance, memory allocation or data structure set-up, but it could also involve some model-level callback required to generate a consistent initial state of simulation objects. As an elementary example, the reader can think of the scenario of disease-spreading simulation, in which usually at the beginning of the simulation, agents are randomly placed in topology, with a set of parameters assigned to them.

After this initialization phase, the actual core of the simulation is fired. Indeed, the actions to be performed by the simulator are repeatedly executed in a main loop serving *i)* event selection *ii)* event execution *iii)* simulation clock update *iv)* statistics update *v)* termination condition evaluation. A high-level implementation of a basic DES kernel is shown in Algorithm 2.1.

Algorithm 2.1 DES kernel main loop

```
1: procedure INIT
2:     endingCondition ← false
3:     INITIALIZEVIRTUALCLOCK()
4:     SCHEDULEINITEVENT()
5: end procedure
6: while endingCondition ≠ true do
7:     nextEvent ← FETCHNEXTEVENT()
8:     clock ← nextEvent.timestamp
9:     EXECUTEEVENT(nextEvent)
10:    UPDATESTATISTICS()
11: end while
```

2.2 Parallel Discrete Event Simulation

While many studies in the literature have proven that discrete event simulation can represent the best choice for various scenarios to simulate, it faces considerable limitations when dealing with contemporary computing platforms. Indeed, even if what we presented in the previous section is a very basic implementation of a DES kernel, it should have hinted to the careful reader that the proposed solution inherently presents limitations.

Indeed, a serial implementation is by design under-utilizing the nowadays multi/many-core architectures it runs on top of: a single event is retrieved from the event queue at each simulation step while at the same time, other logical processes, which could be ready to launch possibly unrelated events, wait for this event to conclude its execution. Historically, this kind of problem was addressed by the definition of distributed DES ([Sut13]), which was trying to involve as much computing power as possible by spreading the DES building blocks (event queues, global clocks, etc.) over different CPU or cores. Naturally, splitting the work over several instances was not enough, as synchronization points were needed to share the independently computed data.

While this would have been a viable solution when CPUs were growing vertically (i.e., the increasing computational power of a single core, according to Moore's law [Moo06]), it eventually became unfeasible when physical limitations, namely the clock-frequency wall, were approaching ([Sut13]). In other words, the concepts introduced in the previous section do not take into account parallelism.

Starting from 1979, research in parallel and distributed simulation began to become more and more of interest, finally yielding to the birth of Parallel Discrete Event Simulation (PDES, [CM79]). At the basis of PDES strategies, there is the concept of logical process, introduced in the previous chapter. However, what differs from DES is that to exploit multiple computing units, those LPs need to avoid sharing portions of the overall state of the simulation model. Formally, this means that the whole simulation state is the composition of the states of all the logical processes, i.e.,

$$S = \bigcup_{k=1}^{N} S_k \quad s.t \quad S_{LP_i} \cap S_{LP_j} = \varnothing \quad \forall i \neq j \tag{2.2}$$

needs to hold. Furthermore, by eliminating the possibility to transfer data by the use of shared variables, inter-LP communication can be performed only via message passing.

Moreover, the other key concept related to PDES lies in the fact that each LP controls its own virtual clock rather than being dependant on a globally synchronized clock. These assumptions allow LPs to behave as autonomous entities, each one taking care of its own simulation trajectory and interacting with other LPs using time-stamped event messages between corresponding LPs. Having stated this concept, we can better delineate a more technical picture of a possible PDES architecture. A classical architecture is depicted in Figure 2.4.

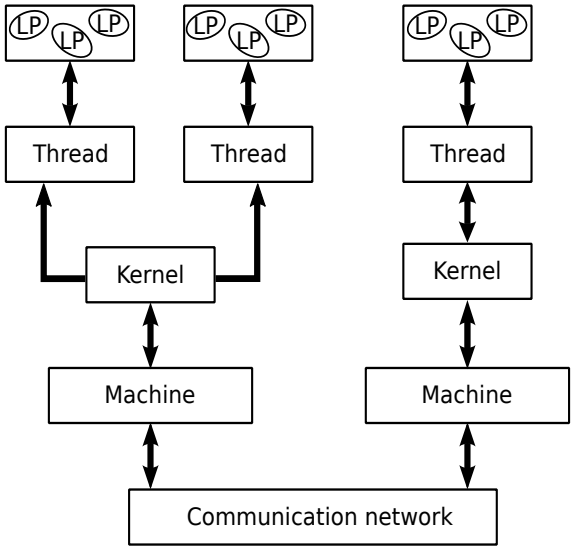


FIGURE 2.4: *Classical PDES architecure*

First of all, also PDES architectures are based on the idea of simulation kernels, which are usually implemented as user-space processes running on top of host operating systems. Simulation models drive the execution of a simulation kernel through minimal programming interfaces, allowing the modeler to specify which kind of events to generate, the ending condition, the simulation state, etc. Furthermore, simulation kernel instances manage a set of logical processes, which advance their execution independently according to the rules defined in the model. Finally, simulation kernels (or even LPs, transitively) can be either logically assigned to a specific computing unit or managed by a set of them, thus effectively taking advantage of the *Symmetric Multi-Processing* paradigm splitting up the whole load among all available CPUs/cores.

However, more sophisticated techniques need to be put in place in this case as differentiation between threads (e.g., workers vs. communicating) is crucial to avoid bottlenecks. While this was a revolutionary technique to speed up the processing of simulations, Moore mentioned above's law brought PDES to the usage of distributed clusters of machines intercon-

nected with each other through a network. Thus, logical processes can interact with each other either via the use of the host operating system facilities (e.g., Inter-Process Communication) if they belong to kernel instances running on top of the same machine, or using message passing primitives (e.g., Message Passing Interface, MPI[1]) if they are based on remote simulation kernel instances.

An essential characteristic of the latter organization is the fact that, in general, LPs may or may not directly access other one's state, even if they lie on the same simulation kernel. Instead, again, this kind of operation is only supported via message exchange, which is an important point to stress since it will be discussed in the remainder of this thesis.

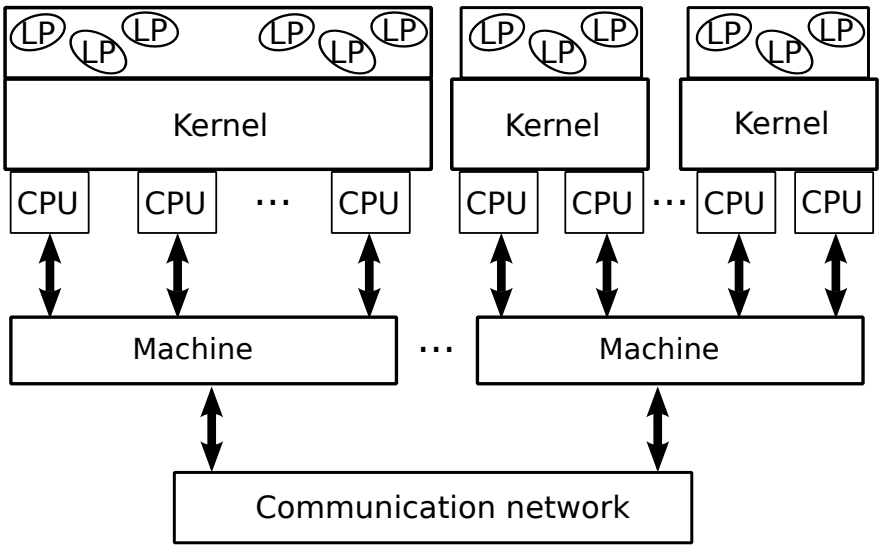


FIGURE 2.5: *SMT/distributed PDES architecure*

2.2.1 Synchronization approaches

Parallel discrete event simulation has been historically realized to exploit the growing multi-core/distributed architectures thoroughly. However, eventually, causal consistency became more and more the real bottleneck of such a

[1]https://www.mpi-forum.org/

kind of implementation: speculative executions came to the rescue by making simulation objects care "only when really necessary" about the consistency of data shared with other LPs ([Jef85]).

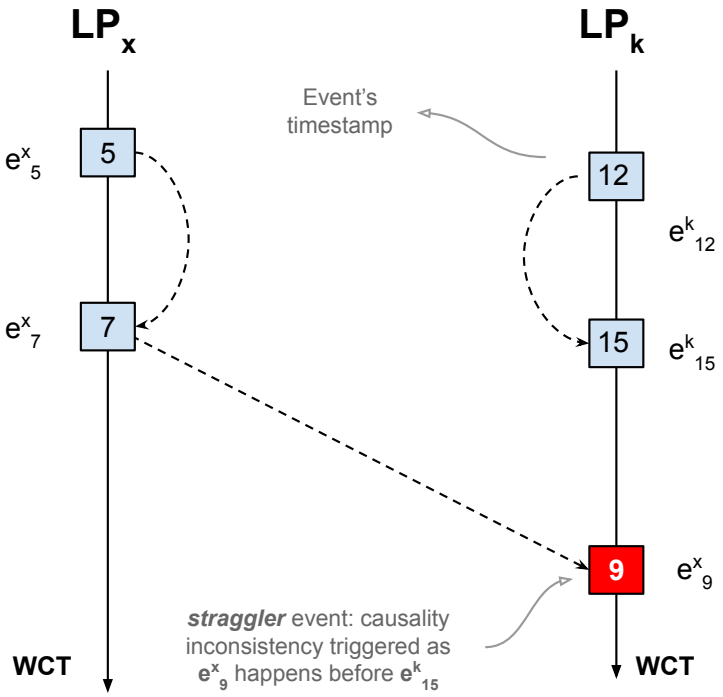


FIGURE 2.6: *An example of causality violation*

In Figure 2.6, we show an example of a situation in which some kind of technique must enforce a consistent situation. In particular, LP_x is scheduling an event with timestamp 9 towards LP_k, which, on the other hand, is logically ahead in time concerning LP_x as it is executing an event at local virtual time (LVT) 15. However, the latter logical process isn't allowed to simply discard the received event because the logic associated with it can bring some modifications that should have been observed by events with timestamp higher than it (event e^k_{15}, for instance).

In other words, ordering the timestamps between events matters as the state of a logical process can possibly be modified by executing any of them. These modifications would lead to incorrect outcomes if executed in a

causal inconsistent way. As a general rule, a causality violation takes place whenever an LP_i receives an event E_j with timestamp $T_j < LVT_i$. This, of course, entails the necessity of a methodology to enforce consistency, which depending on the approach taken into account, spans over conservative, optimistic and hybrid techniques.

Conservative synchronization

The first technique that can be adopted to avoid inconsistent situations is called conservative synchronization ([Bry77], [CM79]), and its main objective is to force the system not to generate occurrences of time-inconsistent conditions. The idea behind this approach is to use FIFO reliable links, which are static by their definition and assign to each LP a predetermined amount of FIFO channels, characterized by a timestamp value. This value can be either set to the element's timestamp at the top of the queue or the time of the last processed event. Then, the scheduling procedure will select, for a given LP, the next event from the queue with the smallest timestamp among all possible queues. This ensures that the system is always choosing the earliest among all possible events to be scheduled: no erroneous situation can be generated in any way, as they are avoided by design.

Although this could be both easy to implement and safe to execute, conservative synchronization comes with a number of drawbacks. First of all, efficiency could be hampered by a situation in which, for instance, a queue Q_i marked with the smallest timestamp among all queues is empty. Even if another list, say Q_x, is not empty, events can't be extracted from it as this would lead to executing an event "in the future", eventually bringing the system in an erroneous state whenever an event is inserted in Q_i.

Moreover, this kind of solution needs to deal with deadlock occurrences. Consider, for instance, the example shown in Figure 2.7. Here, three logical

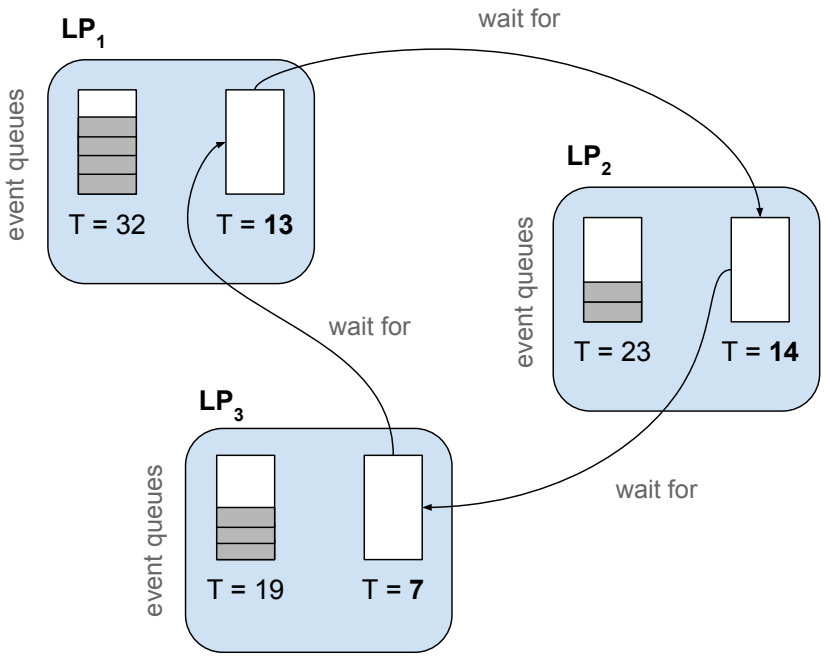


FIGURE 2.7: *Example of deadlock in the conservative synchronization.*

processes are maintaining events in two different lists, and all of them need to select events from an empty queue (i.e., their current queue with the lowest timestamp is the empty one, which is an empty box in the figure).

Definitely, this is a situation in which the advancement of the simulation is stalled: the system can't either pick a new event to schedule, neither will insert new events in the empty queues of the LPs. This problem can be addressed by detecting this deadlock situation via the usage of *dummy* events ([CT92], [Fuj90]), which do not entail logic but are useful to notify receiving LPs that the sender will not send any event marked with timestamp $T < T_{dummy}$. Thus, LPs are free to execute events belonging to different queues, resolving the hanging situation and re-enabling execution.

Overall, conservative synchronization is an approach that presents pros and cons: *i)* it is easy to program since the level of complexity is reduced by the fact that out of timestamp-order is explicitly avoided, *ii)* it is said to be *riskless*, in the sense that no incorrect data is produced at any instant

of time, letting the observed results to be delivered without the need of any inference, *iii)* it is not meant to effectively exploit parallel and distributed architectures as, to provide consistency, execution needs to present many serialization points, and processing is not meant to be really partitioned.

Optimistic synchronization

The second technique used to synchronize event execution is the optimistic one, also known as the *Time Warp* technique. Time Warp is a paradigm originally presented in [Jef85], which is based on the idea that every logical process is free to execute events independently, regardless of the simulation time of other entities. However, while processing events, a logical process needs to be able to:

1. detect if an inconsistent situation is arising due to the delivery of a *straggler* event

2. restore a consistent situation from which execution can be restarted until the LVT of the *straggler* event is reached, and continue from there.

. Therefore, this method is optimistic in the sense that every logical process executes without any guarantee that the next event will be consistently processed: any event is *speculatively* scheduled, and if a wrong situation is generated, execution is rolled back at a consistent point in the LP's simulation trajectory, from which it is restarting regular events' processing.

To better explain how such an approach works, we depict in Figure 2.8 a possible scenario in which consistency is optimistically enforced. In particular, LP_x is scheduling an event with timestamp 9, which is smaller than the timestamp of the last executed event at LP_k (namely, 15). In fact, LP_k is free to go ahead independently in its trajectory, as opposed to the

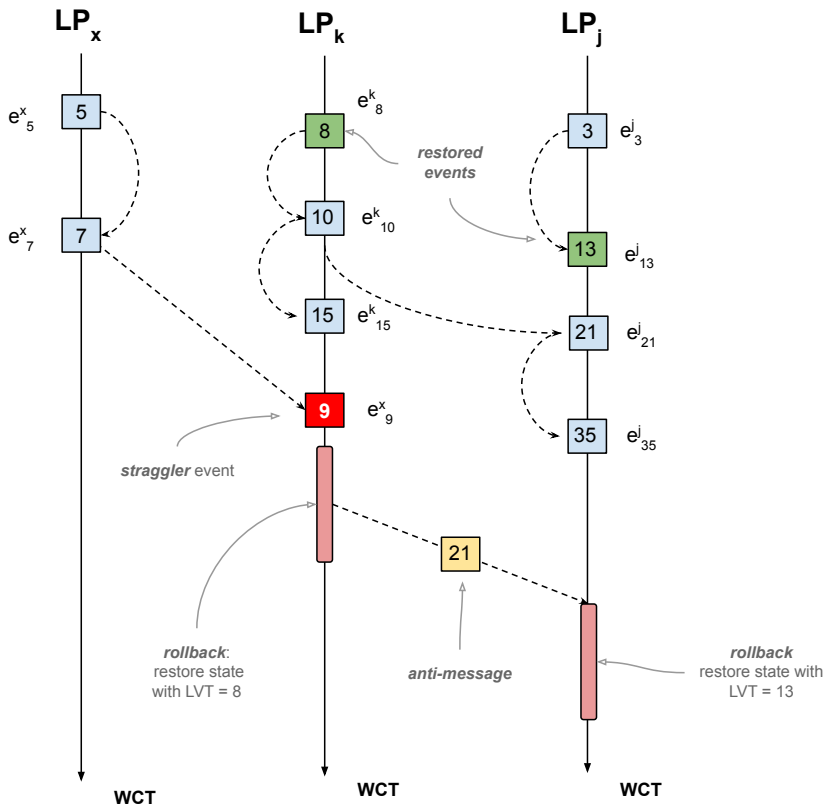


FIGURE 2.8: *An example of optimistic synchronization approach.*

previous conservative synchronization approach. However, LP_k needs to restore a consistent situation by bringing back the execution to the most recent event with a timestamp lower than 9. Note, however, that rolling back to a previous state could lead to inconsistencies at other logical processes too. In fact, by restoring a state preceding event e_9^x, LP_k needs to somehow "undo" the scheduling of event e_{21}^j at LP_j, notifying it that this event has been canceled. In this case, LP_j needs to roll back too since it executed events bringing its execution to an inconsistent situation: LP_j is influenced by the changes event e_{21}^j has generated, thus being it canceled at LP_k (which is going back to LVT 8), the first consistent event from which to restart computation is e_{13}^j.

This is the reason why LP_k is sending an *antimessage* to LP_j. This kind

of message is associated with a positive event that needs to be canceled. In fact, its purpose is to annihilate the corresponding positive event both in case it has been already executed or not. In the first case, a rollback would be necessary, but in the latter, the event can be simply discarded from the event list.

As far as the rollback operation is concerned, it is clear that it could potentially cause a drop in performance. To restore a consistent situation, we need to have a causal-consistent state to restart the execution from. Moreover, in cases where a chain of many logical processes is involved in event scheduling, a domino effect on a rollback at the root LP could trigger a pathological situation in which a considerable number of cascade rollbacks is generated.

This being considered, there are many works in the literature (e.g., [BCJL13], [BJCH09] and [GFS93]) proving that such a kind of approach can effectively make use of multi-core/distributed architectures. Indeed, it is not hard to understand that letting LP not synchronize with each other on the next event to be scheduled and designing the data structures with the highest possible degree of data separation would lead to speedups caused by parallelization of the workload. Moreover, it is a common practice to provide optimistic inconsistencies handling as a feature of runtime environments. Thus, the model designer doesn't need to care about consistency, and possibly dangerous situations are automatically detected and managed.

Finally, although rollback strategies will be thoroughly discussed later in this thesis, it is essential to mention that most Time Warp strategies need to deal with memory management while executing. In fact, when checkpoint policies are put in place, it is crucial to make logical processes agree on a moment in simulation time before which execution is considered committed, and thus memory dedicated to state saving can be safely released.

This particular point in time is called *Global Virtual Time* (GVT), and the calculation of such a value is required to be precisely evaluated. On the one hand, if the computed value is too large, the runtime tends to lose the benefits associated with optimistic synchronization, draining too many resources from the underlying architecture. On the other hand, if it is too small, the GVT calculation will be consequently triggered too often, falling into a situation in which computational costs related to synchronization do not represent an advantage with respect to conservative approaches.

Note that computing the GVT in a parallel environment, or even worse in a distributed cluster, can be a really challenging task: this is the reason why many works in the literature provide a solution both to speed up and make this process more accessible to developers. The reader can find an exhaustive discussion of GVT calculation in [Bel90], [LL90] and [TPQ+17].

Hybrid synchronization

The last methodology we introduce concerning LPs synchronization is a mixture of the previously discussed ones. Indeed, although both the previously mentioned solutions provide techniques to overcome problems related to causal consistency, they also introduce some limitations in terms of performance or in terms of ease of use.

Therefore, the idea behind adopting a hybrid approach lies in the possibility of mitigating the drawbacks of both approaches and trying to use this revised version at runtime. A more in-depth description of this technique is shown in [RAT93], and it mainly focuses on both tuning and mixing the following features:

Constrained optimism: aimed at limiting, as far as possible, the increased amount of memory usage typical of Time Warp systems, thus finding a procedure to garbage-collect memory areas which are safe

to be freed. As hinted before, this is related to the computation of a *committment horizon*, namely a GVT, and let the optimistic execution run only in a time window spanning between two different (and consecutive) GVT values.

Constrained conservativeness: which has the objective of avoiding the system to be completely conservative. For instance, LPs can be allowed to execute events speculatively only locally, not involving other LPs with optimistic scheduling of events. This method could avoid the generation of cascade rollbacks situations, reducing the aggressiveness typical of speculative approaches.

2.2.2 State saving & restore

Simulations based on the Time Warp protocol are gaining interest in literature for their ability to exploit nowadays multi-core/distributed platforms. However, to restore consistent situations, rollback management strategies need to be put in place.

The most used approach to cope with rollbacks in time-warp simulations is called *state saving & restore* and, as the name hints, basically consists in taking a snapshot of LPs' simulation states to be eventually restored if violations are detected. The snapshot needs to include all the necessary information to restart execution from the specific point in which they were taken, thus they may involve timestamps, variables, data structures, and so on. Depending on the frequency of snapshots and the amount of data saved in them we move across three main different solutions: *Copy State Saving*, *Sparse State Saving* and *Incremental State Saving*.

Copy State Saving

The most trivial implementation of the *State Saving & Restore* technique relies on taking a snapshot of the state of LPs involved in a simulation run as soon as an event is about to be processed. The snapshot is marked with the timestamp of the event which is going to be processed, in such a way that if an event generating an inconsistency occurs, the simulation platform only needs to come back to the consistent state matching the highest timestamp among the ones strictly lower than the straggler event's one. In other words, denoting with $T_{straggler}$ the timestamp of the event creating the inconsistency, and with $T_{rollback}$ the simulation time at which the execution is restored, there will always be a state to be restored, marked with timestamp $T_{restore}$, such that $T_{restore} \leq T_{rollback} < T_{straggler}$. Obviously, this is a technique that incurs a high overhead since the ease of this approach is compensated by the fact that the number of events executed during a simulation run can be huge, and logging a checkpoint before each of them would entail a great demand in terms of computational resources and of memory. In turn, this requires executing garbage collection much more often to reduce the memory currently in use: although necessary, *fossil collection* (and GVT computation) is known to be a costly operation. Figure 2.9 shows an example of such a kind of methodology.

Sparse State Saving

Sparse State Saving (SSS) is a technique that aims at overcoming the limitations of the CSS methodology, mainly reducing the frequency at which snapshots are taken in order to minimize both memory and CPU usage for reconstruction and *fossil collection*. The idea behind SSS is that it could be not necessary to save a snapshot before each event to be executed so that periodically keeping a checkpoint (according to some heuristic) can reduce

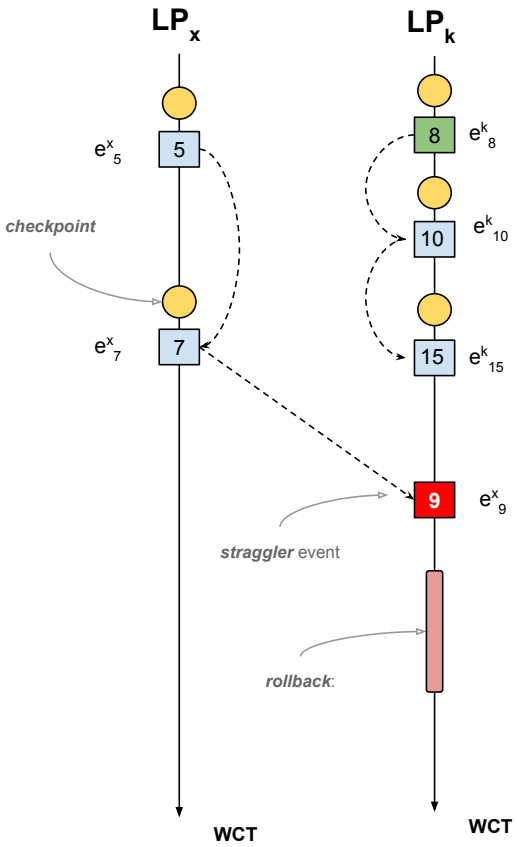


FIGURE 2.9: *An example of the copy state saving technique*

the costs mentioned above. Therefore, in this scenario, if a rollback occurs at time $T_{rollback}$, then either of the following two situations can arise:

1. A checkpoint of simulation state with timestamp $T_{snapshot} = T_{rollback}$ exists. In this case, $T_{snapshot}$ is simply restored. Note that this is an edge case in which restoration is carried on exactly as in CSS.

2. No snapshot such that $T_{snapshot} = T_{rollback}$ exists, and the state snapshot to be restored is the one associated with the higher LVT among those having timestamp lower than $T_{rollback}$.

If we define as *rollback length* the number of events which are undone by a rollback procedure, we can consider the average of this value a good measure of the overall amount of wasted operations performed during an opti-

mistic simulation run. Therefore, reducing the number of available snapshots would mean increasing the rollback length, as no guarantee is ensured on the time at which a snapshot is taken. Indeed, it could be the case that the last snapshot dates back to a very old timestamp with respect to the time at which the inconsistency occurs. If the tradeoff between saving/restoring an older state and taking a reduced amount of snapshots is acceptable, this represents a viable solution. Whenever the second of the two situations listed above occurs, the simulation platform also needs to re-execute some events in order to re-align the clock of the rolling back LP to the state at $T_{rollback}$. This operation is called *coasting forward*, and needs to deal also with the processing of events scheduled for the execution for other LPs. In particular, it is often the case that the *coasting forward* re-processing entails a *silent execution*, a special kind of event execution that avoids sending events to LPs if restoring the state due to a rollback. Indeed, sending events to other LPs in this phase could lead to simulation errors, as those events were already sent during *forward processing* before the occurrence of the rollback.

To better understand the situation we described, let's consider the example depicted in Figure 2.10. Here, LP_k receives a straggler message with timestamp 19, being $LVT_k = 24$. Considering that the adopted policy is taking snapshots every three events, LP_k needs to roll back to the event marked with $T_e = 8$, as this is the event with a timestamp associated with the latest state snapshot available. Thus, LP_k needs to re-execute events e^k_{11} and e^k_{17} before being able to handle the straggler event. Note that e^k_{17} originally sent a message to LP_x in order to make it execute an event with timestamp 14, namely e^x_{14}. In this case, while executing in *silent mode*, LP_k won't send this message again and will proceed with its standard *coasting forward*.

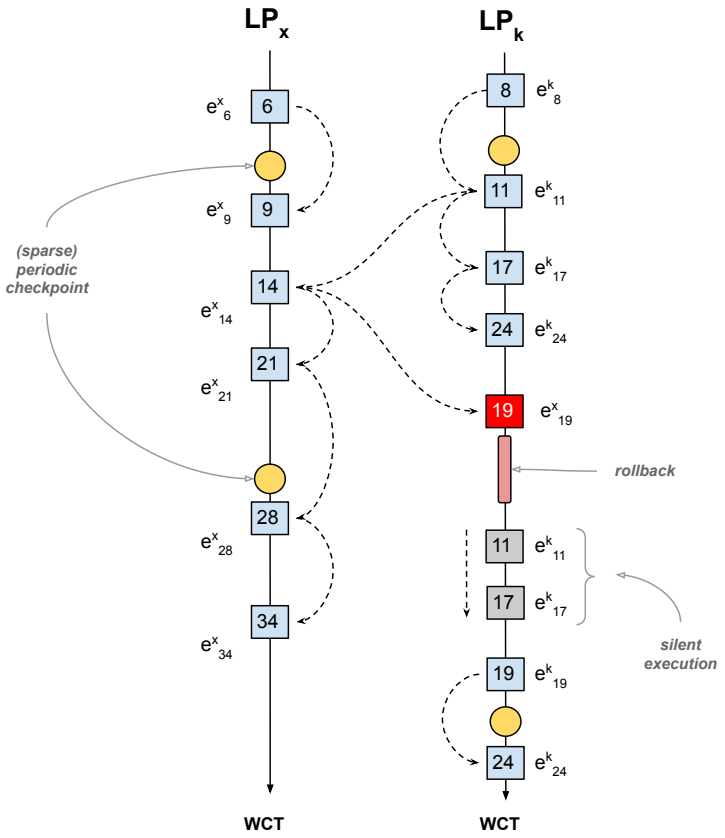


FIGURE 2.10: *An example of the sparse periodic state saving technique*

Periodic state saving suffers from the same performance issues of CSS, as it can be thought of as an expanded version of saving states right before each events' execution. For this reason, many *Adaptive State Saving* techniques were studied by literature, trying to fine-tune (at run-time) the value of the checkpointing period. Naturally, smaller checkpoint periods lead to memory inefficiency, while larger intervals would bring to performance decrease due to costly *coasting forward* executions. An example of approach which aims at computing the optimal value for the checkpointing interval is presented in [PW93b]: if we assume that the execution of events is non-preemptive and that the rollback length is independent of each other, then we can define

the optimal checkpointing interval as

$$\chi = \left\lceil \sqrt{\frac{2\delta_s}{\delta_c} + (\frac{N_c}{N_r} + \ \delta_r - 1)} \right\rceil \tag{2.3}$$

being:

- δ_s the average time required to save the state of the simulation
- δ_c the average time required to peform a *coasting forward* execution
- N_c the overall total number of committed events
- N_r the overall number of processed rollbacks
- δ_r the average length in time of a rollback

Although those parameters need to be computed at run-time via statistics evaluation, this technique can significantly increase performance related to rollback handling. Indeed, dynamically tuning the value of χ while executing events would adapt the costs associated with restorations to the requirements of the simulation, whether it is more prone to rollbacks or not according to the logic of the model.

Incremental State Saving

The last technique we introduce about the state saving methodologies tries to cope better with the problem of the state save that needs to be stored upon a checkpoint. Indeed, even if the frequency of checkpoints is reduced, the size of a state S_x to be saved/restored could impact performance, especially when only a small portion of this state is updated between a series of events. In this specific case, a non-minimal amount of time and space would be consumed for possibly redundant information.

The idea behind *Incremental State Saving* (ISS) is that when an event is executed, only the state variables it touches are eligible to be persisted in a checkpoint. Therefore, only the *delta* between two states (namely, the difference in modified variables/structures) is kept. Upon a rollback, the simulation platform scans through all these (usually tiny) states to reconstruct a more significant state where to restore normal execution from.

Depending on the transparency offered to the final user, we move from less to more sophisticated techniques. To make an example, the solution in [RLAM96] provides the possibility to modify the simulation kernels' data structures and to overload functions dedicated to event processing. This way, leveraging Object-Oriented overloading, model developers can enhance state variables to mark them as part of an incremental state. By invoking overloaded functions, they are therefore allowed to use such a fine-grained restoration of the overall state. The highest level of transparency is reached when the user is not required to write any additional code and is not even aware (at least by reading the application-level code) that such a policy is running underneath. An example of such a more sophisticated implementation of ISS is presented in [WP96]. Here, transparent incremental state saving is provided by using software instrumentation that automatically detects instructions updating simulation state and performs a copy of this cumulative data. Given that this operation is conducted at the level of assembly instructions (although only targeting x86 architectures), no user interaction is required at all. The very first work related to this topic was presented in [BS93], while more innovative approaches can be found in [PVQ15].

2.2.3 Reverse computation

While the *State Saving & Restore* can be considered the most spread technique when dealing with Time Warp consistency issues, other methodologies are tackling the same problem in a completely different way. One of them is the so-called *Reverse Computation*, which is trying to eliminate, as far as possible, the usage of state checkpoints. As the name hints, this method is based on computing in a reverse way the computation associated with events.

Given an event e_i, *Reverse Computation* will try to revert all the changes e_i possibly made on the simulation state S_i by performing, in reverse order, all the actions involved in the execution of e_i. If the considered rollback involved multiple events, this "undo" operation needs to be performed on each of these events, from the rollback restoration to the straggler's timestamp.

```
1 if(condition > 0){
2   condition--;
3   counter++;
4 }
```

LISTING 2.1: *Example of forward execution of an event*

To make an example, let's consider the code snippet presented in 2.1. Here, a simple `if` branch is checking the current value of an integer variable, then this value is decremented, and a counter is increased to remember some sort of information of the variable itself. Computing these operations in reverse order would entail triggering an anti-event with the code presented in 2.2 as associated logic. As it can be noticed, undoing arithmetic operations is quite straightforward, as we simply do the opposite operation that the original event performed. However, when dealing with the branching conditions, some issues arise since we need to check the old value of the involved variable (hence the *was* keyword in 2.2). This information isn't available

while executing the anti-event, as we don't remember the state of internal variables in this scenario.

```
if(condition "was" 0){
  counter--;
  condition++;
}
```

LISTING 2.2: *Example of reverse execution of an event*

For this reason, the introduction of a *bit variable* turns out to be essential to query the state of the variable checked in the branching condition. Note that this variable should be transparently added by the run-time implementing the *Reverse Computation* technique, as it shouldn't be the programmers' responsibility to take care of logic associated with rollback's restoration. Thus, the original code snippet would be transformed to the one presented in 2.3, and when backward-executing the events involved in a rollback, the *anti-event*s will exploit the value of the inserted variable to infer the actual branch condition outcome (as presented in 2.4).

```
if(condition > 0){
  entered = 1; //bit variable: the branch was entered
  condition--;
  counter++;
}
```

LISTING 2.3: *Example of instrumented code exploiting bit variables*

```
if(entered){
  counter--;
  condition++;
}
```

LISTING 2.4: *Example of reversed code exploiting instrumentation*

While this solution completely avoids the procedure of storing and restoring checkpoints, it comes with a series of issues that need to be faced. First of all, the condition presented in the code listings above hints that LP's

state could increase in a non-negligible fashion depending on the operations related to an event's execution. For instance, if the number of branches involved in the logic associated with an LP's events is n, then the number of loaded variables (namely, *bit variables*) would sum up to $log_2(n)$. The same reasoning can be done for `while` statements (or loops, in general), as we should maintain the number of iterations in order to re-execute them in the opposite order while rolling back.

Secondly, note that not all operations can be voided while undoing portions of code. Indeed, a particular class of instructions, called *disruptive operations* must be handled via state saving techniques, as they produce changes in the system which are not reversible. For instance, assignments (`=`) or addition assignments (`=+`) can't be simply reversed; they need to keep a previous state. Moreover, concerning reversible code, a note on jump instructions has to be made. In fact, operations like `goto`, `continue` or `break` could be non trivial to reverse, as the actual execution flow needs to be supported by a set of *bit variables* in order to remember the steps attempted during forward execution. In this case, the reverse computation logic could entail the usage of `switch/case` constructs in order to be able to reproduce the same original flow, inverted. Unfortunately, the amount of generated code could reach an important size for such a situation, ultimately enlarging the overall LP's state.

Finally, the *Reverse Computation* technique can represent a viable solution when the checkpointing operation and the costs associated with it are considered unfeasible for some simulation scenarios. However, the more the number of *disruptive operations*, the more this technique falls back to the state saving one. Thus, the tradeoff needs to be thoroughly analyzed. Although *Reverse Computation* isn't the technique adopted for the results of this thesis, many works in the literature provide a complete explanation

of this methodology (e.g., [CP99])

2.3 Agent-Based modeling

Agent-Based Modeling is considered a powerful simulation technique and has recently been employed to cope with many real-world business applications.

An Agent-Based Modeling System (ABMS) is represented by a collection of autonomous decision-making entities, namely, the agents. At the most superficial level, an agent is a physical and/or virtual actor that perceives the surrounding environment and acts according to specific rules via actuators. Defining an agent-based model consists of describing a system from the perspective of the interaction between agents. Although agents are considered independent entities, one of the most distinctive features of ABMS is that they model interactions among agents and with the environment: changes of behaviour can influence the structure of the area they're currently traversing, or either effectively alter the behaviour of other entities.

Therefore, as will be analyzed in chapter 6, modelling interactions between agents entails the specification of *i)* the strategies used for communicating between entities and *ii)* the definition of the linkage between agents in terms of possibility to cooperate. The latter specification ultimately refers to the definition of an environment that governs the possible movements of the agents. On the other hand, strategies for communications control the transmission of information between entities. Indeed, agents typically have limited visibility of the whole system; hence, the exchange of information between them takes place locally, allowing for a more realistic depiction of complex real-world scenarios. Consequently, an ABMS environment is the

area in which agents live and interact according to the topology and the set of rules defined at the model level. Depending on the situation of interest, ABMS can provide many advantages with respect to other simulation paradigms, such as:

Flexibility: An agent-based model can be considered flexible from many points of view. First of all, the simulation scale can be easily tuned by adding or shrinking the number of entities participating in the run. Secondly, even the agents' complexity can be modified to customize their behaviours. Sophisticated models define precise rules managing agents' degree of rationality, ability to learn, dynamically change interactions, etc. Moreover, ABMS provides flexibility on a global scope concerning agents: not only their complexity can be augmented, but also the aggregation of multiple entities can lead to complex group-based actions.

Easier model design: For many real-world scenarios, ABM provides a more natural way of describing the system in terms of defining rules and evolutions of it. To make an example, it results to be more reasonable to model how cars move inside a city area rather than define complex equations governing the dynamics of the density of cars at a specific place. Additionally, since the movements of vehicles depend on drivers' behaviour, ABM allows studying even aggregate properties related to traffic jams. While traffic jam is just a typical example of an ABM system, it is easy to understand that this level of simplicity allows tackling critical real-world situations more easily, which is favorable when results need to be provided with strict time constraints, as happened during the new coronavirus outbreak.

Capturing emergent behaviour: Emergent behaviours are phenom-

ena resulting from the interaction between individual agents. Interestingly, an emergent behaviour may present properties that are decoupled from independent single interaction specifications. In other words, emergent behaviour isn't simply the *sum* of the interactions that lead to its birth. Considering the example of traffic jams, the direction of movements of cars may be inverted due to interactions between individual vehicle drivers. Therefore, emergent behaviours are challenging to predict, as they can be counterintuitive for human beings. However, agent-based modelling is, by definition, a viable approach to understand emergent phenomena: by defining the set of rules a unit can perform and the interactions it can produce, the capturing of the emergent behaviour is a natural consequence of the whole simulation run.

The applications of ABMS are multiple and cover many real-world and scientific areas, such as evacuation strategies ([SRC09]), flow management (i.e., traffic simulation, [YEGL14]), disease spreading ([MSGNCG+15]), disaster recovery ([Fie06]), market forecasts ([PAH+94]) and many others.

2.4 The technical context

In this section, we describe the technical context used to gather results presented in this thesis. In particular, all the experiments discussed in the remainder of the thesis have been carried on making use of an open-source, multithreaded, distributed and general-purpose simulation kernel called the **RO**me **O**p**T**imistic **Sim**ulator (ROOT-Sim). This platform was introduced in 1987 by the *High Performance and Dependable Computing Systems* (HPDCS) research group born in Sapienza, University of Rome, which also expanded to Tor Vergata University of Rome. Basically, the

codebase is written following the C/POSIX standards, targeting x86-64 architectures, and supports the Time Warp protocol described in previous sections. ROOT-Sim supports almost all of the aforementioned features related to PDES, and its architecure is based on a series of building blocks:

Event scheduler: having a potentially high number of LPs, the runtime needs to select at any point in time which of them is allowed to execute next. Thus, an ordering among all the LPs is provided in the simulation kernel, and many policies are actually available. The default one, which is the most common, is the *Smallest Timestamp First* one, which activates the LP whose next event in its queue presents the nearest timestamp with respect to the current time, among all (local) LPs.

Committed and Consistent State Manager: while executing events, the simulator needs to understand whether the run should be halted or not. This is typically done by evaluating some conditions either defined at user level, such as a certain situation reached in the simulated phenomenon, or at simulation kernel level, e.g. reaching a specific virtual time or a certain number of executed events.

GVT Manager: evaluating an instant in time before which computation is considered completed results to be a core operation for memory release policies. The GVT manager takes care of periodically evaluating this value, according to the supported implemented algorithm.

Memory Manager: dealing with memory allocations and releases and the management of memory areas in rollback scenarios is considered a core feature for a time-warp-based simulator. The memory allocator that ROOT-Sim integrates is called Dynamic Memory Allocator (DyMeLoR), which basically wraps the well-known `malloc/free`

standard C functions to perform operations aimed at dealing with simulation-specific concerns. In particular, DyMeloR essentially assigns equally-sized amounts of memory whenever an allocation request is issued, by rounding up the requested size to the nearest power of 2. To keep track of which memory chunk an LP is currently using, a per-LP meta-data structure is maintained. This data structure, named `malloc_area` is fundamentally storing bitmasks memorizing the information about a block of chunks, namely, whether the chunk was dirtied or whether it is currently delivered to the application for usage. An example of a DyMeLoR's memory area is depicted in Figure 2.11, and for more details, we refer to [PVQ09].

Since ROOT-Sim supports multithreading, parallel, and distributed computing, the paradigm this simulation platform relies on is the *worker thread* one, in which every logical process is controlled by a worker thread and is assigned two unique identifiers: the *Local Identifier* (LID), used to discriminate a certain LP on the same machine it is currently residing, and the *Global Identifier* (GID), which is instead determining the identity of an LP on a distributed scale. Moreover, even threads are identified by a unique identifier, namely *Thread Identifier* (TID). It is the responsibility of the simulation kernel to manage their data structures and handle inter/intra machine communications.

The interaction with application-level code is made possible by exposing two callback functions, `ProcessEvent` and `OnGVT` and a function call, `ScheduleNewEvent`. The logic associated with these entry points is described in the following:

- `ScheduleNewEvent(int receiver, time_type timestamp, int event_type, void *event_content, int event_size)`: this is the entry point for a model developer to inject an event within

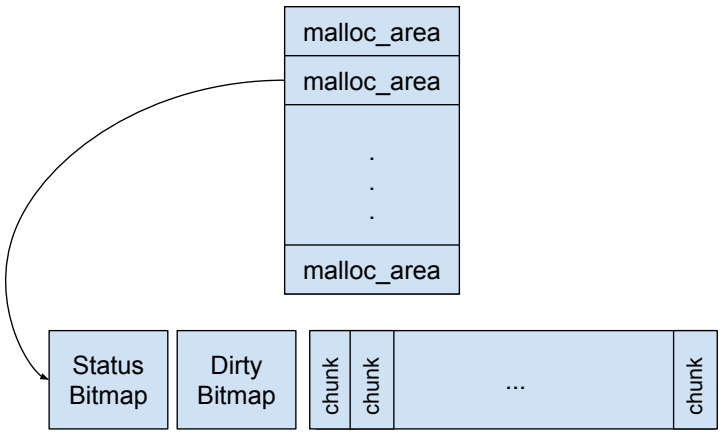


FIGURE 2.11: *The ROOT-Sim* `memory_area` *structure details*

the system. Indeed, a new event characterized by the passed `timestamp`, `event_type`, `event_content` and `event_size` is generated and inserted into the event queue of the LP having the global identifier matching `receiver`. Note that the invocation of this method doesn't trigger the immediate execution of this event. Still, it will be eventually scheduled for execution whenever the conditions will be met (e.g., it is the next event with the smallest timestamp). At the same time, the correctness of the whole execution is guaranteed since events inserted in input queues are ordered according to their timestamp.

- `ProcessEvent(int me, time_type now, int event_type, void *event_content, void *state)`:
  this callback allows processing of simulation events at application level. The contract of this function is that all its content is executed by the simulation platform speculatively, meaning that it could be potentially undone, and even events generated inside this scope could be annihilated. However, the programmer doesn't need to cope with issues related to this: everything will be handled, under the hood, by the ROOT-Sim platform, and she only needs to implement state transitions according to the logic of the model. Although this represents

an interesting feature, there are some non-rollbackable actions that the model developer should be aware of. For instance, printing on standard output (via standard calls like `printf`) is a non-cancelable operation. Here, the passed parameters concern both the LP running the event, identified by `me`, and having a state `state`, and the event to be processed, with type `event_type` and enveloping `event_content`. Also, the `now` variable can be used to read the current logical virtual time of the LP.

- `OnGVT(void *snapshot, int gid)`:
  whenever a *committment horizon* is reached by the simulation platform, this callback is invoked. This allows the model developer to introduce custom logic over the committed snapshot of the simulation, for example evaluating a termination condition (users can easily notify the request of termination to the ROOT-Sim platform by returning `true`). This, in turn, implies that the code executed within this scope is not speculative. The final result is that this callback can also be exploited to collect statistics, as I/O operations would be correctly performed. Also, note that invocations of `ScheduleNewEvent` are not allowed within the boundaries of this callback since they could involve the generation of rollbacks of already committed events (i.e., before the computed GVT value). As for the parameters, `snapshot` represents the portion S_x of the CCGS state S, while `gid` is the (global) identifier of the current LP.

In Figure 2.12, we show a schematization of the building blocks of the ROOT-Sim platform, describing the interconnections between multiple submodels and the available APIs enabling user interaction with underlying facilities. For additional details about this open-source[2] project, the reader

[2]https://root-sim.github.io/

can refer to [PQ13].

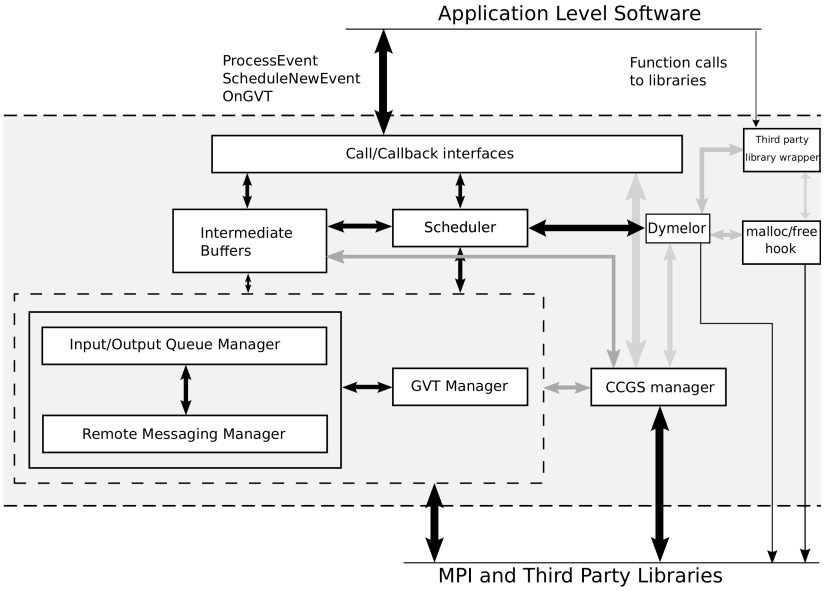


FIGURE 2.12: *The ROOT-Sim platform building blocks scheme*

2.5 Attained results

This thesis explicitly targets PDES platforms in order to provide software tools and methodologies aimed at

- reducing the complexity of programming on top of parallel and distributed clusters.
- allowing, through direct interventions, to tune the underlying system to reach the required level of performance, yet with a high degree of freedom in customization.

This reflects both in theoretical and practical solutions that allow the scientists from any area to write simulation models since the provided abstractions are already coping with the efficient exploitation of architectural resources.

The solutions proposed in this thesis were built relying on the ANSI-C programming language, primarily to enable the model writer to have the maximum level of freedom in developing code and control of any portion of memory. Note, however, that by any means this impacts the theoretical reasoning associated with the provided software tools: they are applicable to any programming language and could be re-implemented in different, custom environments.

More in detail, Chapter 4 introduces an innovative memory management subsystem that permits writing sequential style code while accessing memory areas belonging to different LPs consistently while respecting the PDES principles and exploiting parallelism and distribution on a cluster architecture. In particular, such a solution allows accessing remote memory portions with ease and provides optimizations that autonomically adapt to the current memory access patterns that the environment is experiencing.

Chapter 5 focuses on the cost reduction of critical operations in speculative PDES environments, namely rollbacks and state restorations. The Time Warp protocol was selected in the implementation of the simulation platform of interest, as it provides efficient usage of multi-many core (distributed) machines. However, this implies the generation of inconsistent trajectories, which involve synchronization points and restoration of consistent situations. The solution proposed in Chapter 5 proposes a shift with respect to the traditional options offered by literature: giving the possibility to the modeler to select the memory areas that are required to be restored and the ones that can be discarded (for specific simulation time windows), we proved that efficiency could be improved while increasing the level of freedom of the final programmer.

Finally, Chapter 6 presents a set of semantically rich APIs allowing to implement Agent-Based Models in a simple and effective way. What distinguishes these APIs from existing methodologies is that it enables the possibility to benefit from more effective runtime paradigms, such as speculative PDES systems. The discussed experimental assessment shows how our proposal allows implementing complicated interactions with reduced complexity while delivering a non-negligible performance increase.

CHAPTER 3

Literature overview

Look wide, and even when you think you are looking wide – look wider still.

— Robert Baden-Powell

In this Chapter, we will present a comprehensive argumentation about the literature related to the solutions that are introduced in this thesis. For all of them, a specific literature analysis will be carried on, in order to have a broader understanding of the improvements that this dissertation brings to the scientific area it belongs.

3.1 Cloud-based distributed simulation

In recent years, research in the field of PDES involved a lot of effort into enabling simulation applications to fruitfully exploit cloud-based, or more generally virtualization, technologies. Many works, such as [YP13], have been carried out with the final scope of evaluating the effects of hypervisor configurations over the model-execution dynamics. These proposals have taken into consideration the presented differentiated synchronization schemes, namely conservative or optimistic ([DM14], [YP13], and [YP15]): they now stand as baseline assessment of whether cloud platforms can be considered beneficial for complex and large scale PDES applications.

Clearly, making a wise use of distributed/virtualized resources was a

central target of the results presented in this thesis, and for this reason a thorough discussion about these reference works has to be put in place. Indeed, one of the main results that will be presented in the remainder of this thesis is related to a novel run-time middleware enabling PDES applications to be run on top of distributed (virtual) machines in the cloud, enabling in-place accesses to the state of any LP by any event handler, rather than being limited to a single shared-memory machine as in [PQ19]. This represents an objective fully orthogonal to (and of similar relevance of) the one pursued by the aforementioned literature works. We refer to this solution as the distributed version of the Event & Cross-State protocol. As stated, the main goals of this solution were: *i)* provide a fruitful exploitation of clusters of distributed/virtualized resources in the context of PDES applications *ii)* ensure ease of use by enforcing transparency to the model developers, eliminating the burden of taking care of PDES-specific (or even distributed computing) related issues.

The literature offers few solutions oriented to enabling some form of data sharing across logical processes through the enrichment of programming facilities. To make an example, the proposal in [Bru95] discusses how state sharing can be emulated by using a certain LP hosting the common data and acting as a centralized server. By introducing the concept of version records, multi-versioning is used for shared data maintenance in order to handle read/write operations occurring at different logical times, and to avoid, in case optimistic synchronization is adopted, unnecessary rollbacks of the "centralized server" LP. A similar solution can be found in [MH95], which provides a theoretical presentation of algorithms to implement a Distributed Shared Memory mechanism, having as main purpose keeping replicated instances of a variable coherent. Again, one of these algorithms is based on managing variables as multi-version lists, where write operations

install new versions and read operations retrieve the most updated one. All the cited literature proposals differ from the solution we present for two main reasons:

- shared variables' read/write operations are realized via message passing, namely, event schedule operations. Thus, they are taking place at the application level, while we support in-place access to any (by default shareable) buffer within the logical process' state via standard programming constructs, i.e., pointers. As a matter of facts, the retrieval of the actual memory portions towards which the accesses are finally carried out is completely transparent to the application and is demanded from the novel distributed ECS middleware.

- LP memory sharing is not limited to a portion of the state image of a specific LP (like the centralized server mentioned above). Indeed, we enable shared accesses to any memory buffer representing a portion of the whole simulation model state. At the same time, the presented solution supports distributed deployments of PDES systems, entailing such sharing features and overcoming the limitations of the original ECS run-time support [PQ19]. In other words, this enhancement overtakes the original confinement to a single shared-memory machine.

Stressing the first of the these two points, [G91] integrates the support for shared state in terms of global variables, basing the architecture on [CS89]. Conversely, in [FD97], the notion of state query is introduced, according to which a logical process requiring to access a portion of the state belonging to a different LP can issue a *query message* to it and then waits for a reply containing the proper payload. If this value is later detected to be no longer valid, an anti-message is sent, which invalidates the query. However, neither of the two proposals provides transparency: although the

first supports in-place read/write operations as distributed ECS does, the application-level code must explicitly register a logical process as a reader/writer on shared variables. On the other hand, the second one relies on message passing, and the application programmer needs to embed the usage of query messages within the application code. Also, [G91] does not scale to distributed memory clusters. The proposal presented in this thesis avoids all these limitations, by also allowing the sharing of dynamically-allocated buffers within the object state, for which pre-declaration of the potential need to access cannot be raised at startup—in fact our solution allows coping with scenarios where the actual accesses depend, in an unpredictable way, on the specific model-execution trajectory.

The work in [CLY+11] proposes a framework targeted at multi-core machines and based on Time Warp synchronization ([Jef85]), where so called Extended Logical Processes (Ex-LPs), which are essentially collections of LPs, have public attributes associated with variables that other LPs, belonging to different Ex-LPs, are free to access. The accesses to shared attributes is envisaged by relying on a specific Transactional Memory (TM) implementation, where events are mapped to transactions and the implementation of the TM layer is based on [G91]. Once again, the proposal in [CLY+11] requires a-priori knowledge of the attributes to be shared, which need to be mapped to TM-managed memory locations manually. Rather, our proposal allows for sharing any memory area from the heap, without the need for a-priori knowledge of whether some sharing can (or will) occur along model-execution; this increases the level of transparency. Further, as a second core difference, the work in [CLY+11] does not support cross-LP accesses on distributed memory systems, which is instead the primary target of our work—according to the aim of enabling the exploitation of clusters of resources.

Considering the context of agent-based modeling and simulation, RepastHPC, outlined in [GIOY16], shares a few low level architectural concepts with the ECS middleware, since it includes the support for dynamic data (in this case, agents) migration. However, in RepastHPC the objective of migration is essentially to increase locality in the access to data (by agents), while our ECS middleware has the objective to transparently allow in-place access to whatever data belonging to the simulation model state on distributed memory platforms, while still enabling speculative processing of simulation events.

Still in the area of ABM, similar considerations can be made for PDES-MAS [LLT08]. It is based on sharing data across logical processes: specifically, LPs modeling agents—by relying on given objects hosting portions of the shared data. This is different from what our ECS middleware enables since in PDES-MAS in-place accesses are not supported. On the other hand, PDES-MAS has orthogonal objectives with respect to our work, such as enabling scalable data sharing via optimized data partitioning/distribution [STL13] and explicit query mechanisms, like range queries [ST13].

In [PPQV16], a programming approach and its run-time support are presented, where shared data in PDES applications are allowed to be accessed by concurrent event handlers without the need to pre-declare the intention to access them, e.g. via code annotations. This has been achieved via user-transparent software instrumentation, in combination with a multi-version scheme, either allowing the redirection of read operations to the correct version of the target data—depending on the timestamp of the event being processed—or forcing rollbacks of causally inconsistent reads. This solution is targeted at the management of global variables. Instead, our proposal is suited for data sharing of dynamically allocated memory chunks logically incorporated within the state of each individual logical process, while still

providing parallelism and synchronization transparency. Moreover, the proposal in [PPQV16] is limited to shared memory machines, while our primary focus is to port ECS to distributed memory clusters.

The distributed ECS middleware can also be compared to approaches that bridge shared and distributed memory programming in more general contexts. Among them, we can mention Partitioned Global Address Space (PGAS) [Sti09], distributed shared memory systems [IS93], distributed file systems [Blo15] and distributed transactional memory systems [HNP+13], [RRCC10]. The uniqueness of the ECS protocol stands in the fact that our solution is already specialized for speculative PDES, while others that bridge shared and distributed memory programming would require the development of additional modules and software layers in order to accomplish the same objective, possibly relying on the approach presented in this thesis. Specifically, these solutions do not cope with virtual time-based speculative synchronization, thus not enabling the local materialization of remote data versions complying with timestamp-ordered accesses—in fact these approaches are not able to directly support timestamp-based causality relations across concurrent tasks touching data in read/write mode.

Although belonging to a different area, we can also mention a relation to Massive Multi-Player On-line Gaming (MMPOG) architectures, especially the ones based on data sharing via proxies [MGSF07]. More in details, in our ECS middleware we enable migrating the state of a logical process across multiple computing nodes, depending on the cross-state accesses that occur. However, usually proxy-based MMPOGs are limited to the migration of predetermined slices of information: we directly overcome this restriction in our distributed protocol.

Generally speaking, it is to be noted that full state partitioning as in traditional PDES, with the event handler only accessing the state of a single

target LP, is a programming model whose run-time environments have been shown to be capable of good scaling. Indeed, this approach presents the capability of exploiting extreme scale distributed infrastructures and supercomputing oriented facilities, as stated in [BCJL13] and [CP10]. This can be seen as a reflection of the fact that state partitioning can lead to increased locality, a condition that as a consequence avoids bottlenecks related to the employment of a single centralized memory system.

However, enabling ECS-style coded models to be run on distributed memory systems allows to exploit differentiated classes of computing clusters. Offering the possibility to merge higher-end and bare-metal machines, in conjunction with the flexibility in the exposed programming model—which breaks disjointness in the accesses to the LP states by the event handler—provides an additional degree of freedom to the end users.

As will be discussed in the following chapters, the distributed ECS middleware allows performance scalability trends (over the number of used machines) well matching those provided by traditional PDES run-times. This is a core aspect enabling the effective exploitation of ECS on generic platforms. At the same time, an improvement of the data access locality when running with the ECS middleware could be achieved by adopting dynamic LPs' clustering schemes like, e.g., the one in [MNPQ16].

3.2 State saving optimization techniques

Traditionally, the problem of the cost reduction in managing state-rollback operation has been an hot topic in the field of simulation, and has been long studied in the literature.

To make a first example, a few solutions propose to exploit infrequent state snapshots to reduce the CPU-cost and memory footprint for logging

state information and to restore a missing snapshot by reloading a previous one and reprocessing intermediate events.

All these solutions are based on the optimization of both the costs associated with taking snapshots and with the rollback operation itself, performed via a temporary reprocessing phase. Many works, e.g. [PLM94], [Qua01], propose several methods to determine the best-suited policy to undertake when dealing with the frequency of the recording of the snapshots.

Other approaches, e.g. [RLAM96], [WP96], are based on taking incremental checkpoints, namely logging only the portions of the LP state that have been modified since the last checkpoint operation (i.e. delta changes). As stated in the previous Chapter (2.2.2), this approach can drastically reduce the CPU-cycles and memory footprint to create the checkpoints along the forward execution phase. However, the drawback of it stands in the rollback phase: it requires to backward apply all the incrementally logged checkpoints to enable the resume of the logical process execution from the correct past state. Sometimes, the length of the sequence of backward steps is unpredictably long, leading to negative performance effects.

To cope with this problem, some solutions [PVQ15], [SE98] have proposed mixed schemes where incremental and non-incremental checkpoints are taken in interleaved manner along the forward execution phase of a logical process.

When the reduction of the memory footprint associated with state saving is pushed to the limits, we find reverse computing approaches, which were also extensively investigated [CP99]. With the only exception of irreversible state changes (as discussed in section 2.2.3), with this kind of approach no logs of state information is required. However, for (very) long rollbacks the cost of backward reprocessing the events might be a dominating factor, ultimately hampering performance.

Again, mixing approaches came to the rescue: the work in [CPQ17] studied how to combine reverse computing with classical infrequent checkpoints, so as to minimize the number of backward-processed events in a rollback phase.

Furthermore, many works in the literature aim at optimizing the log/restore operations costs by exploiting some hybrid (combined) techniques or based on some model-specific characteristics (see, e.g., [SR96], [PW93a], [Qua99] and [Qua98]).

Anyway, all the aforementioned approaches share one main characteristic, which is the "perfect" reconstruction of the past (i.e., to be restored) state. In other words, this means that the restored state is an exact copy of the corresponding state observed along the forward execution phase, before the occurrence of the causality error (i.e., the occurrence of a *straggler* event).

What we will present in the following chapters proposes a different approach, in which reconstructing the state to be restored is performed in an approximated manner, hence investigating a new dimension in terms of the trade-off between the CPU-cost and the memory footprint to save state information, and the latency to restore the last correct state upon rolling back. We refer to such a solution with the name *Approximated Rollbacks*.

When dealing with transparency to the application programmer, we note that a few of the above cited solutions do not provide the support for it (or, at least, not completely, e.g. [PLM94], [Qua01], [RLAM96], [SE98]). We remark that this is one of the main goals of the solutions described in this dissertation, and this is the reason why we minimized the intervention of the programmer to the minimum possible: the concept of rollbacks is completely hidden, meaning that the programmer doesn't need to know even what a rollback is, thus not handling any associated logic at all. The

only required action is to include in the application code specific calls to an API that enables the approximated rollback scheme, implementing a callback function to exploit the platform-level state-saving tasks in order to approximately rebuild the state to be restored. Put differently, the model designer only needs to be aware of how the state of a logical process can be reconstructed starting from a minimal set of state information transparently managed (i.e., logged and restored) by the underlying PDES platform.

Our proposal is also related to solutions based on the notion of uncertainty in the occurrence of events [Fuj99], [QB04]. In these proposals, the PDES platform can take decisions aimed at reducing the execution time of the simulation by making use of a relaxed specification of either time or space for the occurrence of events.

The natural effect of such a choice can be a bias in the evolution of the simulation model trajectory, possibly leading to an approximation of the collected statistics—compared to a scenario where no relaxation is used and exploited by the run-time system. Although the innovative approach we present shares this kind of concept, a core difference in our solution is that the effects of the approximation can be somehow managed by the application programmer, since the execution phases where the approximated rollbacks can occur can be explicitly specified in the application logic.

Other proposals have studied the trade-off between relaxing strict causality of the events and its effects on performance in (speculative) PDES [IPGCMW18], [RTRW98]. Basically, the idea behind them is to skip running some state rollbacks if the effects of processing events out of strict timestamp order is considered acceptable in terms of the statistic outcome computed by running the simulation. Our solution can be seen as fully orthogonal to these approaches, since we still comply with strict timestamp ordering of the events at each individual LP. At the same time, we allow

a given event to observe a subset (i.e., an approximation) of the state that would have been observed in a rollback-free run of the same simulation model.

3.3 Effective agent-based PDES frameworks

Along the years, ABMS and PDES have been extensively investigated. For this reason, many attempts to support Agent-Based models into PDES platforms have already been carried out, and a large number of frameworks can be found in literature. Some of the most known ones are NetLogo [TW04], Mason [LCRP+05], RepastHPC [CN13], Swarm [Iba13], and JAMES II [HU09].

However, many of these solutions are not meant to wisely take advantage of future exascale computing infrastructure. Thus, effort has been put into devising new frameworks able to significantly scale up over the amount of available resources like, e.g., [LLT07], and [STL13]. For a more complete comparison of these (and many more) ABMS frameworks, we refer the reader to the comprehensive work in [ATLO17].

However, it is worth mentioning some proposals which are of interest with respect to the solution proposed in this thesis. Therefore, we provide in the following a somehow comprehensive list of similar frameworks, describing common points and enhancements this thesis brings:

MASON [LCRP+05] pays special attention to the performance of simulation execution, addressing computing-intensive models (i.e., scenarios with many agents), along with portability and reproducibility of the results across different hardware architectures. A parallel/distributed version (D-MASON) has been presented in [CDM+12], which relies on time-stepped synchronization and on the master/slave

paradigm. We similarly address the performance of agent-based simulation execution, yet we do this for the case of speculative asynchronous (non-time-stepped) PDES. In particular, we benefit from the performance improvement which can stem from the Time Warp synchronization protocol, while allowing a simple implementation of agent-based models via an expressive API.

Pandora [WRC12] is a C++-based simulation framework enabling executions in parallel/distributed environments. It features several AI algorithms for supporting agents' decision making and provides python bindings (which is a benefit for inexperienced programmers). At the same time, Pandora does not hide its internal complexity by design, allowing (and sometimes requiring) the model developer to extend its fundamental classes, just to perform simple actions. Conversely, in our proposal we offer the simulation model developer an API that is specifically tailored for implementing agent-based models, and which hides away all the idiosyncrasies of synchronization in a distributed simulation. This allows for a simplified implementation of simulation models, giving transparent access to highly optimized synchronization facilities to support efficient computations on clusters of modern parallel machines.

AnyLogic [BC18] is a commercial multi-method general-purpose simulation modeling and execution framework which can run simulations also in the cloud. AnyLogic allows users to spawn multiple simulations in parallel thanks to cloud resource provisioning. Recent versions of AnyLogic allow to deal with arbitrary topologies as we do, making more evident the importance of this aspect for modern simulation platforms. Anyhow, the ultimate goal of the AnyLogic framework is to scale out simulations, while our proposal is intended particularly

to increase the performance of single simulation runs.

FLAME [HCS06] is a simulation framework targeting large, complex models with large agent populations to be run on HPC platforms using MPI and OpenMP. The counterpart FLAME GPU [RR08] targets 3D simulations of complex systems with a multi-massive number of agents on GPU devices. We keep the ability to deal with large number of agents, yet we rely on traditional CPU-based execution of the simulation model.

RepastHPC [CN13] and **Swarm** [Iba13] are two ABMS run-time environments which have been successfully used to deliver high performance of agent-based models. These run-time environments support different programming languages, and allow agents to interact through the exchange of discrete events. Differently from our proposal, they require high programming skills to be effectively used. Therefore, they are commonly regarded as complex-to-use frameworks [ATLO17].

RAMSES [CPQ15] is an ABMS run-time environment, with a focus on transparency. An ABMS API has been already proposed in [CPQ15], with a goal similar in spirit to that of our proposal. The main differences between the two works are that: *i)* in [CPQ15], the API is based on the implementation of complex functions which are passed via pointers to the API, making it difficult to create bindings in different languages; *ii)* RAMSES supports only reversible computation, while the API which we propose can be implemented in both reverse computation-based and state saving-based run-time environments; *iii)* if an agent has to make decisions based on the state of the surrounding environment, this has to be implemented via explicit message passing; *iv)* dynamic topologies are not supported.

OpenABL [CPJ+18] is a recent proposal of a domain-specific language which allows to formulate agent-based models in a way which is independent of the actual hardware on which the simulations should be run. While the benefit of relying on domain-specific languages is undoubted to simplify the development process of a simulation model, we focus on run-time environments. In this sense, our proposal is complementary to that of OpenABL

3.4 The Event & Cross State approach

Since one of the solutions presented in this dissertation strongly relies on a previous work already present in literature, in this section we provide a thorough explanation of how the Event & Cross State synchronization scheme ([PQ19]) works, analyzing in depth the implementation choices and the benefits of such an approach. This in depth analysis will be preparatory for the explanation of the ECS enhancement, that will be outlined in details in Chapter 4. Indeed, some of the concepts introduced later in this thesis need to have a specific introduction to provide a complete comprehension of the achieved result.

3.4.1 Introduction

Historically, Parallel Discrete Event Simulation was mainly built upon an explicit partitioning of the whole simulation model into several Logical Processes, which concurrently proceed in their simulation trajectory [PPQV16]. In this organization, LPs' states are completely disjoint, and direct memory accesses are limited to the state of the LP which is currently in control of the event. To support such disjoint-memory access, simulation objects are required to communicate via message-exchange protocols. Nevertheless,

the development of many/multi-core and heterogeneous platforms increased the adoption of programming paradigms that could resemble the sequential style approach became progressively desirable. Indeed, the shared memory standard way of programming is generally preferred by a wide spread community of research experts which need to deal with simulation environments.

Given this context, the focus of this section is to introduce a highly flexible synchronization protocol for PDES named *Event and Cross-State* (ECS) Synchronization. With ECS, the developer is allowed to program handlers that are able to access memory locations contained in the state of any logical process through pointer dereferencing, thus being able to exploit the aforementioned implicit communication paradigm thanks to the use of a very spread methodology in modern programming languages.

The code portion shown in figure 3.1 presents an example of an event handler in a PDES model implemented using cross-state events to provide a better idea of the aforementioned solution. The input parameters to this function are well-known in the context of PDES coding conventions:

- The identifier of the LP (`obj`) for which event processing is taking place and the state base pointer (`state`)
- The address of the data structure starting from which the object can access any other dynamically allocated buffer belonging to its state, via pointers.
- The timestamp of the currently dispatched event.
- The event payload, containing all information necessary to perform the logic associated with it and provided by the model.

This code snippet can trigger a cross-state event since in-place pointer-based accesses happen both in write and read mode (lines 9 and 7, respec-

tively). In particular, the target memory portions involved in the accesses could belong to a simulation object that does not necessarily coincide with the one actually running the event. In other words, the LP that initially generated the event corresponding to the payload passed as a parameter could be different from the one currently executing this code portion (`obj`, in the snippet). Put more generally, a cross-state event e_i is a simulation event scheduled for a logical process LP_i with a corresponding target LP_j accessing in read/write mode the state of LP_i, with $i \neq j$.

```
1  void ProcessEvent(object_id obj, state_t* state, time_t event timestamp, event_payload t event) {
2
3    char* p, q;
4    ...
5    p = event.reference;
6
7    q = *p; // reading the content of memory location pointed by the 'reference' field in the event payload
8
9    *
10   p = q; // writing the content of memory location pointed by the 'reference' field in the event payload
11
12 }
```

FIGURE 3.1: *An example of cross-state event generation in the C programming language*

Hence, the ECS synchronization scheme provides *i)* a mechanism allowing transparent runtime detection of remote accesses; the system can automatically understand if cross-state access has taken place thanks to operating system facilities entirely transparently for the final programmer and *ii)* realignment of the state of the simulation object. Indeed, whenever a cross-state event occurs, a causality inconsistency can take place as the remote access may be at a different simulation time with respect to the timestamp of the original LP (i.e., the memory owner). This concern is addressed via the Time Warp synchronization protocol extension, which is finally named *ECS synchronization*.

The introduced programming model results to be innovative with respect to the solutions offered by the literature so far. Indeed, it offers the possibility to the programmer to only rely on pointers to the simulation

states in order to handle memory operations. Moreover, the code example depicted before shows how this solution doesn't require any further action from the final programmer: most of the newest programming languages rely on decorators (e.g., annotations) to enhance statements and mark them as particular kind to easily perform optimizations over them. In this case, instead, the programmer does not need to manually specify whether memory accesses refer to the state of a local or remote logical process: she could even don't know anything about the location of the involved logical processes (and, besides, this is most likely the case, as transparency is ensured throughout the simulation run).

This solution relies on pure runtime tracking of the occurrence of such a kind of accesses, while the PDES simulation is speculatively running. Again, the correct causal-consistent handling of memory accesses is fully transparent and hidden to the application developer, masking away any burden related to the possibility for two LPs targeted by a cross-state event to be contemporarily scheduled on many different threads. Furthermore, this implementation turns out to be inherently different from classical shared-memory approaches in which potentially common memory areas need to be manually defined. This is the case, for instance, of *Transactional Memory* (TM), which are offered, e.g., by the GCC compiler. In this case, it is easy to understand that the offered synchronized data access, which provides data consistency via isolation, is not enough to guarantee consistency among threads running events optimistically. Indeed, a cross-state event has the additional requirement to access simulation states according to the timestamp of the cross-state event, namely, the LVT of the LP processing the event. This clearly involves LPs state realignment operations related explicitly to PDES environments, a capability that TMs cannot provide.

3.4.2 The internals

Let us now discuss how the introduced synchronization scheme was implemented in an actual use case scenario. As a premise, it needs to be considered that even if this solution specifically targets Linux systems running on top of x86-64 architecture, the introduced concepts are general and can be ported on any other processor supporting multi-level paging (which turns out to be a very widely spread methodology in nowadays architectures).

In the ECS state management design, we define as a *stock* a bunch of virtual memory reserved to the use of a logical process, which can be employed to allocate buffers dedicated to its internal state managing variables useful to carry on the simulation. Specifically, whenever an LP demands new memory areas, the ECS memory manager assigns a block of page-aligned virtual memory addresses (thus, the name stock) through the use of the standard POSIX API `mmap`. The final programmer is not required to access any custom API: memory allocation is still offered via the standard `malloc` function, which is transparently redirected at run-time to a custom memory allocator able to manage PDES peculiarities (e.g.: memory causal consistency issues).

To have a clear picture of the stock mechanism designed to follow memory accesses occurring upon cross-state events, the IA32e paging scheme offered by x86-64 architectures needs to be taken into consideration. Indeed, in such a system, when dealing with low-level memory management, the virtual to physical address translation is achieved via a technique called paging, being a page the smallest memory manageable unit. This organization is based on storing some data structures, called page tables, that consistently keep track of saved mappings between virtual and physical pages. This kind of architecture, in particular, supports (up to) 5 levels of page tables, each one memorizing, in a single entry, the address of the

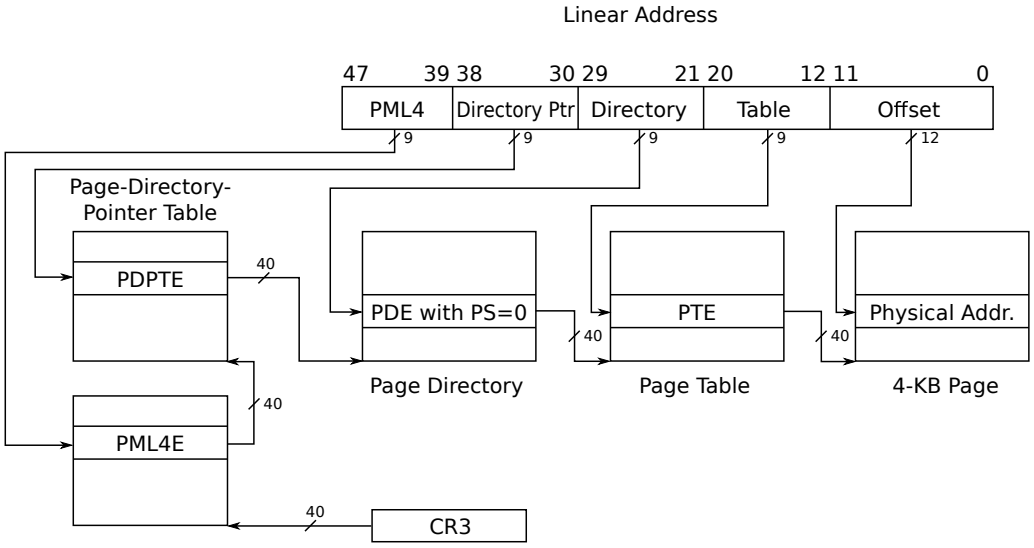


FIGURE 3.2: *The IA32e paging scheme with 4KB pages in x86-64 processors.*

lower level one, until the last one, which is pointing to the physical address of the actual 4 KB page. Figure 3.2 depicts the reference paging scheme, in which for any 64-bit logical address, 48 bits are considered valid for the memory mapping procedure. Indeed, they are used to handle addresses for the 4-level paging scheme, ultimately supporting pages of 4 KB in size. The top level page table is called `PML4` (or also PGD—*Page General Directory*) and keeps 512 entries. All the lower-level page tables are also able to store 512 entries each.

Whenever any virtual page needs to be accessed, it is essential to know where this indirection starts. This is the reason why the *Control Register 3* (CR3) maintains the physical address of the `PML4`, namely the first level page table.

In the ECS paging scheme, a stock of virtual memory pages, which in turn are related to memory buffers allocated for a certain LP, directly maps to a collection of (contiguous) virtual pages. The translation of this set of pages is associated with a single entry of the *Page Directory Pointer* table

(PDP), which is the page table placed at the second level of the indirection. An item of this table is commonly called PDP table entry (PDPTE). That said, given that the lower-level table (*Page Table Entry*, PTE) stores 512 entries, it follows that a single LP stock can store an overall total of 1 GB of virtual memory.

With this organization, LP cross-state accesses detection and management is ultimately supported via the use of ad-hoc operating system's kernel facilities, which are basically bundled into a custom *Loadable Kernel Module* (LKM). Upon loading, the LKM generates a special single-access device file that, at simulation startup, the simulation kernel opens in order to notify the module that the worker threads spawned by it need to follow the logic specified by the ECS protocol. Moreover, the simulation kernel reports the LKM with the address range of virtual addresses destined to a specific LP: an operation carried on using the SET_VM_RANGE ioctl command.

This command allows the driver to change its state, ultimately enhancing the kernel-level memory mapping scheme. Indeed, for each LP-reserved memory stock, a unique identifier of this logical process is stored in a constant-time accessible map which is memorizing the single PDP entry reserved for only that specific LP.

In figure 3.3, we show an example where a given PDP table has its i-th entry (hence the corresponding stock of virtual memory pages) reserved for LP_k, and its $(i + 1)$-st entry reserved for LP_j. Taking advantage of this orchestration, if a logical process LP_i accesses any virtual address residing in the stock dedicated to LP_j ($i \neq j$), the LKM is able to understand that such access, whether is in read or write mode, is crossing the boundaries of the local state of LP_i, finally involving the simulation state of a "remote" simulation object: this is the case of a cross-state event.

However, determining that such a cross-state memory reference occurs

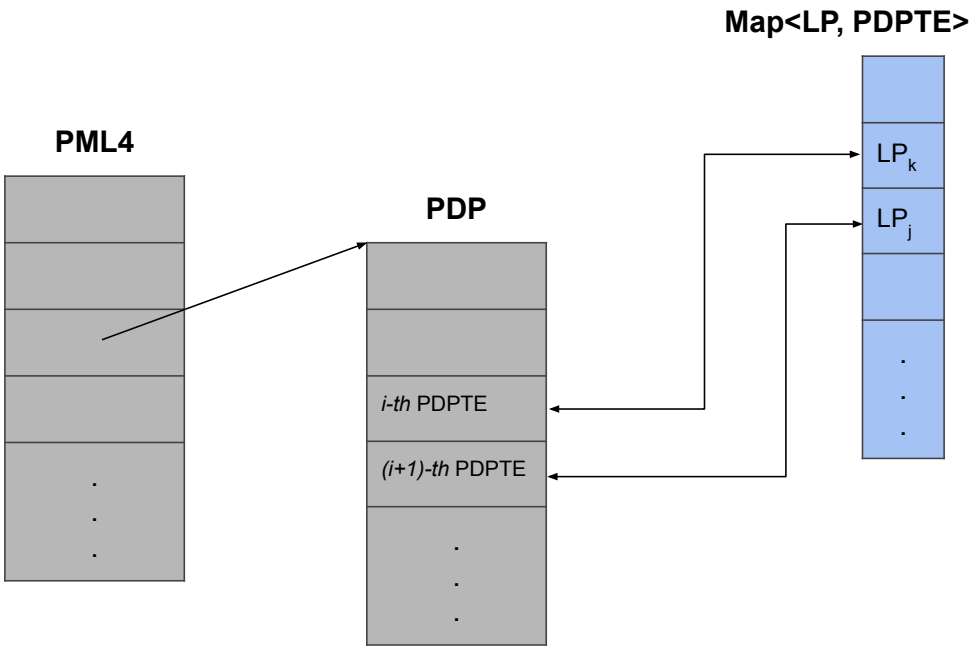


FIGURE 3.3: *Mapping example of logical processes (simulation objects) to stock of virtual memory pages*

during event execution remains a technical problem to deal with. Indeed, two classical approaches could be carried out:

1. **Rely on OS's memory protection mechanism:** typical memory protection APIs (e.g.: `mprotect` POSIX api) would prevent accesses to stocks related to a thread T_i to any other thread, including itself. While this theoretically could be a viable approach, in practice would lead to degraded performance, as would produce unneeded memory faults—by threads running objects other than current LP_i—in situations where the LP_i does not require any access to "remote" stocks while processing the event.

2. **Rely on code instrumentation:** by instrumenting code, we could hijack the actual behavior of memory writes, ultimately understanding the final target of the operation. Again, this could be a possible solution but would produce an overall non-negligible amount of overhead

related to additional code run upon each memory write operation, including those not associated with cross-state accesses.

The ECS memory management scheme can overcome this issue in a different way. Here, any worker thread T_j is associated with a *sibling* chain of page tables, keeping the same number of entries of the original ones. However, every entry of the sibling `PML4` page table points to a NULL-initialized sibling `PDP` entry, ultimately referring to T_i's stock. Therefore, whenever T_i "launches" LP_x to process an event, the entries related to the sibling `PDP` are "unlocked" to seamlessly allow the access to lower-level page tables which in turn will contain the actual mapping to the physical required data. This operation is possible by copying the corresponding entries of the original (i.e.: non-sibling) `PDP` tables onto the destination entries of the sibling `PDP` page tables. On the other hand, this sibling organization will trigger a memory fault whenever LP_x attempts to retrieve data residing in `PDP` entries destined to a different LP (note that a memory fault could occur even if LP_x were accessing its own data, but the actual referred page was not present due to, e.g., swapping. The distinction of such kinds of situations is better explained in the remainder of this chapter). Figure 3.4 depicts such a scenario in which the stock related to LP_i is linked to the i-th entry of a given `PDP` page table.

At this point, a worker thread switches between two different execution modes according to the operation it needs to perform, via the ioctl `SCHEDULE_ON_PGD` command:

- **Simulation object mode**: while in this mode, worker thread T_i is only able to access stocks dedicated to the dispatched logical process LP_i, thus enabling the aforementioned sibling page table scheme. As mentioned above, also the CR3 register needs to be modified in order to make it store the address of the sibling `PML4` and redirect virtual-

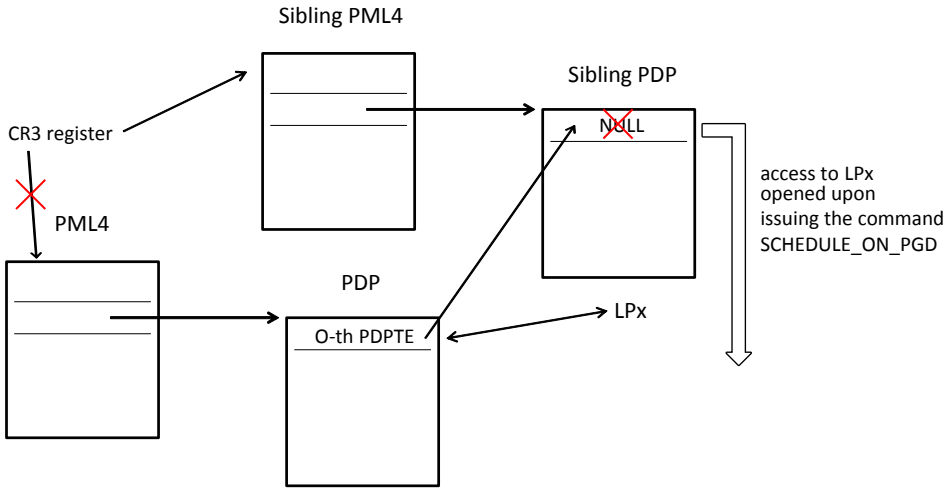


FIGURE 3.4: *ECS schedule example*

to-physical address resolution to the ECS custom one.

- **Platform mode**: in which the standard page table scheme is restored in order to both handle faults happening within the executing LP's stock and other faults related to concurrent processes currently running on the host OS.

By maintaining multiple copies of the original `PML4` (namely, multiple sibling tables), ECS is able to run different LPs concurrently as each thread managing them is granting access to the stocks associated with the currently dispatched simulation object, and is aware of accesses that occur over the boundaries of this LP. However, there are still technical issues to cope with in order to provide such a kind of solution seamlessly.

First of all, the ECS management system needs to run alongside the Linux kernel scheduler, which is not aware of the sibling `PML4` organization. Indeed, according to the Linux memory-context scheme, threads within a certain process would share crucial information related to the memory area they're allowed to access, and this involves the address of the original `PML4` too. This information is stored in the CR3 register upon a context-switch,

which is giving back control to the actual thread. The Linux kernel is however not aware of the aforementioned modes a thread could be in the ECS management, in which the content of that register needs to be customized with the address of a sibling `PML4`. Here, the `kprobe` facility supported by the Linux kernel allowed us to give dynamically control to a custom routine (`schedule_hook`) right before the Linux scheduler completes its standard execution. In particular, when the `schedule_hook` function takes control, a custom CR3 management is put in place, which *i*) understands whether the current thread is running in simulation object mode by checking some per-thread meta-data previously marked via the `SCHEDULE_ON_PGD` command; *ii*) loads the corresponding `PML4` address into CR3, namely the sibling or the original one, thus persisting the thread in its current mode; *iii*) gives back the control to the Linux scheduler in order to perform actions related to this thread's scheduling (i.e.: realignment and return). Note that performing those actions turns out to be a much more flexible solution with respect to providing a recompiled version of the kernel implementing an ad-hoc modified scheduler, which would have been a much more cumbersome realization.

Another issue to cope with is related to the fact that memory faults generated due to cross-state detection cannot be traced via the standard segmentation-fault handling provided by Linux's kernel. The reason behind this lies in the fact that `mmap` facilities have already validated those memory pages, so the Linux kernel would handle this situation by re-allocating the complete chain of page table entries (`PDPTE`s, `PTE`s) in order to properly map the accessed virtual page to the corresponding (present) physical one.

Clearly, in this case, we would have a clone of the whole chain of page tables for specific virtual memory addresses. This situation is not only incorrect but also introduces an unneeded overhead. For this reason, upon LKM

loading, the *Interrupt Descriptor Table* (IDT) is automatically modified to replace the standard Linux page fault handler with our custom callback. This custom handler enhances the standard one by being able to detect whether the triggering page-fault is concerning cross-LP memory accesses, by taking advantage of the sibling page table scheme explained before. If this is not the case, then the control is given back to the original logic of the standard page-fault handler, as to provide, e.g., correct physical page materialization. Conversely, if cross-state access is detected, a synchronization protocol is started by switching back the running thread to platform mode in order to deal with the reconciliation of logical processes with respect to their virtual times, an operation which is needed to ensure that the memory portions to be accessed are consistent among the simulation trajectory of both LPs. As this is crucial for correct PDES execution, it will be thoroughly discussed in next sections.

Moreover, a worker thread can also manually come back to platform mode by issuing the `SCHEDULE_ON_PGD` ioctl command provided by the driver. This on-demand operation was designed to let the thread access page tables of concurrently scheduled LPs. In figure 3.5 a state machine describing this process is depicted.

3.4.3 The synchronization scheme

The features described in the previous section allow the simulation platform to discriminate between memory accesses, taking place at some specific LP_i, having as the target areas belonging to a different LP_j in a completely transparent fashion. However, this is not enough to provide a correct and consistent execution of an optimistic simulation run. Indeed, in speculative execution, it is always the case that LP_i needs to access memory areas updated at logical virtual time LVT_i. At the same time, LP_j could be at

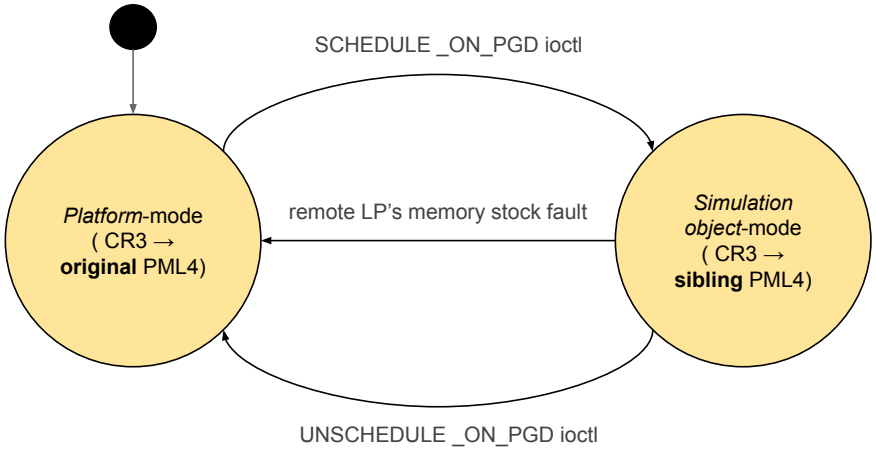


FIGURE 3.5: *State machine describing the switch from platform to simulation object mode.*

a different time LVT_j either in the future or in the past. Thus, Event and Cross State synchronization focuses on two main objectives:

- Allow logical processes to complete event execution in non-decreasing timestamp order.
- Allow logical processes to observe, at any time T of occurrence of cross-state memory access, the state that would have been observed at the same time in a conservative run, i.e., have a consistent view of the memory, no matter how much the local virtual times differ during the whole run.

In order to fulfill these two goals, the simulation platform needs to be able to *i)* temporarily block the execution of an event carried out by an LP *ii)*; notify LPs that a cross-state access is taking place by means of enhanced control messages. The latter point was achieved in this solution with the introduction of *rendezvous* events, which, as the name suggests, are related to a "meeting point in time" between two logical processes. These kinds of events are mapped to platform-level control messages, which do not have any effect at the application level but at the same time bring with

them crucial information about the state of the "meeting". Particularly, they are exploited to temporarily disable an LP to perform updates on its own state while continuing on its simulation trajectory. Rather, it lets the LP access its memory to complete the action and unlocks it when it is safe (w.r.t. simulation consistency) to restore normal event execution. The same, naturally, happens at the other side, i.e., at the LP requesting memory access.

With this in mind, let us now go deeper into details about how the synchronization mechanism works. Let LP_i be a generic logical process managed by thread T_k. We indicate as CSD_i the set of cross-state dependencies containing the identifiers of the logical processes currently involved in a cross-state memory access with LP_i. This set is emptied every time an event is dispatched for LP_i. The protocol starts its execution whenever a cross-state access is detected at LP_i, thanks to the ad-hoc memory manager explained in the previous section. Because of the enhanced memory-fault handler we introduced, LP_i is also aware of the identifier of the LP the cross-state access is targeting. Let's suppose this logical process is LP_j. With this information, the synchronization protocol can start and would entail the following rounds:

- **Protocol setup**: the logical processes need to both stop their execution in order to allow memory sharing. This involves three main steps:
  - LP_i, which generated the cross-state access, stops the processing of the event e_i and temporarily blocks the execution in its simulation trajectory
  - The event e_i is uniquely marked with a *rendezvous* identifier, $RUID(e_i)$

  - A *rendezvous* event re_j is generated targeting LP_j and having the same timestamp as the originating event e_i. Also, it is assigned the same *rendezvous* unique identifier as e_i, thus $T_{e_i} = T_{e_j}$ and $RUID(e_i) = RUID(re_j)$.

- ***Rendezvous* processing**: when a *rendezvous* event is received at LP_j and is scheduled for execution, the following steps are taken:

  - Execution of events at LP_j is preempted, and this logical process enters a blocked phase.
  - A *rendezvous* acknowledgement event re_i^a is generated for LP_j, such that $RUID(re_i^a) = RUID(re_j)$.

- **Synchronization phase**: whenever the acknowledgment is received at LP_i, this logical process can resume its execution. However, it also needs to insert the identifier of the targeted LP_j into its CDS_i. Indeed, at this point, LP_i is assured that LP_j is blocked and that it is safe to access its memory areas. In order to do so, the `SCHEDULE_ON_PGD` ioctl is issued passing as a parameter the $\{CDS_i \cup i\}$ set, required to know which memory stocks need to be open at PGD level.

- **Protocol finalization**: when LP_i completes the actions related to memory accesses of LP_j's state, it needs to unblock the other LP and let simulation proceed. In particular:

  - A *rendezvous* unblock event re_j^u is generated towards LP_j, bringing as *rendezvous* identifier the exact same of the original event that triggered the protocol, i.e. e_i. Hence, $RUID(re_j^u) = RUID(e_i)$.
  - Upon receipt of this unblock event, LP_j is allowed to continue the event's execution from the point where it was interrupted.

Figure 3.6 shows an example of the instantiation of the protocol.

However, since ECS targets speculative PDES environments, there are still a couple of edge cases to handle. In particular, the following causality violations may take place while running the ECS protocol: either after the generation of the *rendezvous* event re_j, event e_i is rolled back at LP_i, or while completing the protocol, LP_j rolls back its execution to an event with a timestamp lower than the time of re_j. In both cases, LP_j could restore some old data in the memory buffers LP_i requested to read/write through the ECS synchronization.

The first scenario is the most trivial of the two: whenever e_i is rolled back, an anti-event for the *rendezvous* re_j is issued to LP_j. This kind of event would effectively annihilate any action of the corresponding event, possibly forcing LP_j to roll back to a consistent situation. If LP_j results to be already in the blocked phase of the protocol, no "unblock" event (re_j^u) is sent to this logical process, but it will simply be automatically unlocked by the execution of the anti-event.

The latter scenario needs to be handled in a more sophisticated way. In fact, if LP_j is required to roll back to an event with a timestamp lower than the time of protocol starting, then a new instance of it should be generated. In fact, in this case, LP_j issues a special *rendezvous*-restart event rre_i, carrying the original identifier (i.e., $RUID(rre_i) = RUID(re_i)$) in order to allow LP_i to match it with the *rendezvous* event which needs to be aborted, or better, annihilated. Correctness is guaranteed by the fact that when LP_i reaches e_i, which is going to generate re_j, its *rendezvous* identifier will be different from the previous one, evading any possible mismatch.

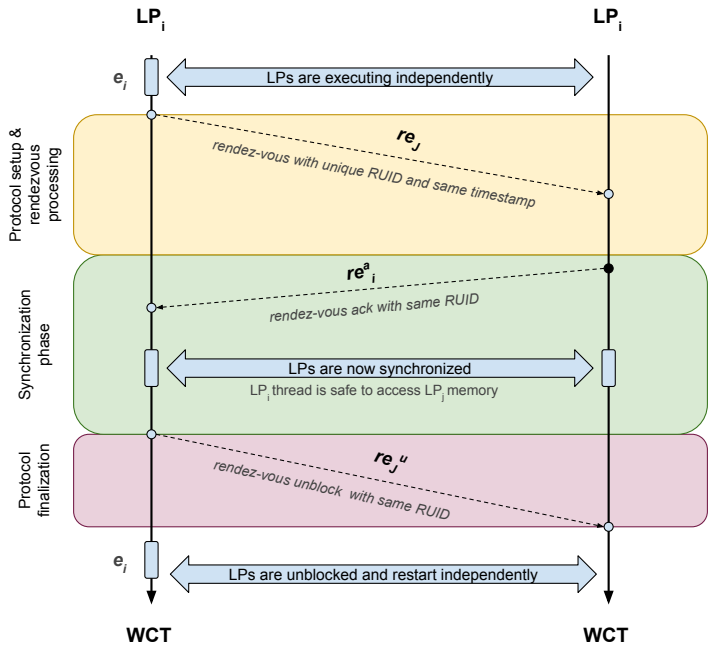


FIGURE 3.6: *An example of ECS synchronization protocol instance*

CHAPTER 4

A distributed shared-memory PDES middleware

People are good at intuition, living our lives. What are computers good at?

Memory.

— Eric Schmidt

As mentioned earlier, over the last decades, the spreading of high performance multi-core platforms was one of the main drives that brought the research in PDES towards the exploitation of new programming paradigms, meant to fruitfully exploit shared memory resources. However, the reduction of the costs related to put in place a distributed memory cluster on cloud platforms introduced, at the same time, the necessity to bridge the gap between shared-memory programming and seamless distributed execution. Taking into account what has been previously described regarding the ECS synchronization protocol, in this Chapter we will present a distributed middleware that transparently allows a PDES application coded for shared memory systems to run on clusters of (Cloud) resources. Being an enhancement of the already discussed protocol, all the concepts related to the ECS implementation internals also apply to this middleware, which expands ECS's original capabilities and unlocks the (distributed) development of many types of PDES applications.

4.1 Going Distributed

The solution presented in the previous Chapter has a significant limitation: the synchronization between logical processes can only entail entities residing on the same machine. This is inherently related to the fact that it is based on kernel-level facilities that are unaware that other simulation kernels regarding the same instance of the simulation may be running in the same cluster in which the machine is placed. Indeed, an excellent way to scale performance over massive simulation runs is to split the work among many CPUs/cores and different machines, each carrying out a slice of the load.

For this reason, the ECS protocol was improved to support synchronization between LPs hosted on the same or on different machines. In other words, the synchronization was ported to distributed clusters. To provide this, a series of enhancements to the protocol described previously has been put in place. Schematically, the new features introduced in this version are:

1. The possibility to detect, again via standard programming language facilities (i.e., pointer dereferencing) the materialization of a cross-state memory access involving logical processes hosted on remote machines

2. The possibility, for a given logical process, to transparently gain a temporary lease on the memory pages requested during the cross-state event and remotely hosted.

3. The possibility to understand, at runtime, the type of the memory access operation that triggered the cross-state event (namely, a read or a write) in order to both prefetch pages and materialize them to write-back (in case of a write operation) the content in the accessed

LP's memory state.

Each of these features require a thorough description. Thus, the following sections will expand these concepts to make the reader better understand how the solution was realized.

4.1.1 The distributed memory view

One of the main problems that arose when designing the ECS synchronization protocol for distributed environments was to find an efficient way to make LPs transparently understand whether they were accessing memory related to another local or remote LP. While the protocol described in the previous section could fulfill the first of the two scenarios, the latter can't be automatically solved by it as the sibling `PML4` wasn't enough to discriminate the locality of simulation objects. Indeed, when a distributed cluster is involved, there needs to be a sort of agreement on which memory areas belong to which LP across all the simulation kernel instances that are carrying on the simulation (according to the architecture depicted in Figure 2.5). Maintaining a good performance level and an abstraction layer able to hide complexity to the final modeler were the crucial points that were taken into account when designing this solution.

The distributed ECS protocol addressed this by providing a deterministic memory map manager shared across every simulation kernel instance. We recall that every logical process has a dedicated virtual memory stock of size 1 GB. Thanks to this manager, the base address of the stock of every LP is deterministically computed by all of the simulation kernel instances taking part in the simulation run. Also, this base address results to be unique among all LPs entailed in the simulation, finally producing a static view of the memory to every machine. Hence, every simulation kernel sees the memory view of all the instances and can act properly according to it.

Indeed, every simulation kernel instance has a way to discriminate whether an LP is touching pages related to a stock of another simulation object that is residing on the same machine or not (although a discrimination algorithm needs to run querying this kind of information to the deterministic memory map manager). To better understand what this solution provided, let us consider the example in figure 4.1.

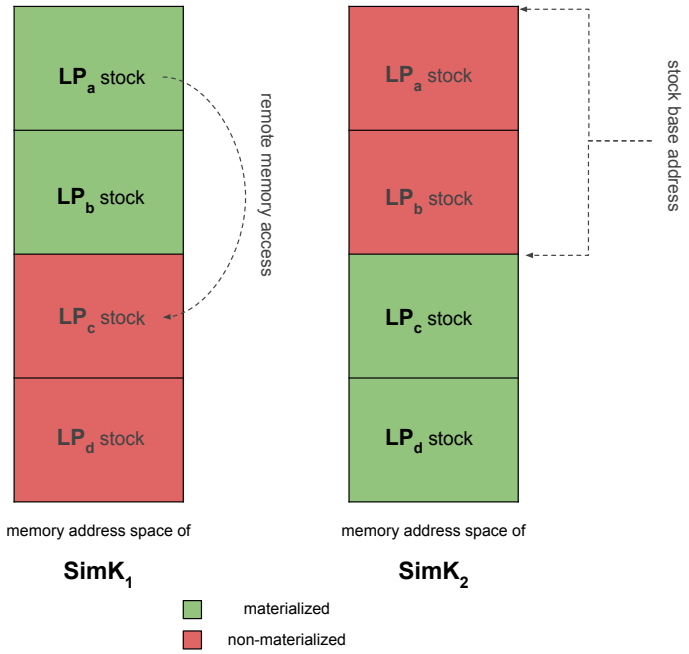


FIGURE 4.1: *Example of distributed deterministic memory map organization*

In this picture, for simplicity, only two simulation kernels are taking care of the simulation run, and the total number of logical processes in the whole simulation amounts to four. Namely, LP_a and LP_b belong to simulation kernel $SimK_1$, and their memory stocks are green in the address space of the simulation kernel they reside in. Note that also stocks of LP_c and LP_d are placed in this memory view. Still, they will not be used by LPs belonging to $SimK_1$: the benefit of this design lies in the fact that, for a given local logical process, accessing a memory area falling inside one

of them (graphically, going from a green stock to a red stock) would mean that this access is related to a remote logical process. This is precisely what ECS needed to achieve: have a the possibility to detect remote cross-state memory accesses.

Overall, the delivery of memory buffers in this memory map organization turns out to be non-anonymous. For example, suppose a logical process is locally hosted on a simulation kernel. In that case, all the memory access requests coming from another LP are guaranteed to ship memory pages falling inside the boundaries of the stock related to that LP, which is known a-priori. As far as remote LPs are concerned, memory stocks assigned to them will be initialized even on "guest" simulation kernels. However, they will never be used to serve memory requests.

4.1.2 The enhanced state machine

Given that the ECS solution can recognize whether memory accesses involve local or remote logical processes, the next issue to cope with is the modality in which these LPs synchronize. Indeed, the already explained protocol is not enough as we not only need to stop the execution of an LP on the same machine the cross-state event occurred, but also on a possibly remote simulation kernel, and potentially both of the situations need to be managed at the same time. It follows that the introduction of new control messages (the so-called *rendezvous* events) and new synchronization states are required to enhance the current picture of an LP.

The correctness of this improved version of the protocol is enforced by two fundamental features. First, the execution of a simulation object can be suspended at any time. This was already achieved in the first version of ECS, thanks to already discussed techniques. For more details on how this can be achieved, see [PQ17]. Secondly, every logical process is required to

fall into one of the states of a well-defined state machine and can transit from a state to another only via the rules bound to it. The updated version of the state machine is depicted in figure 4.2. As the reader can notice, there are three different types of state an LP can reside in at any time:

- **Running**: an LP in this type of state is currently running the logic associated to an event. This is a "normal" situation in which no cross-state interaction has occurred. They are highlighted in green in the below picture.

- **Blocked**: an LP resides in such a kind of state if it was stopped while executing an event due to the occurrence of a cross-state event. Turning into this state is possible via the reception of control messages related to the protocol, namely the *rendezvous* events. These events only trigger an ephemeral state transition. Blocked states are colored in gray in figure 4.2.

- **Ready**: an LP in this state is able to continue the execution of events but is waiting for some other LP to complete actions related to the continuation (or finalization) of the protocol.

As a final remark, the state transitions are triggered by certain actions happening between LPs. Specifically, some of them are the same ones that were already introduced in the first version of ECS (e.g., ECS ack or ECS unblock). However, other actions were outlined to cope with new situations that could happen in a distributed scenario. Briefly, the concepts of major and minor (page) faults were introduced and will be discussed in the next section.

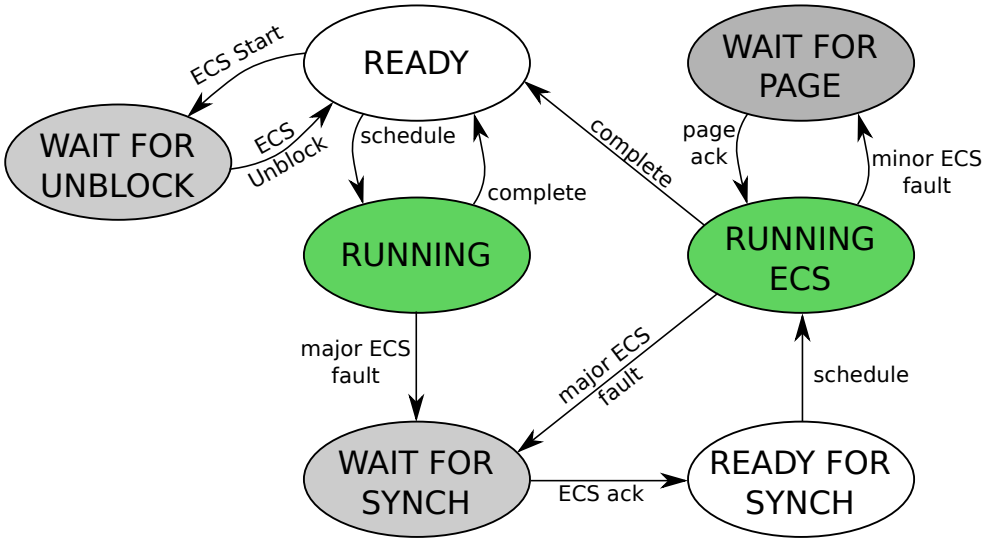


FIGURE 4.2: *The distributed ECS state machine*

4.1.3 The smarter Linux kernel module

To provide the reader with a comprehensive explanation of the functioning of the whole solution, in this section, we outline a detailed description of the LKM devised to trigger the (distributed) ECS synchronization protocol. In order to understand whether a memory access involves different LPs, the LKM intercepts memory faults (namely, *page faults*), which are generated whenever a memory page cannot be found in the current memory view of the thread accessing it. We therefore enhanced the standard mechanism used to manage the materialization of memory pages to make it aware of the sibling page tables scheme, and to allow it to invoke our custom logic to synchronize, at the application level, the involved logical processes.

In algorithm 4.1, we provide the pseudo-code of the page fault handler able to tackle cross-state memory accesses, augmenting the already existent `do_page_fault` function of the Linux kernel.

First of all, there are three cases in which this standard page fault handler needs to be invoked without any custom intervention of the ECS han-

Algorithm 4.1 ECS page fault kernel handler

```
1: procedure FaultHandler(pt_regs *regs)
2:     if current → mm = NULL then                                  ▷ F1
3:         DoPageFault( )
4:         return
5:     end if
6:     if current → pid is not registered then                      ▷ F2
7:         DoPageFault( )
8:         return
9:     end if
10:    target ← ReadCR2( )
11:    if PML4(target) not in LP range then                         ▷ F3
12:        DoPageFault( )
13:        return
14:    else
15:        if PDP(target) = NULL then                               ▷ F4
16:            fault_type ← Major
17:        else
18:            if GetPteStickyBit(target) then                      ▷ F5
19:                fault_type ← Minor
20:                SetPresenceBit(target)
21:            else
22:                if ¬GetPresenceBit(target) then                  ▷ F6
23:                    DoPageFault( )
24:                    if GetPdeStickyBit(target) then
25:                        fault_type ← Minor                       ▷ F7
26:                        SetPageStickyFlag(target)
27:                    else
28:                        return
29:                    end if
30:                else                                             ▷ F8
31:                    fault_type ← AccessChange
32:                    SetPagePrivilege(target, WRITE)
33:                end if
34:            end if
35:        end if
36:    end if
37:    Switch to the original Page Table                            ▷ F9
38:    Copy to userspace fault information
39:    Push on userspace stack regs → ip
40:    regs → ip ← EcsHandler                                       ▷ F10
41: end procedure
```

dler:

- a fault at Linux kernel level arrived (thus, not regarding the specific simulation run). Line **F1** handles this.
- the fault was generated by a process that is not recognized to be simulation-related (line **F2**).
- the address targeted by the memory fault resides outside the boundaries of the memory stocks dedicated to LPs (i.e., it falls inside a `PML4` entry not dedicated to this simulation run). This is managed at line **F3**.

If none of these three cases arise, then the algorithm inspects what kind of fault occurred. More specifically, we distinguished between *ECS Major Fault*s and *ECS Minor Fault*s, regarding the first ECS page fault and the subsequent (artificial) ones, respectively. The first scenario takes place whenever a memory fault occurs on a page ultimately linked to an entry of the sibling `PML4`, which has not been initialized, as they are manually zeroed (i.e., set to `NULL`) when the original `PML4` is copied. In this case, the standard page fault handler is interrupted because the logical processes involved in this cross-state access need to stop and wait for each other in order to have a consistent memory view. This is the reason why at this point (**F10**) the user-level callback `ECS_HANDLER` is invoked, by restoring the original value of CR3 (thus reinstalling the old page tables) and copying into a per-memory buffer the information related to this memory fault (whether it was related a read or a write, the involved address, etc.). The behavior of this handler will be analyzed later as it requires an ad-hoc discussion.

In order to make LPs "return" to this code and execute further inspection about the fault, artificial page faults are forced by exploiting some available bits in the Page Table Entry (`PTE`), the structure maintaining in-

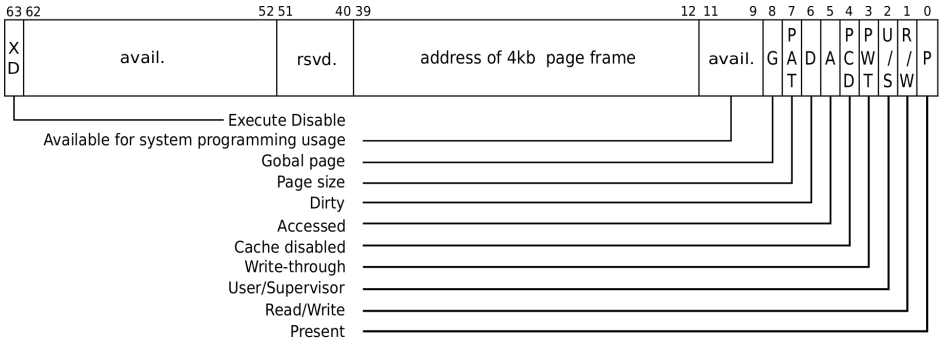


FIGURE 4.3: *The Page Table Entry (PTE) structure*

formation about actual memory pages. As hinted, the `PTE` and `PDE` entries offer the possibility to developers to use some of the unused bit to implement a custom behavior. Figure 4.3 depicts the composition of such a structure. In particular, according to the standard kernel behavior, all `PTE` entries which have been materialized keep their presence bit set to discriminate them from the ones that still need to be allocated. The ECS page fault handler loops through all available `PTE`s (related to a given LP) and explicitly sets this bit to 0, thus provoking an artificial page fault the next time they will be accessed. Naturally, we needed a way to distinguish synthetic and standard memory faults, a condition achieved by setting one of the `PTE` available bits (namely the one at position 9, which we called *sticky* bit). Moreover, to distinguish between memory stocks dedicated to remote or local LPs, we exploited an available bit of the `PDE` entry, namely the one at position 11.

Eventually, the logical process generating ECS synchronization will be rescheduled and will execute the custom page fault handler again since an artificial page fault has taken place due to the custom modification discussed above. By checking the value of the sticky bit, the algorithm then understands whether the accessing page was already materialized, a condition that can happen if a precedent execution of an ECS synchronization is issued. If the check succeeds, it means that the scenario the execution

is falling into is the *ECS Minor Fault* one; thus, the control is given back to the user-space handler as the retrieval of the required pages needs to be put in place. We also reset the old value of the presence bit to restore a consistent situation. On the other hand, if the sticky bit results to be unset, further checks are required. Indeed, if the presence bit is unset as well, then the page needs to be actually brought to main memory: the standard `DO_PAGE_FAULT` would take care of this, as specified at line **F6**. After that, the algorithm needs to understand whether the inspected page belongs to the stock of a local or a remote LPs. As outlined, the bit at position 11 of its relative `PDE` was used to mark this: if this check succeeds (**F7**), this is notified to the userland synchronization protocol by triggering an *ECS Minor Fault* and setting all the sticky bits of the page table entries linked to the just-checked `PDE` entry.

So far, the algorithm did not care about the mode with which the pages causing faults were accessed. Line **F6** deals with this aspect: when a page is accessed in read mode, we explicitly inhibit successive accesses in write mode by setting the read/write bit (at position 1 in figure 4.3) using a custom `IOCTL_SET_PAGE_PRIVILEGE` ioctl command. Modifying this bit would generate an (artificial) page fault whenever an LP tries to access this page in write mode. The purpose of this new page fault is to let the user-space handler know that an *ECS Access Change Fault* is taking place, meaning that a logical process requires to have write access on a (possibly remote) page it already retrieved (access is granted by resetting this bit at line **F8**). This is crucial for optimizations of the protocol, which will be discussed in the remainder of this Chapter.

As a final note, we remark that the proposed algorithm supports both the page sizes that the standard Linux kernel can handle, namely 4KB and 2MB. For simplicity, we only described the first of the two cases, and the

outlined algorithm is related to this kind of configuration.

4.1.4 Synchronizing remote entities

Let us now discuss how the user-space ECS handler, triggered with information provided by the LKM about the occurred cross-state accesses, works. The main logic is presented in the pseudocode fo algorithm 4.1.

When two or more logical processes are involved in a cross-state access request (namely, an *ECS Major Fault* occurred, **H1**), they need to synchronize as their respective Wall Clock Times need to be aligned to avoid causal inconsistencies. The distributed version of the ECS protocol resembles the one described in section 3.4.3, enhancing the states and messages that LPs can manage. Thus, considering two remote logical processes LP_x and LP_y, the synchronization procedure relies on the following steps (having in mind Figure 3.6):

1. LP_x sends a rendezvous start control message directed to LP_y and associated to a system-wide unique rendezvous mark.

2. LP_x execution is suspended, and its state is switched to the *Wait For Synch* one.

3. LP_y will eventually receive the rendezvous start event and generate a rendezvous ack towards the originating logical process. LP_y switches its state to *Wait For Unblock* until a notification from LP_x is received.

4. When the acknowledgment of the synchronization is received by LP_x, its state switches to *Ready For Synch*, an operation which in turn allows the logical process to restore its execution. Given that LP_y is inserted into CSD_x, memory stocks relative to the target LP are opened and are now accessible. Indeed, LP_x and LP_y are now synchronized, and it is safe to access each one's memory. Since LP_y is

remote, though, if LP_x needs to access additional pages, new kinds of messages are triggered to notify this request. As depicted in Figure 4.2, if LP_x needs to access a remote page, it means an *ECS Minor Fault* has occurred at this point: the logic associated with the kernel module will provide the correct information to transmit into the event to be sent to LP_y. In particular, algorithm 4.2 at line **H3** takes advantage of an in-place dynamic disassembler ([Pel13]), which is able to provide information related to the faulting instruction generating the cross-state access. Specifically, it can understand whether the fault was generated due to a read or a write operation and the amount of memory involved. The crucial point is that this is done without any user interaction: all of the required information is transparently retrieved by this tool. The latter LP will then access the requested page and include its data into the payload of the response control message, namely the *page ack*.

5. Whenever LP_x concludes the actions requiring remote data, the synchronization protocol is finalized by sending a *rendezvous unblock* event.

Moreover, it is worth mentioning that this protocol can synchronize multiple logical processes simultaneously, scattered across all simulation kernels related to a single simulation run. This means that LPs are able to cross-access states of both LPs running on the same machine or even placed on a different one, entailing possible synchronization abortions due to rollbacks or causal inconsistencies occurring at any of the involved LPs.

Furthermore, one additional optimization of the protocol needs to be discussed. In fact, thanks to the aforementioned disassembler, the algorithm has the possibility to infer the base address of the first (remote) page and the amount of pages to be transferred among LPs. Thus, the accessing LP

is someway acquiring a lease on the cross-state accessed pages for a certain time window. This means that it temporarily owns a master copy of the data stored in the leased pages, which represent a portion of the state. Being the memory stocks organized in a non overlapping fashion (as depicted in Figure 4.1), this data can be safely installed in the receiver LP's memory address space.

Algorithm 4.2 Userspace ECS handler

```
 1: procedure EcsHandler(type, info)
 2:     if type = Major then                                   ▷ H1
 3:         ECS_mark ← generate_mark( )
 4:         Send(RENDEZVOUS, info.targetLP, currentLVT)
 5:         LP_state ← WAIT_FOR_SYNCH
 6:         CSD ← CSD ∪ {info.targetLP}
 7:         Deschedule( )
 8:     else if type = Minor then                              ▷ H2
 9:         disasm ← Disassemble(info.rip)                     ▷ H3
10:         write_mode ← disasm.write
11:         page_addr ← BaseAddr(info.target)
12:         pages ← PgCount(info.target, disasm.span)
13:         if write_mode then
14:             AddToWriteList(page_addr, pages)
15:         else
16:             AddToReadList(page_addr, pages)
17:         end if
18:         Send(PAGE_LEASE, info.targetLP, currentLVT)
19:         LP_state ← WAIT_FOR_PAGE
20:         Deschedule( )
21:     else if type = AccessChange then                       ▷ H4
22:         page_addr ← BaseAddr(info.target)
23:         AddToWriteList(page_addr, 1)
24:     end if
25: end procedure
```

However, while executing an instance of the ECS protocol, the number of pages requested by an LP and their mode of access may vary multiple times before finalization. Clearly, it would be very inefficient to start a new instance of the protocol every time such a situation occurs. Instead, the algorithm keeps track of two so-called *touch* lists: one is devoted to store a reference to the pages accessed in read mode and the other dedicated to the pages accessed in write mode (*read-list* and *write-list*, respectively). An entry of those lists also contains the address of the currently owning LP, as the

synchronization may involve any number of logical processes, so the ownership must be respected throughout the execution of the synchronization protocol. While on the one hand, accessing a page in read mode would result for an LP to automatically gain a "read lease" on that page by installing it in the local address space and having the corresponding entry in the *read-list*, accessing a page in write mode would generate an additional artificial page fault and the ECS handler will perform the switch of the entry from the read to the *write-list* (**H4**). In order to bring back the updated pages to

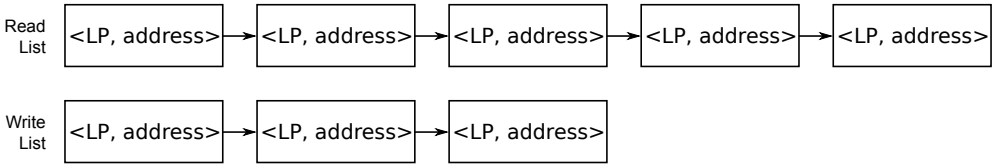


FIGURE 4.4: *The read/write touch lists*

the original owner LP, thus ultimately reflecting changes into the memory address space of the LP that received the cross-state access, the rendezvous unblock event was augmented. In this case, it needs to piggyback all the pages for which a lease in write mode was acquired during synchronization. On the recipient side, this payload is settled immediately before restarting the normal execution of the LP. Hence, upon restart, the simulation object will see a reconciliated state due to the state updates brought back by the protocol.

4.1.5 Pages prefetch

The scheme we introduced in the previous section supports cross-state events making remote logical processes communicate in a causal-consistent way. When accessing remote pages, it even provides a way to buffer the accessed pages by allowing the gaining of leases on them. However, even if this reduces the number of multiple ECS instances by tracking subsequent

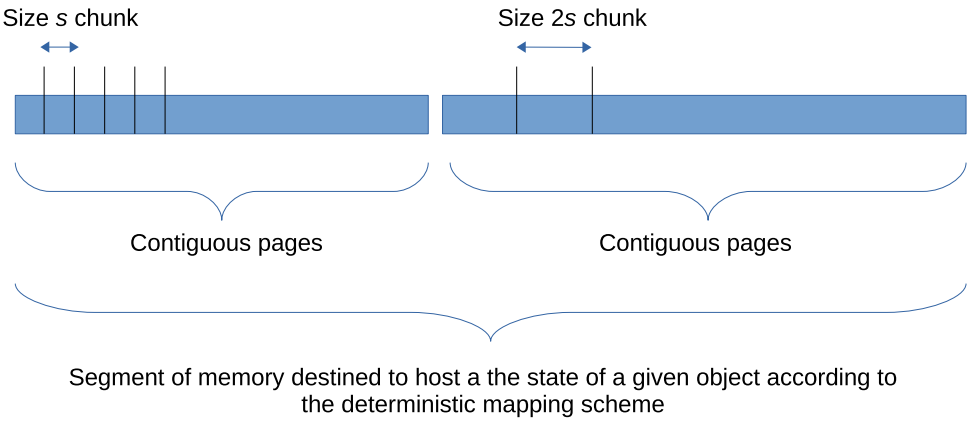


FIGURE 4.5: *Example of a memory allocator with chunks of same size placed in specific buckets*

memory accesses (during a synchronization process), the overall number of memory faults could be optimized if some *speculative* fetch of pages is put in place. This section presents an approach to determine at run-time whether the prefetching of remote memory areas could be convenient, according to multiple prefetching policies, which pick those pages according to their rules.

The idea behind this optimization is to reduce the number of *ECS Minor Fault*s that happen within a cross-state window by loading memory pages beforehand, upon an *ECS Major Fault* (i.e., upon sending the very first rendezvous control message to start the synchronization). Nevertheless, the objective is to optimize the tradeoff between the cost of multiple *ECS Minor Fault*s and the overhead associated with a wrong memory pre-fetch, i.e., pre-load of pages that will eventually never be used.

We devised two different prefetching policies to better adapt to different types of memory accesses. The first alternative, *Clustered prefetching* (CP), better suits situations in which a cross-state event refers to memory areas lying in an LP's state in a contiguous way. For instance, this is the case of simulation models using data structures that are inherently based on contiguous memory, such as huge structs or arrays or even a combination of

them. When this policy is enabled, the ECS middleware requests a number N of contiguous pages to the target LP: the first of the series initially generated the cross-state event, while the other $N-1$ will be the pre-fetched ones. If a number $N' < N$ of memory pages are dedicated to the target LP at the recipients' side, then the maximum available pre-fetch number of pages is returned, namely, N'.

The other prefetching policy, called *Scattered Prefetching* (SP), selects the N pages to pre-fetch in a random fashion, although this randomness is somehow biased on the density of memory chunks currently in use by the LP, placed on a range of contiguous pages. As opposed to the previous policy, this approach is suited for cases in which a memory allocator, exposed at model level to facilitate developers over memory allocation, uses sequences of chunks of contiguous memory pages residing in buckets assigned to chunks of a specific size (resembling, e.g., buddy systems). Hence, depending on the size of the memory requests coming from a specific logical process, this policy picks chunks from different zones of contiguous pages scattered over buckets of adjacent memory. To make this clearer, figure 4.5 depicts such a kind of allocator in which the right portion organizes the retrieval of chunks of size *2s* while on the left, chunks of half the size are managed. Dymelor, [PVQ09], is the memory allocator based on this concept used in the simulator where we carried out experiments.

A typical situation in which a bias on the chunk bucket is evident is how linear data structures are used. As an example, considering a linked list, it is very likely that memory allocation happens within the boundaries of a certain bucket as the list contains elements of the same type that tend to be similar in size. For this reason, concerning SP, we assigned to each bucket b_i a weight w_i, in such a way to pick a random number of pages residing into bucket b_i proportional to its weight w_i. Therefore, if the number of pages

to pre-fetch is N, then the number of pages N_i randomly selected from the bucket of contiguous pages b_i amounts to $N \times w_i$. This weight is calculated as the normalized occupancy percentage of every single bucket, a quantity that depends on:

- the number of chunks that are on average allocated from that zone
- the bucket (and thus chunk) size

Again, suppose less than $N \times w_i$ pages are currently related to chunks residing into a specific bucket b_i. In that case, we pre-fetch only the pages that store chunks related to buffers used by the LPs, i.e., belonging to its state. The autonomic policy driving the transition between these two pre-fetch policies follows the states dictated by the state machine depicted in figure 4.6. Here, the starting state is the *No-Prefetch* (NP) one, in which the platform is not performing any pre-fetch. Then, the transition between policies happens automatically based on two different events: *i)* a certain period of time has passed *ii)* a specific threshold α computed via inequality 4.1 is reached.

$$minorFaults_X \leq \alpha \times minorFaults_{NP} \quad X \in [CP, SP] \tag{4.1}$$

The above inequality compares the estimated average number of *ECS Minor Fault*s (i.e., additional faults occurring after a cross-state event, which can be thought of as "inner" cross-state events) that are taking place while being in one of the available modes ($minorFaults_X$) and the number of faults that was observed while running in the no-prefetch mode, ($minorFaults_{NP}$). In other words, this means that the transition is based on the computed effectiveness of the policy the platform is currently adopting: if it results that, for a specific time window, a model concentrates on contiguous memory areas, then the platform will be prone to switch to the

CP policy, while if for some reason, in a simulation trajectory, the model switches to allocation scattered over chunks of different sizes, then the SP will be preferred, and the run-time environment will automatically switch to this policy. Of course, it could be the case that neither of the two pre-fetch approaches suits the current scenario the simulation is falling in. In this case, the NP policy will be selected as the overhead of this optimization would be too high with respect to the speedup earned by the page-in-advance retrieval. This kind of cyclic retry can be helpful in scenarios in which the execution continuously changes the page access scheme. Conversely, it could be effective even in situations in which the access pattern remains stable throughout the run.

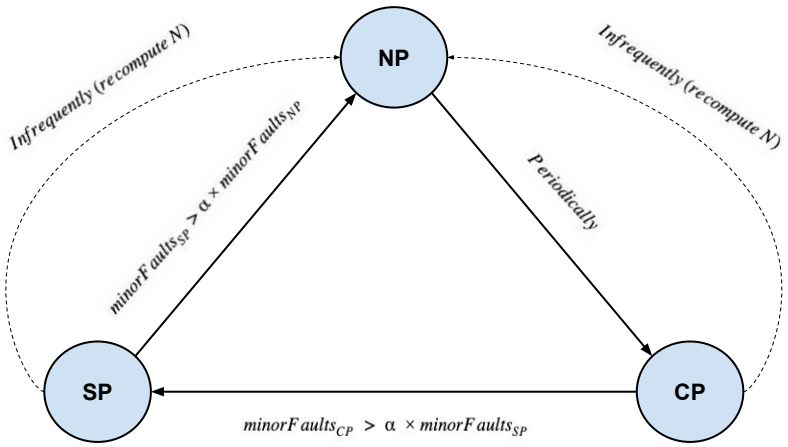


FIGURE 4.6: *The state machine driving pre-fetch mode transitions*

In inequality 4.1 there are still two parameters that need to be discussed. First of all, the number N of selected pages pre-fetched while being either in CP or SP is extracted at run-time as the average number of remote page

accesses occurring during a NP phase. The rationale behind this is to obtain in advance roughly the same number of pages that cross-state events would access to be as close as possible to the actual data that (probably) would have been needed to retrieve from a remote machine. This also explains why, as it could be noticed in figure 4.6, there are additional transitions from CP to NP and from SP to NP. Indeed, the evaluation of N represents a core aspect for the functioning of the autonomic policy, and a mechanism to estimate this value while executing the protocol is required. Although those are marked as a *infrequent* transitions, they were explicitly designed to re-compute the goodness of the pre-fetch policies along the model execution lifetime.

Moreover, the parameter α is dynamically tunable while running this autonomic policy. Based on experimental results, we suggest values falling into the interval $[0.1, 0.2]$. Indeed, small values of α would lead the policy to avoid the switch to any pre-fetch policy, ultimately fixing the state to the NP one. On the other hand, higher values of α would bring the approach to consider a pre-fetch policy consistently better than the NP one, even if the amount of non-required fetched pages is relevant.

4.2 Performance evaluation

This section provides experimental results of the solution presented in this Chapter and discusses a performance evaluation of the features proposed in the ECS middleware. Moreover, this section concentrates on results dedicated to the distributed protocol (the standard and the optimized ones, presented in sections 4.1 and 4.1.5), as it is one of the main focuses of this thesis. We remind the reader that all of the proposed solutions were carried on within the ROOT-Sim simulation platform, presented in section

2.4. The reference testbed platform is composed of two different computing architectures. The first is based on an HP ProLiant server equipped with 8 AMD Opteron 6367 CPUs, running at a speed of 2.6GHz and mounting 100 GB of RAM. Every CPU includes four cores each, amounting to a total of 32 physical cores. To provide a distributed environment, simulations were carried out over a cluster of 16 Virtual Machines (VMs) running on top of the architecture mentioned above. VMWare Workstation (version 10.0.4 build-224991) is the selected hypervisor orchestrating the VM instances, whose virtualized operative system is Debian 6.0.7 with Linux kernel 3.10. Moreover, some of the simulations were run on top of single-vCPU and dual-vCPU configurations of the VMs. The second computing platform relies on a cluster of VMs rented from Amazon Web Services. Specifically, 16 t2.large VMs equipped with Intel Xeon 2.4GHz vCPUs processors and 8GB of RAM each were selected. This configuration was only used for the NoSQL experimental assessment, whereas the software configuration chosen for the whole experimental evaluation is the ROOT-Sim (version 2) simulation platform. Before going into the performance evaluation details, n exhaustive analysis of the models used as benchmark running on top of the simulation platform is required.

4.2.1 Multi-Robot Explore Model

The first model to introduce is a multi-robot exploration and mapping model, coded according to the results given in [FKK+06]. Here, the main entities are robots whose objective is to explore the unknown area they lie in, using sensors they are equipped with. In our implementation, the area to explore is a square grid divided into hexagonal cells. Each robot independently moves into this region, building its map of the currently known world around itself. Indeed, each robot defines its own *exploration fron-*

tier establishing the closest unexplored site it can reach from its current position.

This concept turns out to be crucial when the next movement has to be computed, since every entity relies on its *exploration frontier* in order to calculate the shortest path to reach it and thus continue the exploration. In the beginning, no information about other entities is shared across robots: they don't know the position of other buddies (besides, they don't assume that other robots are even present). However, the cooperation of robots is supported, and if they manage to meet in a so-called *proximity area*, they start interacting by:

1. retrieving each others' sensors information in order to estimate their mutual position, as they may not be placed exactly inside adjacent cells

2. agree on a *meeting point* where to meet again

3. share the partial map they currently own and merge it with their respective view of the area

4. determine their next exploration targets, basing on the evaluation of *cost* and *utility* functions (for more details please refer to [FKK+06])

5. start cooperating constituting a cluster and try to predict the position of other robots not included in their group. If the guessed position turns out to be verified, meaning the demanded robot reaches a *proximity area* related to a previously unknown entity, the steps mentioned above are repeated, and cooperation goes on.

This type of model is well suited to replicate, for instance, natural disaster scenarios in which robots are required to perform risky and unsafe rescue operations. Each robot is represented via a single logical process,

and also every single hexagonal region is modeled via a dedicated LP. In order to properly assess the goodness of the ECS protocol, we provided two different implementations of the *robot explore* model:

- **no-ECS-based**: the communication between robots and the registration of a robot into a hexagonal cell are implemented with standard message passing techniques. For instance, whenever two robots (namely, two different LPs) need to share their current portion of the explored world, the data to be passed is enveloped into the payload of the event modeling this action. This operation needs to be manually coded by the model developer.

- **ECS-based**: all the interactions between LPs, which means the communications between robots and between robots and cells, can be done via dereferencing pointers related to the state of the entity of interest. This means, for instance, that whenever a robot wants to access the memory area of another component of its cluster, it only needs to explore the content of the pointer of the state of it.

In Figure 4.7, we report the execution time in seconds of both the solution while varying the number of involved VMs and having a fixed amount of logical processes. From the data, we can understand that the ECS solution can provide a slight performance increase below a certain amount of computing nodes. This is mainly related to the fact that LPs are more likely to reside on the same node when a reduced number of machines is involved. Therefore, the likelihood that memory accesses can be realized without transferring them over the network is high. By increasing the number of distributed computing nodes, the overhead introduced by the ECS solution raises up to 30%. Nevertheless, increasing the number of nodes while keeping a fixed amount of LPs results in a higher degree of parallelism. Indeed, not only the probability that logical processes generate

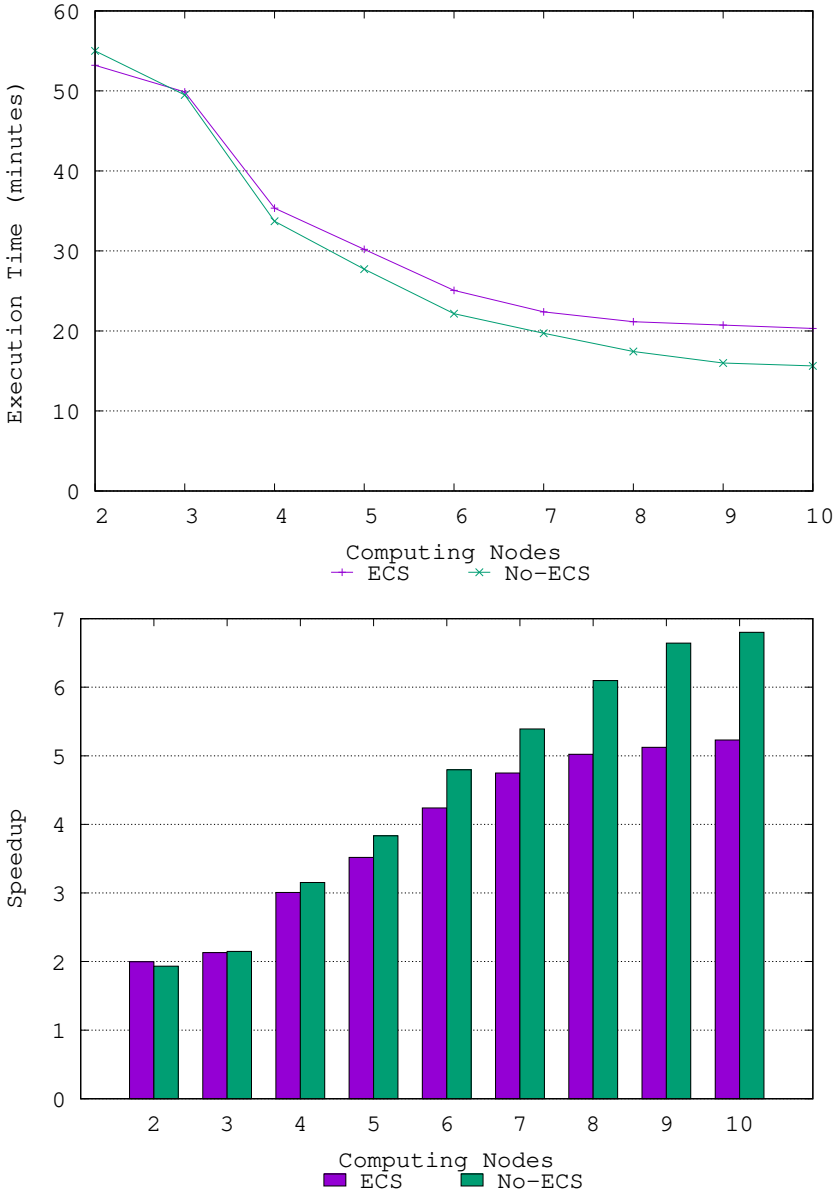


FIGURE 4.7: *The results for the robots explore model with distributed ECS middleware.*

remote cross-state accesses is higher but also the chance that LPs trigger rollbacks to the synchronizing entities raises. In fact, on the one hand, logical processes are forced to synchronize their local clocks, transfer accessed memory pages and write back the accessed data, an operation that clearly comes at some costs both in terms of time and resources. On the other hand, different logical processes may require to interact with entities that are currently synchronizing. Still, considering the optimistic nature of the simulation, they can deliver *straggler* messages forcing the synchronization protocol abortion. As it may be spotted, this represents a worst-case scenario for the ECS protocol. However, despite the adverse situation, ECS can provide a reduced amount of overhead while at the same time providing the modeler an easy to use programming paradigm, speeding up the process of development, avoiding bugs in implementations, and letting the focus of the study concentrate on the model itself, not on the actual implementation.

In figure 4.7, we also show the speedup of the execution with respect to the utterly sequential execution of the same model, relying on an efficient Calendar Queue data structure (for more details, please refer to [Bro88]). Although the fixed amount of LPs is limiting the *ECS-based* execution, from the results, we can notice that the overall speedup is non-minimal, with the final outcome that the communication-intensive widely-distributed scenario can overtake the *no-ECS* execution case. However, when increasing the VM count, the *no-ECS* solution scales slightly better. Again, this is a phenomenon related to the higher degree of parallelism discussed before: the tradeoff stands in the fact that the increased scalability comes at the cost of coding the event-handler in pure data separation across the LPs, which imposes the use of specific APIs for making the LPs interact with each other explicitly.

As far as the enhanced pre-fetch version of the ECS middleware is con-

cerned, further considerations on the *ECS-based* implementation need to be done. In this configuration, each LP dedicated to the modeling of a region keeps in its private state a *presence bitmap* and an array of LP state pointers. The purpose of these data structures is to memorize whether a robot, i.e., a logical process, is currently residing in a region or not. Indeed, every bit of the *presence bitmap* is assigned to a specific LP, and by relying on fast bitmap scans, every robot has the possibility to be aware of the robots the region is currently hosting. After that, by accessing the corresponding entry within the array of state pointers, each robot can directly refer to the state of the LPs modeling the hosted robot. These are the reference data structures for cross-state accesses. Considering that the amount of LPs used to carry on experiments amount to 484, it follows that those per-LP data structures results to be small in size, ultimately fitting one OS page. Moreover, even the state of a robot involves data that can be contained in one OS page: this definitely represents the worst-case scenario for this version of the ECS middleware as the prefetching of pages is essentially irrelevant for the run-time dynamics of the model. Formally, according to inequality 4.1, the following pathological situation will always stand:

$$minorFaults_{CP} = minorFaults_{SP} = minorFaults_{NP} \tag{4.2}$$

In other words, this means that this model is not inclined to actuate prefetch optimizations, being the NP the leading state of the machine driving the transition of the optimization modes.

4.2.2 NoSQL Data-Store Model

The second model to discuss for the experimental assessment mimics a variation of the NoSQL data-store simulation presented in [DSQC+15]. In par-

ticular, this model targets two types of entities:

- cache servers, which manage copies of complete or partial data sets
- clients, which request transactional data accesses, possibly updating data through cache servers

The cache servers can be configured to implement different distributed protocols that guarantee consistency and isolation during transactional data processing: in our design, the chosen protocol is the *2 Phase Commit* (2PC).

Mainly, the kind of events that can be generated at any cache server are the following:

Begin: to simulate the initiation of a transaction requested by a client.

Write: to simulate a write operation concerning the running transaction

Read: to simulate a read operation concerning the running transaction

Prepare request: to simulate the beginning of the 2 Phase Commit distributed coordination protocol

Commit: to simulate the end of the current instance of the 2PC protocol, meaning a transaction has been successfully executed.

Each cache server maintains a set of N data objects, represented by a $\langle key, value \rangle$ tuple. This information is stored in a bucket-based data structure indexed by key values, and buckets are modeled as a list of arrays. The modeling of a cache server is assigned to a single LP. Hence, a logical process is required to simulate resource usage of a cache server, along with the state of the keys the server manages.

Whenever the 2PC protocol is started, every cache server needs to communicate to the transaction coordinator the data that needs to be updated, namely the content of the *write sets* to be reflected as the transaction result.

Those sets may entail hundreds of data objects populated by the transaction coordinator while simulating the execution of a batch of transactions. Therefore, the instantiation of those keys is arranged by the LP modeling the current transaction coordinator, and this is done within its local state.

To have a thorough assessment of how the ECS protocol would behave in this situation, we again rely on two different implementations of the model based on how the *prepare request* event is constructed:

- **no-ECS**: in this implementation the modeler is explicitly required to program the marshaling/unmarshaling of *write-sets*. Thus, the *prepare request* payload consists of the actual data to be written during transaction execution.

- **ECS**: in this implementation, the payload of the *prepare request* only contains the pointers to the memory areas referring to the *write-set* of the data objects to be written during transaction execution.

We present, for this model, data related to the pre-fetch optimization of the ECS middleware. In particular, this model is extremely different in nature with respect to the one discussed in the previous section. Indeed, in this case, the state of a logical process is large, as the *write-set* of the batch of the currently executed transactions are saved in the state of an LP. The keys contained into each *write-set* are then kept into memory buffers which ultimately span several OS-pages: as opposed to the premises of the other benchmark model, this scenario results to be interesting to evaluate whether the prefetching autonomic enhancement can represent a viable solution to actually gain in terms of performance.

Furthermore, the performance scale can be assessed even further with this model as the CPU and memory resources requirements become higher for this situation. More in detail, the state of a logical process is formed by a

unique memory-contiguous table, which is storing, along with some possibly transaction-unrelated metadata, the array used to keep track of the *write sets* of transactions. The size of this array is predefined and sums up to the maximum number of write operations that a group of transactions can execute. Being the number of touched keys by a transaction non-deterministic, it follows that only a part of this array can be fruitfully exploited at any time: the array will be sequentially filled with the correct amount of entries needed to fulfill the *write set*s of the transaction being processed. This means that the number of involved OS pages related to a cross-state event accessing this array is non-deterministic too. The simulated environment consists of 64 cache servers in which the number of active concurrent clients issuing transactions is 64, and the amount of keys touched in write mode by a bundle of transactions is distributed between 1000 and 2000.

The results in figure 4.8 refer to our private cloud computing platform, with different configurations of the α parameter (namely, 0.2 and 0.4) and with time intervals between the automatic switch back to the NP state set to 10^3 and 10^4 simulation time units, respectively. Note that in the results that we collected, we included our non-optimized solution presented in section 4.1. From the plots, we can understand how the pre-fetch enhanced ECS succeeds in achieving scaled-up resources of the underlying platform. Indeed, while increasing the number of VMs, execution time scales down close to linearly. Moreover, when compared to the non-optimized version of ECS, the improved version of ECS even shows better execution times, a noticeable indication of this solution's ability to amortize costs of keeping track of memory accesses spanning multiple (possibly unrelated) OS memory pages involved in cross-state events.

Furthermore, the pre-fetch ECS is proven to provide better performance in the case of more probable cross-state events, as this happens when the

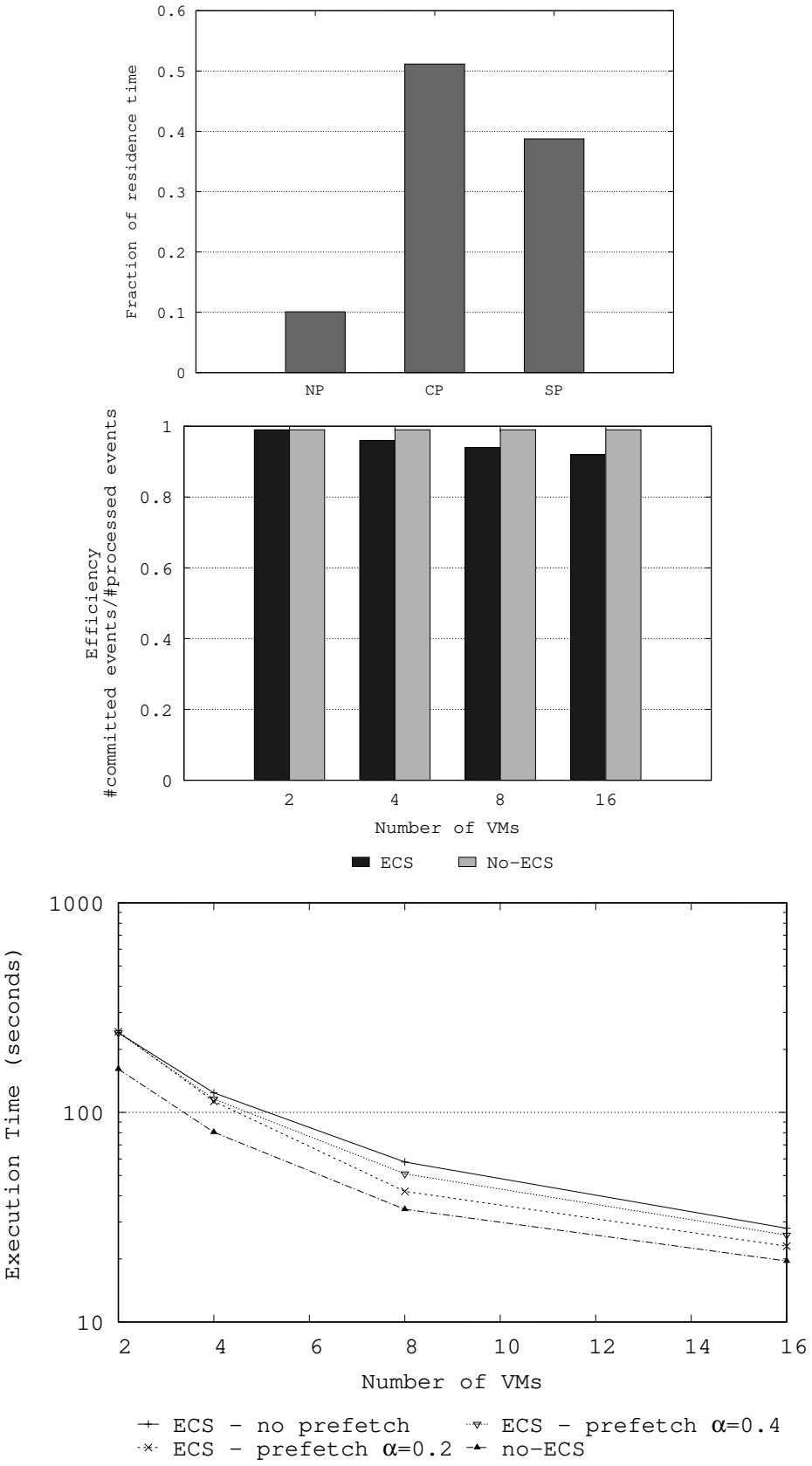


FIGURE 4.8: *Percentage of residence times in prefetch states efficiency and execution time with $\alpha = 0.2$ for the private Cloud deploy - NoSQL model-Single-vCPU VMs*

number of distributed nodes increases, a scenario in which, from the third chart in figure 4.8, such a performance increase is even more evident. Also, it is to be noticed that the value of the α parameter could be relevant to the overall performance of the optimized ECS. Indeed, we empirically observed that lower values of α tend to reduce the ratio between overhead generated by wrong OS-pages gathering and overall speedup caused by fetching pages in advance.

Overall, we can understand that the optimized ECS version provides similar scalability trends with respect to the *no-ECS* one, but an increased performance, even with an increased number of VMs. As hinted previously, this is related to the fact that the middleware inherently inserts some overhead to reduce the cost of marshaling/unmarshaling of data packets (i.e., events) between the involved simulation objects. Additionally, considering the approach used to implement this specific model, the computational cost of marshaling/unmarshaling of data to be shared during a cross-state event is someway lightweight, as, for instance, it doesn't involve complex data structure traversals, which puts our optimized solution in an adverse scenario.

In the first plot of figure 4.8, we also report the data regarding the percentage of residence time in each of the three possible prefetching modes (namely, *Clustered Prefetching*, *Scattered Prefetching*, and *No-Prefetching*). This histogram shows that the autonomic policy well captures the single array layout of the LPs' state, as the most chosen policy for the automatic transition is CP. However, interestingly, the SP one reveals to be a prominent candidate fallback as its residence time is significantly higher than the one in the NP state. To explain this, we need to consider the probability of prefetching pages related to a portion of the LPs' state array not involved in the cross-state event. Recall that SP selects the memory pages to pre-

fetch randomly among those containing memory chunks belonging to an LP state. Thus, that probability depends on the distance between the number of cache server keys included in the 2PC instance of interest and the size of the array itself. In other words, relevant errors in this policy may occur when the write-sets of the currently simulated transaction have a cardinality closer to the minimum size, which will, in turn, leave the array empty for most of its space, thus leading to pre-fetch useless OS-pages.

As a final note on these single vCPU results, the second chart in Figure 4.8 reports the efficiency (i.e., the ratio between correctly processed events and re-processed ones) for the case of $\alpha = 0.2$. The data shows how the optimized ECS provides a slightly worse efficiency, indicating that the rollbacks' incidence is a more relevant factor. However, this is an expected behavior if we consider that the ECS middleware is based on synchronization points that ultimately provoke event execution suspension, which favors the execution of events with higher timestamps. This clearly leads to more aggressive speculation and even more rollbacks over the whole virtual time. When increasing the number of vCPUs, the results follow a similar trend with respect to the already observed ones: the interesting note is that the increased computing capability is well exploited from our solution, which for example, further scales and reduces execution time while increasing the number of VMs (in this scenario we're exploiting a total of 32 vCPUs). Still, lower values of α provide a better performance, and the CP policy is preferred. Data related to this configuration is shown in figure 4.9.

More interestingly, in figures 4.10 and 4.11, we show the execution times and efficiencies for the AWS deployment, both in cases of single and dual vCPU, respectively. We recall that the AWS platform doesn't give any guarantee on the mutual location of the requested VMs, as they could be placed on different physical machines. This is an entirely different scenario

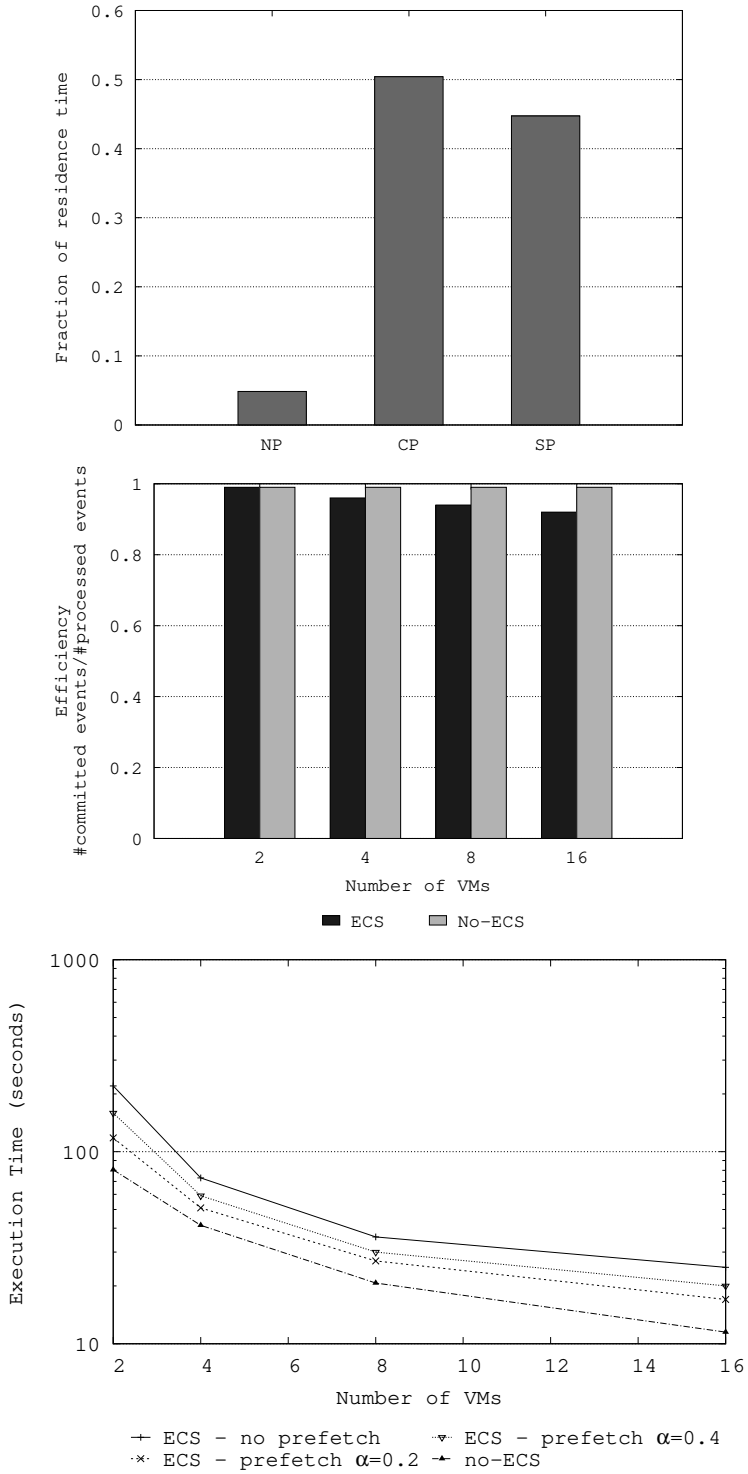


FIGURE 4.9: *Percentage of residence times in prefetch states, efficiency and execution times with $\alpha = 0.2$ for the private Cloud deploy - NoSQL model-Dual-vCPU VMs*

concerning the private cloud infrastructure, discussed for all the previous results. Thus, latencies related to communication between computing nodes become a crucial factor, possibly generating shifts in the dynamics of the simulation runs. For instance, higher latencies could lead to an increased rollback probability, hampering the overall performance. Interestingly, the optimized ECS middleware presents a better performance in this setting compared to the *no-ECS*, which generally behaves better even with higher VM counts. The reason behind this is twofold.

First, we experience an overall comparable efficiency between ECS and *no-ECS* solutions because the increased communication latency across nodes is somehow predominant compared to the increasing aggressiveness of the speculation level mentioned above.

Secondly, the total cost of operations in the *no-ECS* case, such as rollback management, turns out to be more costly when compared to cross-state access processing. Indeed, the enhanced ECS solution reveals to be efficient in optimizing communication between nodes by reducing the overall number of exchanges between logical processes.

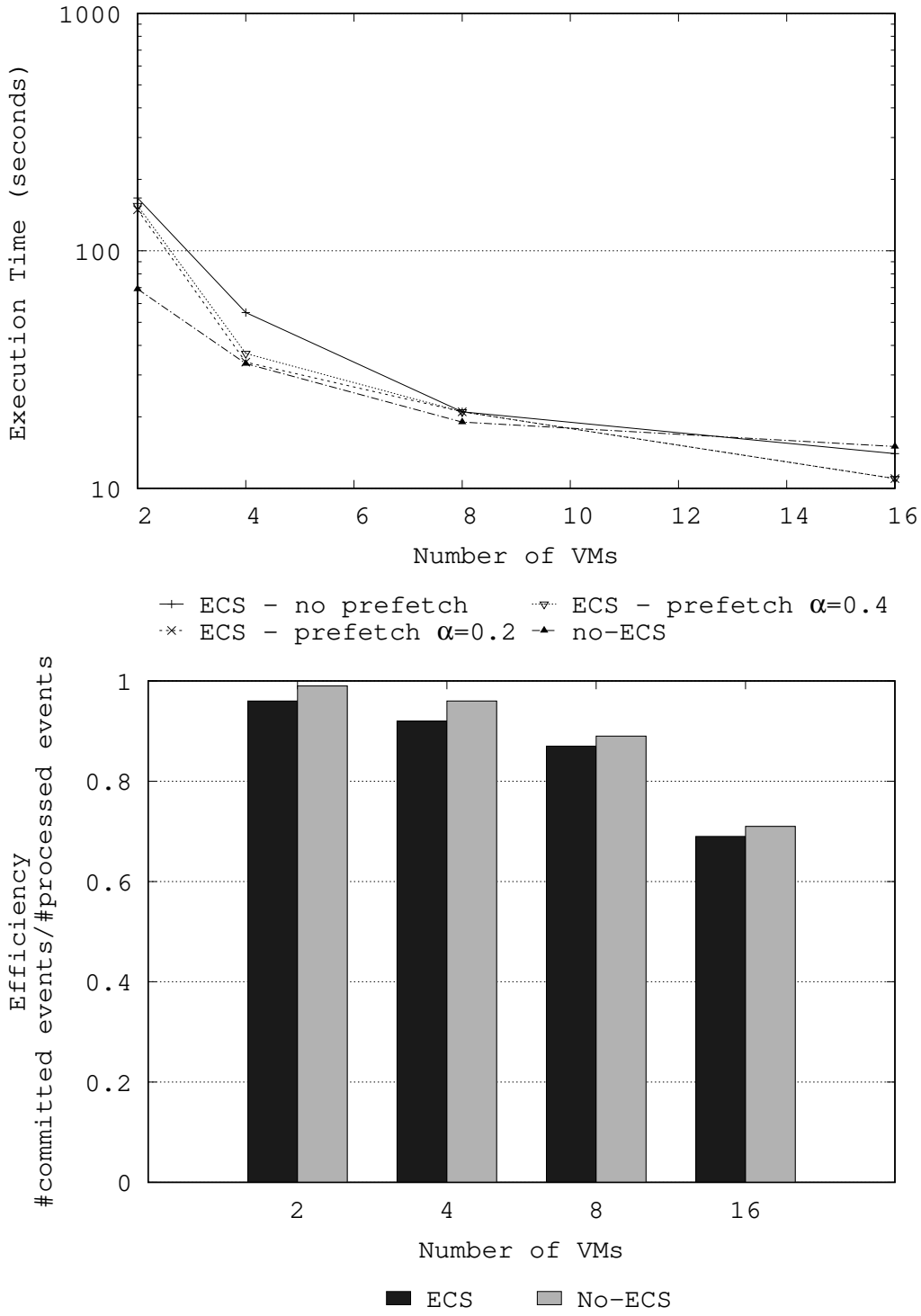


FIGURE 4.10: *Execution time and efficiency (with $\alpha = 0.2$) for the AWS deploy - NoSQL-store model - Single-vCPU VMs*

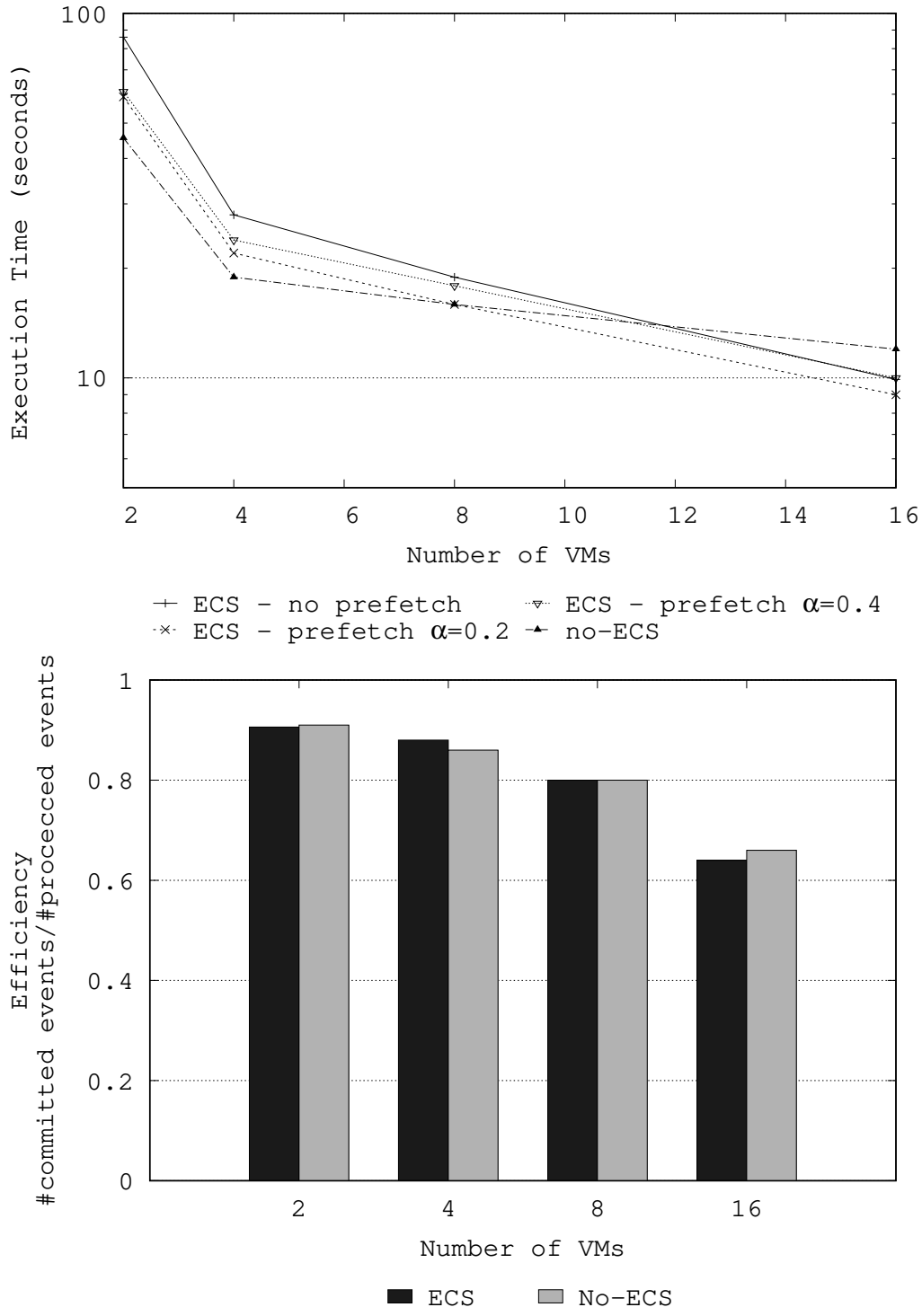


FIGURE 4.11: *Execution time and efficiency (with $\alpha = 0.2$) for the AWS deploy - NoSQL-store model - Dual-vCPU VMs*

CHAPTER 5

Approximated state reconstruction

Although this may seem a paradox, all exact science is dominated by the idea of approximation. When a man tells you that he knows the exact truth about anything, you are safe in inferring that he is an inexact man
— Bertrand Russell

Traditionally, the most common technique to cope with the rollback operation in speculative PDES is to generate a complete copy of the state of LPs at specific times in their simulation trajectory, to be used to restore a consistent state (i.e., the previously checkpointed state that is as close as possible to the inconsistency occurrence) whenever a straggler event is received. Of course, the cost associated with enabling this rollback operation is to be taken into account, along with the overhead introduced with the re-processing to bring the rolled-back LP to the timestamp of the straggler event.

The approximated rollbacks strategy targets an orthogonal approach to reduce the cost of both the state save and restore phases of a rollback. The primary rationale behind this is that, at least for some portions of a simulation run of certain models, it may be redundant for an LP to completely reconstruct a previous simulation state. In other words, there may be situa-

tions in which it could be acceptable to reconstruct a subset of the data that would have been restored upon a rollback, thanks to relaxed constraints on the accuracy of simulation results. To make an example, specific classes of stochastic simulations fall into this kind of situation, as they typically extract properties of the simulated system via statistical procedures.

More practically, as depicted in Figure 5.1, approximated rollbacks relies on the idea that the state of a logical process can be split up into two different portions:

core: this is the part of the simulation object's state which is guaranteed to be completely restored upon rollback execution;

non-core: this is the part of the state that is not ensured to be restored. The modeler can even determine how this portion should be restored in case reconstruction is required.

As for this last point, the possibility of partitioning the state into *core* and *non-core* portions is naturally dependent on the model of interest; thus, the interaction of the modeler is someway required. However, to minimize it, our solution provides transparent state-machine handling that seamlessly takes care of saving and restoring *core*/*non-core* state portions and exposes a user-level callback to be programmed in any way the modeler believes would be the best for the specific application, whose aim is to provide a custom restoration of the *non-core* portion.

Furthermore, we also allow the user to mark particular portions of memory as *core* or *non-core* at run-time, thus enabling dynamic changes of the amount of the state to be perfectly reconstructed and increasing the degree of freedom in implementing sophisticated models. To support such a kind of feature, the run-time environment can alternate between traditional rollback phases and approximated ones, allowing the simulation model to

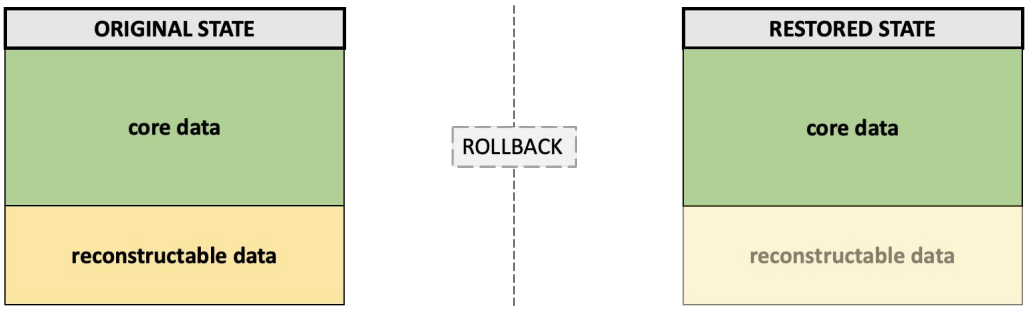


FIGURE 5.1: *The design of core and non-core portion of an LP's state*

decide whether to switch from one phase to another (and vice versa) on demand. This decision could be driven by the observation, at run time, of the current condition of a logical process along its simulation trajectory. By defining a set of predicates, the modeler may choose to trigger/disable an approximation phase whenever they stop or restart holding. In the following sections, the technical details of the approximated rollbacks solution will be examined in depth.

5.1 The Approximated Rollbacks strategy

Let's suppose that a logical process LP_x maintaining a local state S_x, during its normal execution, is required to roll back to an event e_j marked with timestamp T_j. Then, the approximated rollbacks allows restoring a state $S_x^{'}$ which contains a subset of the information originally kept by LP_x at timestamp T_j, S_x. More formally, given the following definitions:

- $LVT(x)$ the virtual time of a state x
- $I(x)$ the information currently stored in a state x
- $S^{'} \leftarrow S$ a relation between two states $S^{'}$ and S regulating the restoration of the state S after a rollback

we can describe the approximated rollbacks technique via the expression:

$$S' \leftarrow S \Rightarrow [LVT(S') = LVT(S)] \wedge [I(S') \subseteq I(S)] \tag{5.1}$$

. In other words, what equation 5.1 is stating is that when a state S' is restored due to a rollback, the corresponding complete state S, which would have been observed in a perfect reconstruction, contains strictly a super-set of the information stored in $S^{'}$: no state-variable that wasn't originally in the state S is inserted in $S^{'}$. This is an essential point as model developers need to be aware that they don't need to deal with new (thus unknown) variables during forward execution after an approximated rollback.

On the other hand, the subset relation between state information, namely $I(S^{'}) \subseteq I(S)$, also implies that whenever a state checkpoint is needed, the platform may be required to persist (much) less data on the available storage. Indeed, the majority of the techniques used to optimize state saving are based on the reduction of the size of this data, but this kind of approaches need to relate somehow the write operations performed during forward execution to the optimized state checkpoints. Instead, the approximated rollbacks technique tries to achieve this with a completely different approach: to restore $S^{'}$, we simply need to discard all the unneeded (namely, the *non-core*) part of the state S. Therefore, no tracing or backward execution mechanisms (as discussed in section 2.2.1) are required to implement it. Still, we only rely on the restoration of the state information $I(S^{'})$ rather than $I(S)$, the *non-core* and *core* portions of the LP's state.

However, the restoration of a state $S^{'}$ containing a subset of the full information may lead to inconsistent situations, and its feasibility needs to be evaluated. To make this clearer, we provide an example of possible incorrect situation of a simulation trajectory caused by an approximated state reconstruction. This situation is depicted in Figure 5.2. Here, two equiva-

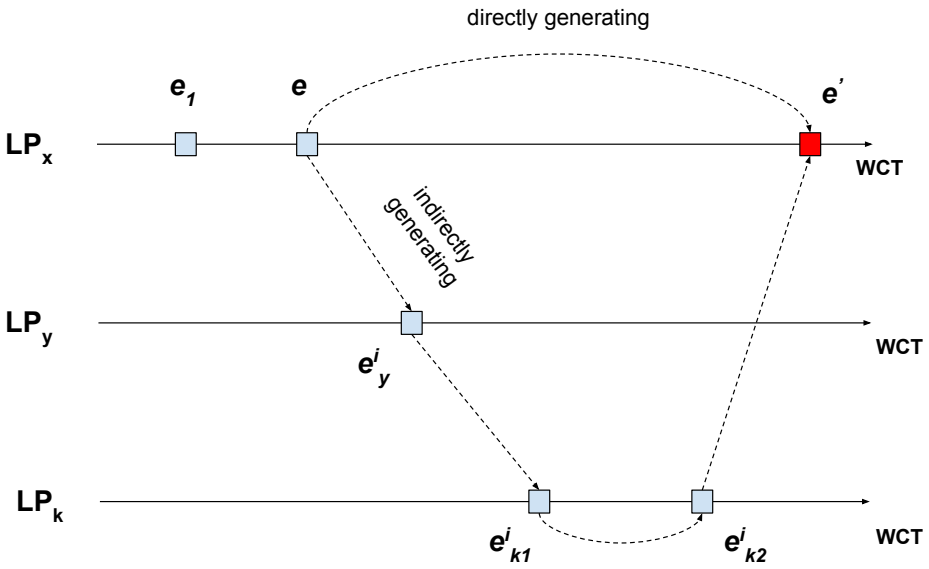


FIGURE 5.2: *Direct and indirect generation of an event e'*

lent (although different) possibilities to schedule an event e' at time $T(e')$ in the future with respect to the event e are presented. In the first one, LP_x generates e' further on its timeline via the execution of e, while the second one relies on the generation of such an event via a chain of events scheduled at different LPs, which are ultimately spawning e' at LP_x. Suppose, then, that event e' entails logic that expects to access state information related to the state initially observed by the event e, namely S. An example of such a behavior in simulation model could be represented by a model logic in which event e' needs to remove from LP_x's state a previously installed key K, as it could happen in the NoSQL model presented in the previous chapter.

If an approximated rollback occurs at time $T > T(e)$ (which besides does not undo the execution of e), such that the state S' rather than S is restored, then executing event e' after the restoration phase would lead to an exception, as the information required for its execution, which was contained in S, is lacking in the memory view observed by this event. That is to say that the approximated rollbacks technique needs to deal with the problem

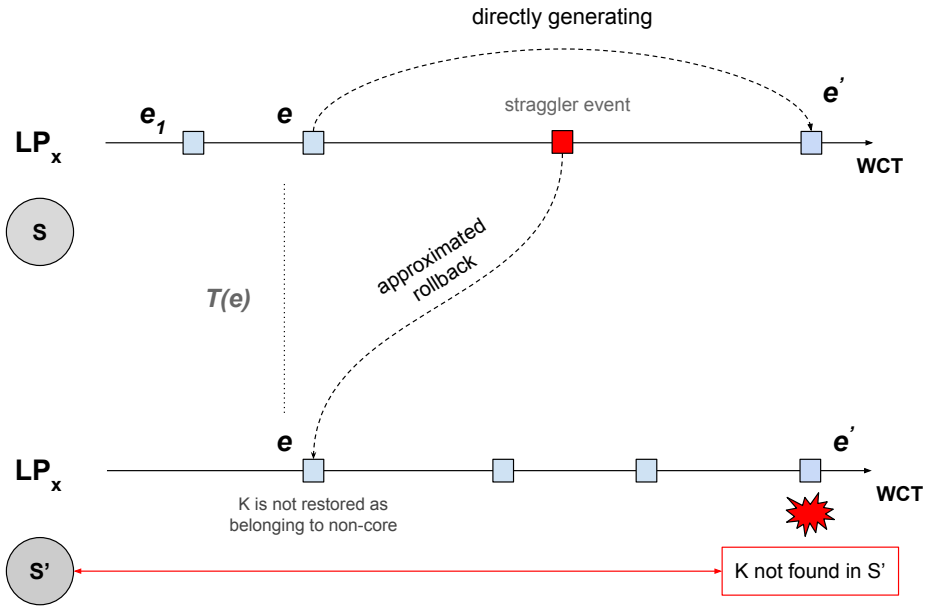


FIGURE 5.3: *Example of exception in retrieving a missing field in an approximately restored state S'*

of execution of events that expects the disappeared (i.e., non restored) *non-core* portion of the state of an LP. A graphical example of such a kind of exception situation is depicted in Figure 5.3. Therefore, the approximated rollbacks scheme provides two different solutions coping with the above discussion.

First of all, the trivial approach would be to mark the information leading to an exception, namely the key K, as part of the *core* portion of the state LP_x. The system would, in this case, guarantee that the exact value of K will be persisted in subsequent state restorations. The other approach, which is more sophisticated, is related to the ability of this technique to dynamically switch between approximated and non-approximated (namely, precise) rollback phases. Considering the example of Figure 5.2, whenever event e schedules (or is willing to schedule) e' requiring to access the full copy of S the system is able to switch to a non-approximated rollbacks phase and then switch back at run-time to an approximated phase at time

$T(e')$. As a matter of fact, it is safe to execute in this mode outside the boundaries of this time interval.

Before discussing about the technical support of such a solution, a final point needs to be addressed. Indeed we, outlined how this technique can manage the correctness of the execution in terms of crashes related to an inherently inconsistent situation generated by the solution itself. However, when dealing with the statistical correctness of the simulation run, a modeler should take into account that the approximated rollbacks scheme can affect the final outcome of a stochastic simulation, as the recorded data by LPs in their own simulation trajectory is strictly dependent on the definitions of the *core* and *non-core* portions of their state.

This aspect is clearly related not only to the selection of the parts of a state to reconstruct perfectly but also to the way in which this reconstruction is carried on. In particular, the approximated rollbacks technique complements the restoration of the state S with the execution of a user-defined callback taking $S^{'}$ as a parameter and modifying it to minimize the actual distance with S, namely the difference $I(S) \setminus I(S')$. If we consider stochastic simulation, the implementation of such a function can somehow be exhaustive in performing a guess on the missing information between the restored states. Of course, the implementation of this function, which we will refer to as *RestoreApproximated* (RA) in the remainder of this chapter, can introduce an overhead to the state reconstruction procedure embedded in the runtime. However, while rebuilding a state $S \approx RA(S^{'})$ includes this cost, the dependence of the number of rollbacks to this cost is still avoided.

As it could be expected, effective implementations of the RA callback carefully consider the selected information to be restored in terms of relevance concerning the simulated stochastic process. Being this intrinsically related to the model of interest, it is left to the application programmer to

determine what information is suited for the approximated reconstruction.

Overall, dealing with large state sizes and coarse-grain events (possibly updating large portions of the state) in a speculative PDES application is no longer a concern when employing approximated rollbacks, especially when $I(S')$ is a very reduced subset of $I(S)$. On the other hand, large state sizes and coarse-grain (write-intensive) events represent a challenging scenario for all the other techniques, including reverse-computing based ones.

5.2 The technical support

Let us now discuss how the approximated rollbacks solution has been implemented in the ROOT-Sim PDES platform, although all the aforementioned concepts are general and can be ported to any custom run-time environment. As hinted in previous chapters, this simulator relies on a chunk-based memory allocator (DyMeLoR, [PVQ09]) which maintains tiny metadata to keep track of allocated, dirty, or *uncommitted* (i.e., that can be rolled back due to non completed speculative operations) memory areas. This is done by intercepting calls to standard functions dedicated to memory management, namely `malloc` and `free`. Thus, ROOT-Sim offers the possibility to the final user to thoroughly control the memory layout of any logical process by means of calls to conventional functions of the standard C library.

As for rollback/restore operations, in such an architecture, the checkpointing procedure would mean to copy the currently allocated chunks into log buffers, while the restore procedure would perform the opposite, that is, moving back the logged chunks into their original memory locations. In particular, given a state S observed by a logical process during event processing, the corresponding information set $I(S)$ is composed of two sections: *i)* the set of the addresses related to the LP's state active memory chunks

and *ii)* the actual content of the memory chunks relative to the LP's state. When dealing with the approximated restoration of a state S, the newly generated $S^{'}$ might miss either one of these two portions, according to the implementation choice taken. We devised our simulator to opt for the approximation of the memory chunks rather than the memory addresses of active chunks. This means that $S^{'}$ results to be an approximation of S in the sense that it misses some memory chunks that were kept in the memory view of S at the time when S was available while processing events during its forward-execution. From this assumption, it follows that:

1. A memory chunk cannot be restored into $S^{'}$ without restoring its content as well;

2. Given two memory chunks belonging to the *core* portion of the S state, any pointer between them is guaranteed to be meaningful, i.e., is pointing to a non-approximated memory area.

In particular, we define the second of the two points as the *strong core-memory connectivity* property, and it is explicitly making the whole implementation dangling-pointers free. Note, however, that this only stands in an inter-chunk fashion: the *strong core-connectivity* does not apply to an individual memory chunk. That is, there could be a situation in which a chunk belonging to $S^{'}$ points to a memory address not mapped to any chunk in $S^{'}$, while this cannot happen with pointers falling within the boundaries of a chunk. To have a broader picture of these concepts, we describe such a kind of situation in Figure 5.4.

Here, we show the memory layout of the two states S and $S^{'}$, being the first the state where an LP passed through during forward execution and the latter the corresponding restored state after an approximated rollback. The memory areas colored in red represent chunks of S which are no longer

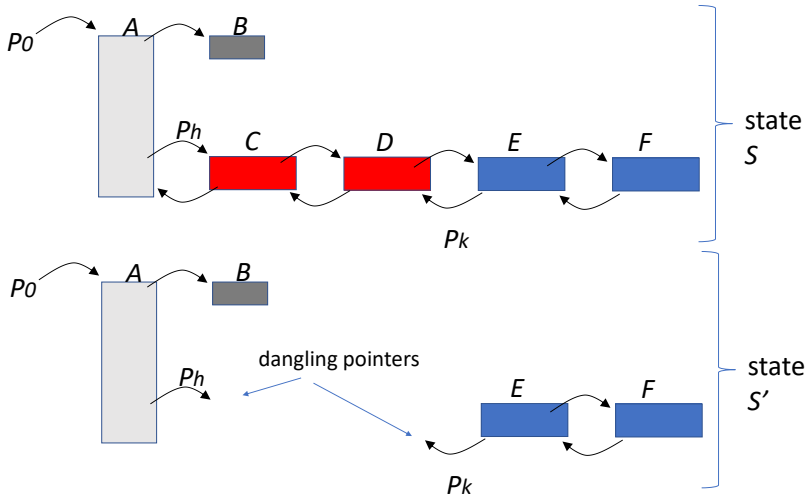


FIGURE 5.4: *A possible post-approximated-rollback dangling pointer situation*

present in $S^{'}$. However, all the memory chunks found in $S^{'}$ have identical content of the corresponding entries of S (depicted in blue), namely satisfying $I(S^{'}) \subseteq I(S)$. In the picture, the pointer p_0 refers to the state information at the beginning of S's memory layout, namely, chunk A; such a pointer is the one returned in the event handler routine whenever an event has to be processed. Thus, *strong core connectivity* is respected even though chunk E presents a dangling pointer p_k which has no destination, as well as p_h belonging to chunk A.

Therefore, considering the property mentioned above, a chunk B can belong to the *core* portion of state S if and only if the chunk A keeping the pointer to it, i.e., satisfying the relation $A \xrightarrow{\text{p}} B$, belongs to the *core* memory portion of S too—the only exception, of course, is the initial pointer p_0, which cannot be dangling as we need to reach the very beginning of the state. This, in turn, implies that the state $S^{'}$ is fully reachable starting from the initial pointer p_0, which is a crucial property when considering the feature, offered by the approximated rollbacks scheme, of manually transforming portions of $S^{'}$ via the RA callback function. Indeed, full reachability

is a central aspect as the model developer may be interested in reconstructing the whole original state stochastically to better approximate the overall original state S. In our design, in the body of the RA function, it is deterministically possible to recognize dangling pointers since, as depicted in Figure 5.4, developers are aware whether an accessed memory chunk belongs or not to the *core* section of the state.

To unburden programmers from the cost of determining whether the pointer is dangling or not, we devised a custom type of data structure, named `discriminable_pointer`, such that the associated virtual address, used to point to memory chunks of a state S_x, is coupled to a bitmask discriminating whether the pointed chunk is residing into the *core* or in the *non-core* portion of S_x. In this way, while implementing RA, all buffers belonging to the passed state S_x can be safely scanned in order to either inspect relevant information if they belong to the *core* portion or fill them with stochastically meaningful values if they need to be rebuilt otherwise. Moreover, the APIs (or macros) offered to the final user provide the possibility to:

- Map the `discriminable_pointer` structure to a more readable object, namely a pair of `<address, type>` where `type` is an abstraction built by parsing the `discriminable_pointer` bitmap and hence corresponds to either *core* or *non-core* depending on the nature of the address.

- Read the pointer value and its state, namely dereferencing it and understanding whether it is dangling or not

- Assign a new value to the pointer, ultimately updating the `address` field.

- Switch the state of an address from *core* to *non-core* and vice-versa,

API Macros
`void set_pointer(struct discriminable_pointer, void *address, int type)`
`void set_pointer_type(struct discriminable_pointer, int type)`
`void set_pointer_address(struct discriminable_pointer, void *address, int type)`
`int get_pointer_type(struct discriminable_pointer)`
`void *get_pointer_address(struct discriminable_pointer)`

TABLE 5.1: *Approximated rollbacks API macros*

API Functions
`int core_memory(void *address, int command)`
`void start_approximated(void)`
`void end_approximated(void)`

TABLE 5.2: *Approximated rollbacks API functions*

namely modifying the `type` field.

More technically, all the mentioned features were implemented in the DyMeLoR memory manager, which has been enhanced with specific API functions and macros, listed in tables 5.2 and 5.1 respectively.

In particular, we provide in the following an in-depth description of their functioning:

`core_memory`: this is the entry point to be used in order to set a given memory chunk as *core* or *non-core*. Specifically, the arguments that can be passed to this API are the memory address of interest and the `command` flag which can assume either the `INCLUDE` or `REMOVE` values. These are the flags specifying whether this address needs to be *included* or *removed* from the *core* memory portion of the state it belongs to. Note that this API can be called at any time during the execution of events at user level, so that model developers can dynamically set specific areas to the preferred value. Moreover, all the invocations of this function are rollbackable in the sense that the memory manager can transparently revoke their effect. This entails

that at any simulation time T, both the simulation platform and the application logic observe the same *core* memory chunks, independently of the changes made after time T but before the rollback brings back execution to time T. Thus, the model developer can focus only on the forward execution of the events, and concentrate on a sequential style programming model rather than dealing with ambiguities about *core*/*non-core* memory areas.

`start_approximated`: this API allows the application coder to start an approximated rollbacks phase, in which some portions of the state may be discarded during post-rollback restorations. This API is, under the hood, marking as *non-core* the specified parts of the state by modifying the metadata related to them.

`end_approximated`: this is the counterpart of the previous API, thus restoring the mode of checkpoint/restoration to precise. This API was also designed to cope with the scenario bringing to exceptions, depicted in Figure 5.3. If the model developer is aware of the fact that some event e' needs to access a portion of the state that could be discarded via an approximated restoration, this API can be invoked so as to manage such a kind of inconsistency.

Another important aspect regarding these APIs is related to the activation point of the RA function. This callback is, in fact, invoked right after the state $S^{'}$ is restored due to an approximated rollback and before the execution of any subsequent event. Therefore, at the moment of its invocation, RA needs to find a correct view of the `discriminable_pointer` variables lying in the restored state. Hence, we only demand the final programmer to mark these pointers correctly before starting an approximated phase (via the `core_memory` and `start_approximated` APIs, respectively)

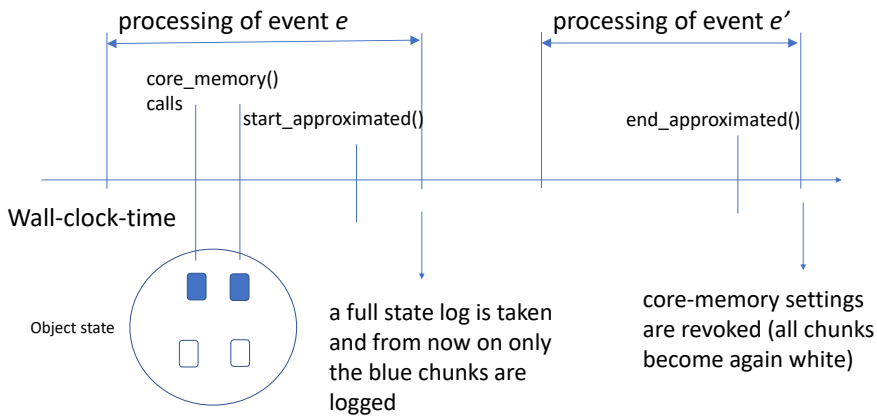


FIGURE 5.5: *A possible execution timeline of approximated/precise rollbacks phases*

and throughout the approximated phase itself.

Moreover, it is worth mentioning that the programmer is free in the usage of the `discriminable_pointers` in the sense that the run-time can still rely on classical pointers to carry on state restorations. Indeed, this could be the case in situations in which the state S keeps pointers to disjoint collections of data, a subset of which needs to be maintained in the *core* portion of the state. In this situation, the run-time could implicitly assume that the *non-core* collections need to be stochastically rebuilt (or repopulated) after the occurrence of an approximated rollback.

In Figure 5.5, we show a possible simulation timeline involving the usage of those APIs. In the picture, event e is scheduled at time $T(e)$ and, while being processed, application-level code marks some memory areas as *core* via the usage of `core_memory`, after which an approximated phase is started via the `start_approximated` callback. Now, a complete checkpoint is automatically logged by the run-time, and only memory chunk previously marked as *core* shall be perfectly reconstructed during a rollback, henceforth. Furthermore, whenever $T(e)$ is reached, a call to `end_approximated`

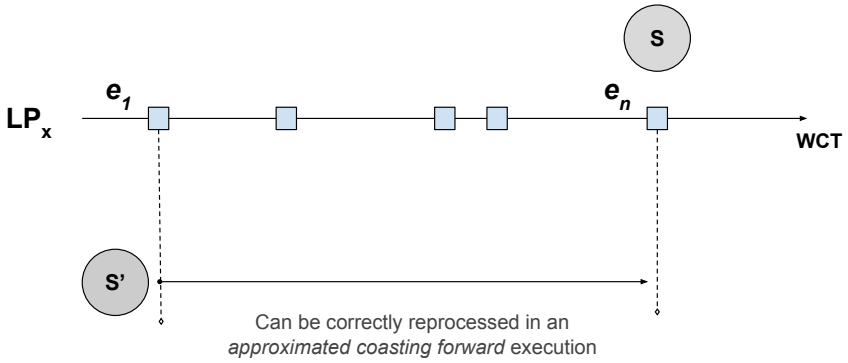


FIGURE 5.6: *An example of a scenario in which approximated coasting forward can be safely executed*

is issued, effectively eliminating any difference between marked and unmarked (i.e., *core* and *non-core*) memory chunks. At this point, the simulation platform switches back to the standard checkpointing based on complete reconstruction of the state, i.e., it starts a *precise rollbacks phase*.

Finally, a last optimization of the overall scheme needs to be discussed. Indeed, until now we discussed how the approximated rollbacks strategy can leverage from the reduction of costs in storing/restoring states by handling only *core* information of a state originally passed through at some simulation time. Nevertheless, this idea can be extended considering the possibility that events processed by a logical process can lie in a simulation window in which they can be correctly handled by means of only approximated rollbacks.

To better explain this, let us consider the simulation trajectory portion of LP_x described in Figure 5.6. Assume that the sequence of events $e_1 \dots e_n$ can be handled by approximated rollbacks, i.e., an *approximated rollbacks phase* has started at e_1 and has not finished yet after execution of e_n. In this case, we can state that no run-time anomaly could take place whenever the sequence of the very same events needs to be reprocessed after an approximated rollback occurs. Then, if this is the case, it means that the

reprocessing of these events would result in a simple series of state updates, an operation that in a precise rollback phase we would refer to as *coasting forward*. Thus, during the simulation time window between the timestamps $T(e_1)$ and $T(e)$ we can adopt the *approximated coasting forward* optimization, since it is starting from an approximately reconstructed state. For the same reasons explained in this section, this kind of coasting forward will produce better performance with respect to the standard one.

5.3 Benchmark applications and results

To start describing the results obtained in the approximated rollbacks evaluation, we should first analyze the test-bed environment and the applications chosen to carry on experiments. First of all, we ran our experiments on top of two different computing platforms:

1. A bare metal platform, based on 4 AMD Opteron™ 6168 multicore processor with 12 cores each for a total of 48 cores and equipped with 128GB of RAM.

2. A virtualized cluster, based on three virtual machines rented on Amazon Web Services, each of them being an m5.8xlarge instance equipped with 32 vCPUs and 128GB of RAM. The selected AWS region for all of them is the same, along with the availability zone to reduce communication latency. We recall that such kinds of AWS instances are run on top of Intel Xeon® Platinum 8175 CPUs, running at a speed of 3.1GHz.

As for the benchmark applications, we based our experiments on a real-world and a synthetic model. We discuss the details of these implementations in the following sections.

5.3.1 PHOLD benchmark

The first application we selected to carry on experiments is an artificial one, in the sense that its purpose is not to model a real-world scenario instead to test the PDES platform features directly. Initially, the HOLD algorithm was designed to evaluate sequential discrete event simulations; therefore, the enhanced Paralled HOLD (PHOLD, ([M.90])) precisely targets parallel executions of discrete event simulations.

The key functioning of the model can be summarized in the following: an algorithm schedules events uniformly randomly across all the logical processes, which while executing one of those events could either generate a new one targeting themselves or schedule an event to a different LP. Further, the targeting LP is still chosen uniformly randomly, and each event is scheduled at a time equal to the LP's logical virtual time plus a value drawn by an exponential distribution.

In our reshuffling of this model, every logical process at simulation startup generates multiple linked data structures in its own state. The content of each entry of these structures is not really of interest, as they only keep their size and a buffer filled with random data. Rather, what is important are the actions included in an event associated with the lists: every logical process, while executing an event, loops through its data structures, updates them by writing random data and, with a certain probability, removes one of them from its state, sending a copy of it to a randomly chosen target LP. On the other hand, the receiver LP chains a copy of the delivered data structures into its state and keeps it updated following the logic described. Therefore, during the whole simulation run, the overall size of the data is kept constant across all LPs.

Even though this is a non-real application, PHOLD somehow resembles some widespread situations in a PDES environment. First, the data

structures exchanged between logical processes replicate a traditional PDES situation where entities communicate (as in the robot explore model, described in the previous chapter). Secondly, being the state of a logical process very likely to be completely different from the state of others, the model is putting in place a situation in which real-world entities are interacting with reciprocal unknown data (again, the robot explore model is an example of this).

As far as the approximated rollbacks scheme is concerned, our PHOLD implementation relies on two recurrent phases. The first one is the standard *precise rollback phase* , in which checkpoints are taken in the traditional manner (according to the implemented policy, e.g., full or incremental checkpointing). After persisting for a certain amount of time, the model switches (via the invocation of API functions described above) to an *approximated rollbacks phase*. While in this condition, only the metadata regarding the number of buffers within a simulation state is marked as *core* memory. Moreover, the amount of state to be reconstructed when invoking the RA function to approximately restore it was parametrized. This feature was specifically designed to better analyze the impact of the overhead introduced by the user-defined state reconstruction callback function.

As far as the configuration of this model is concerned, simulation runs were launched with 100 logical processes. Thus, given that we launched multiple runs with different amounts of involved thread/cores, we were able to test scenarios in which the number of LPs per thread/core was spanning from 2 to 100 for the private deploy and from 1 to 50 in the AWS cluster. This results in an interesting situation since it is incrementing the probability of rollbacks as the divergence of local virtual times increases, ultimately provoking a non-minimal rollback incidence.

In Figure 5.7, we first analyze the single-node deploy running our vari-

ation of PHOLD, entailing a 10K buffer migrations long busy loop. In our resembling, we diversified the amount of reconstructed state by the RA function to span over 25%, 75%, and 100% of the whole state size. The results show that the approximated rollbacks solution can provide better levels of performance in situations in which the amount of rollback is dramatically increased, i.e., when the number of core counts gets higher. This is a sign that the solution is able to optimize the costs related to checkpointing the state, which turns out to be a critical operation when the level of concurrency on a core/thread is growing, as this entails more frequent rollbacks.

Although all state reconstruction configurations can provide better performance, we can notice from the plots that the total reconstruction brings the highest level of increase in terms of overall execution time. On the other hand, in the worst-case scenario, coinciding with small core counts, the precise reconstruction presents an execution time that is better than the 75% reconstruction by a factor of 1.2. This can be explained by the fact that the approximated rollbacks scheme is introducing noise into the sequence of operations on the critical path of the runtime, ultimately generating a slightly higher number of rollbacks. With a small number of threads, though, the costs associated with the RA function could fail in paying off the ones of precise rollback restorations.

In Figure 5.8, we show the data related to the AWS deployment for a 25% approximated reconstruction of the state. Here, the incidence of rollbacks is even raised by network latency which is skewing event interaction between remote LPs. We show the measured execution times, the latencies of checkpoint restoration, and the overall memory usages. Nevertheless, even in such a distributed scenario, the approximated rollbacks scheme can reduce execution times from 25% to 50% (first plot of Figure 5.8), a straight-

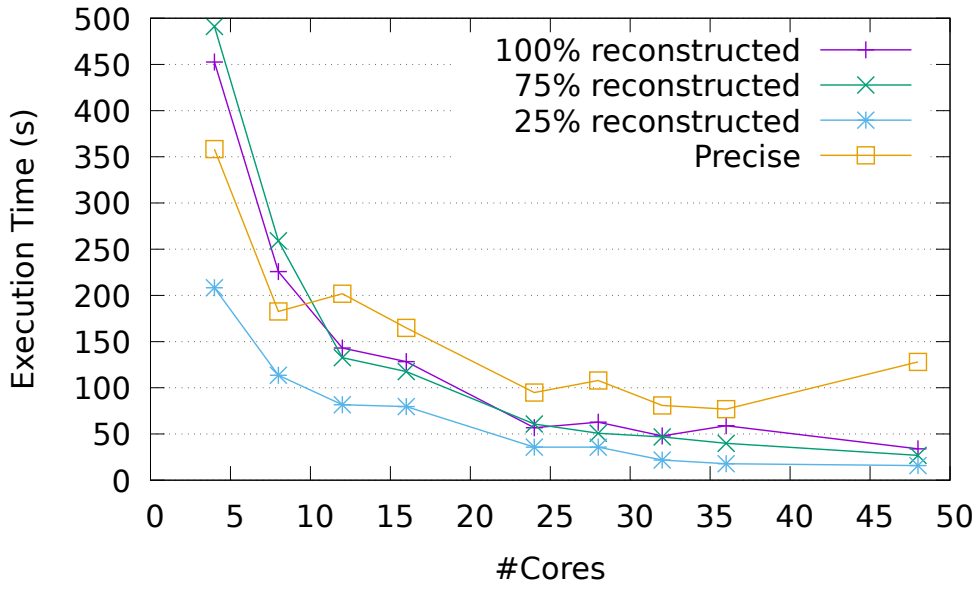


FIGURE 5.7: *PHOLD execution times varying the amount of approximately reconstructed state, in the private deployment.*

forward consequence of the checkpoint restore cost decrease (second plot of Figure 5.8).

Furthermore, as the degree of parallelism increases (thus, the probability of rollback), the approximated rollbacks scheme presents even better performance: this is a sign of the fact that this solution scales well, and that could allow more significant and complex problems to become tractable, yet with a high level of programmability.

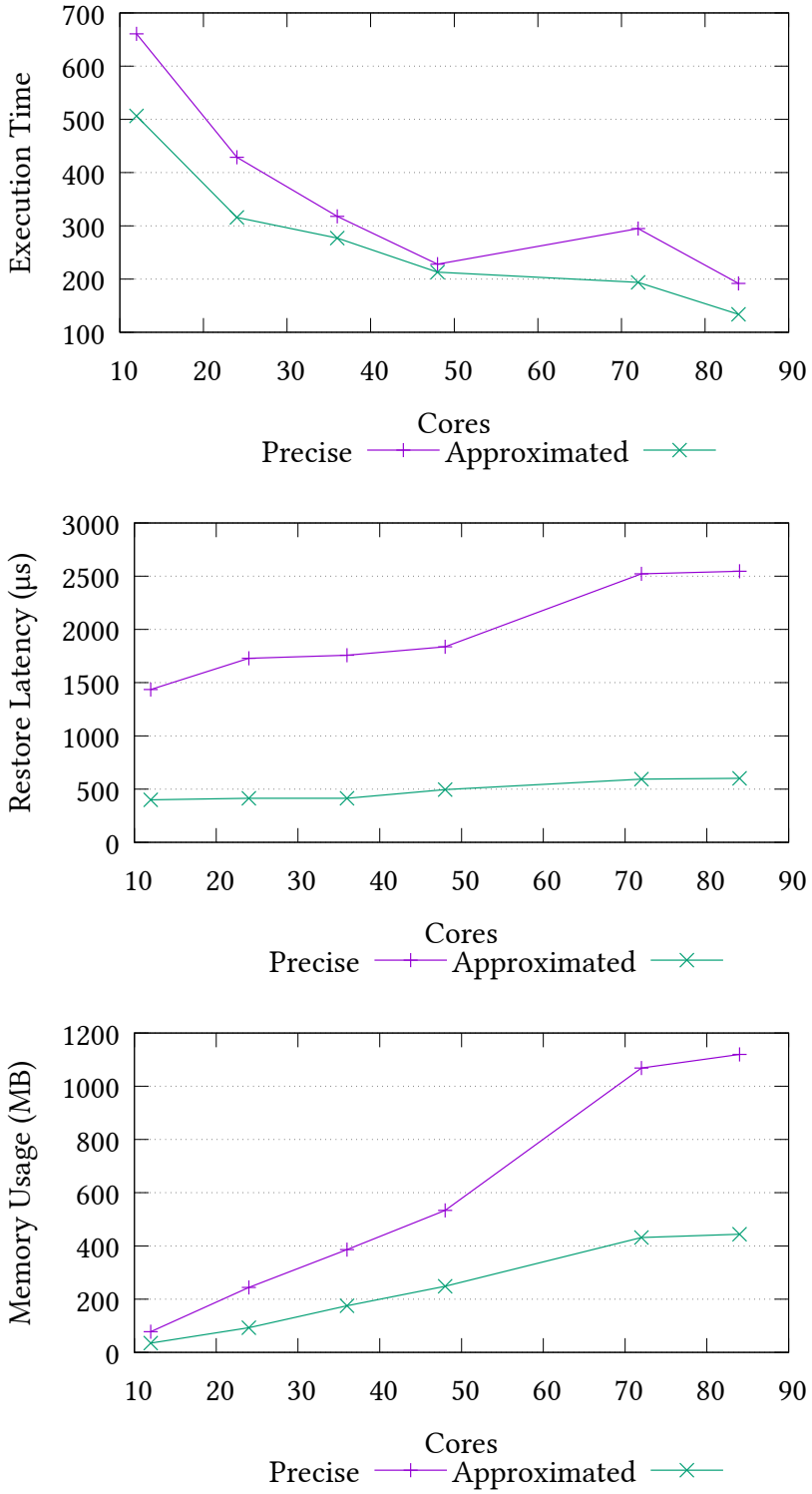


FIGURE 5.8: *PHOLD execution times, restore latencies and memory footprint with 25% state reconstruction for the AWS deployment*

5.3.2 The Susceptible-Infected-Recovery benchmark

The second benchmark application which was chosen to provide experimental results is a real-world one, mimicking an epidemiological Susceptible-Infected-Recovery (SIR) scenario. This kind of model was designed to investigate the spreading of diseases in medium/large demographic areas, specifically targeting tuberculosis infections. The implementation of this model took as inspiration a first implementation provided by the Barcelona Supercomputing Center (BSC), whose details can be found in [MSGNCG+15].

Similarly to the one described in 4.2.1, this is an agent-based model in which each entity moves around a delimited area and interacts with other humans according to a specific logic. Also here, logical processes represent regions of the area, and the state of a logical process is composed as follows: an individual has a current health status, which at a certain time can be only one among healthy, under-treatment, infected (i.e., incubating the disease but still not ill), sick, and recovered. Moreover, other individual-specific parameters are stored, as they could drive their infection behavior. Those parameters include age, gender, native-immigrant origin, possible risk factors (e.g., smoking, addictions), and possible immunodepression (i.e., AIDS). Also, once a person is infected, the presence (or not) of pulmonary cavitation is taken into account. Additionally, an LP modeling a region keeps in its state a counter of the agents currently residing in it, one for each of the mentioned possible infection statuses of agents and their characteristics. To easily distinguish them, agents belonging to different states are mapped in various data structures, namely hash tables, and not all of them are marked as *core* memory. In fact, the agents we considered as *core* are the ones belonging to the healthy and recovered statuses. In contrast, all the other agents are restored in an approximated way upon rollbacks; hence the platform will not record any additional information but

the actual overall number of them, restoring in a stochastic way the personal parameters like age, gender, etc. This last point, however, is a crucial aspect as stochastically recomputing those parameters when the RA function is invoked inevitably leads to a shift in the features of the population concerning the default parameters fixed at simulation startup, and this can generate, for a specific run, slight or considerable changes in the outcomes related to the spreading of the disease. In other words, being this a real-world application modeling a phenomenon driven by accurate parameters, modifying them at run-time could bring to different conclusions and predict disease spreading situations in various manners.

The tuberculosis model was configured as follows: 1,600 logical processes move into square regions of $0.06km^2$, for a total of $96km^2$. The overall number of agents involved in the simulation amounts to 1.6 million, and agents move according to a random walk in the available areas. Hence, each LP modeling a region handles, on average, 1,000 individuals (16,000 people per square kilometer is the actual population density). Indeed, to comply with this model's original resorting, this configuration maps a medium metropolitan city, namely Barcelona, in 2019. As for the default parameters, at simulation startup, 95.59% of the population is marked as healthy, while 4.28% is instead infected. Only 0.12% of the individuals start as recovered, and the remaining 0.01% is either sick or under-treatment. Even if the number of ill individuals could appear to be significantly reduced, agents are frequently moving over the areas, and the dynamics of the movements are fast, thus bringing the simulation scenario into a pandemic environment quite quickly during the run. Finally, the simulated time taken into consideration sums up to 10 days.

To understand how we carried on experiments, we recall that we called χ the state saving interval, which represents the amount of time between

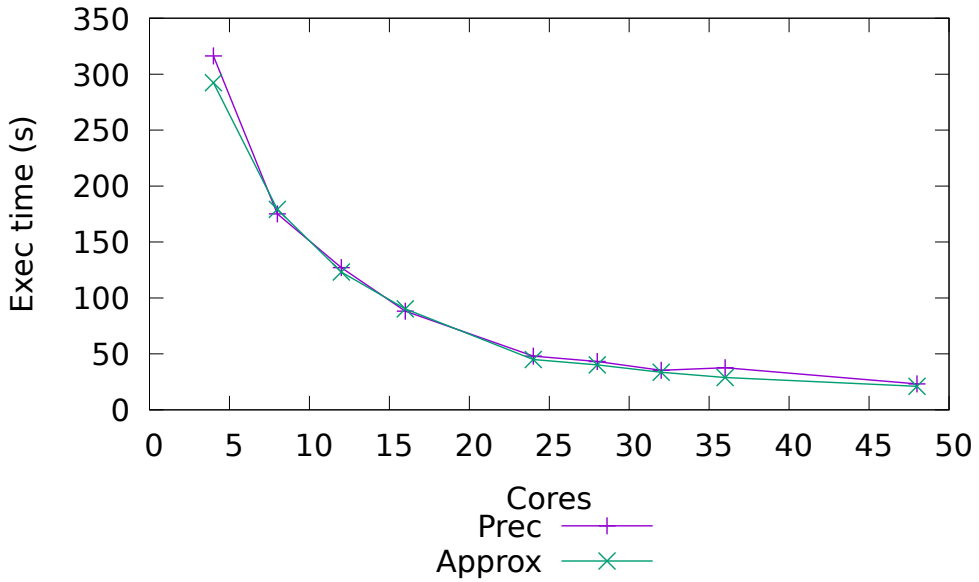


FIGURE 5.9: *TBC execution times for precise and approximated rollbacks schemes, in the private deployment*

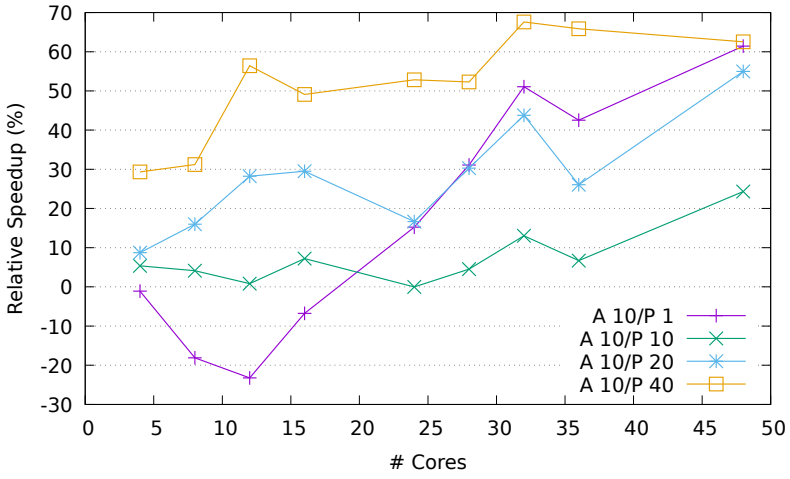
two consecutive checkpointing operations. This is a tunable parameter in the ROOT-Sim simulator, and we modified it to span over 1, 10, 20, and 40: using different configurations of this parameter can provide a broader picture of how exploiting infrequent checkpointing of *core* memory portions can impact overall performance.

As a preliminary result, we show in Figure 5.9 the execution times of precise vs. approximated runs performed on the private single-node architecture, having set χ to 10 (the default parameter for our simulation platform). Even though the resulting trends are very similar between the two solutions, the primary purpose of this plot is to prove that the approximated rollbacks scheme can scale in performance when increasing computing power. Indeed, this is the starting point to a more in-depth analysis of the overall log/restore approximated support. The rest of this assessment is on simulation runs relying on the same sequence of pseudo-random numbers, meaning that the seed from which random numbers to be possibly rolled

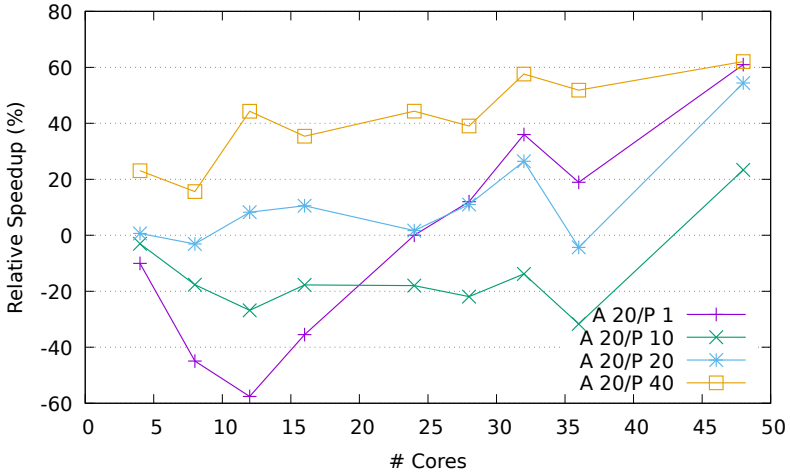
back has been consistently and specifically set. All the trends refer to results averaged over five different runs, and we denote with the letter A runs relative to the approximated rollbacks solution and with P the precise counterpart scheme. Numbers after each letter express the selected checkpoint interval.

In Figures 5.10a, 5.10b and 5.10c, we show the performance comparison between precise and approximated rollbacks schemes while varying the checkpoint interval value and the number of involved CPU cores. We explicitly decided to plot the results from all configurations in the same chart to visualize the differences better. More in detail, the relative performance is measured by subtracting the overall precise and approximated rollbacks execution times, and dividing the result by the total execution time of the precise run. Observing the trends, we can understand that most of the time, the approximated rollbacks scheme can benefit from increased performance, except for the runs associated with the checkpoint interval set to 40 for the approximated approach and lower values for the precise solution. Indeed, these situations present a negative speedup, and we explain this phenomenon by the fact that with these configurations, the simulation runs present an increased amount of rollbacks, characterized by a relatively short length in simulation time. This, in turn, means that the approximated solution is forced to re-execute a non-negligible number of events with respect to the precise rollback scheme. All the advantages related to the reduction of checkpoint or restore operations are therefore dramatically degraded.

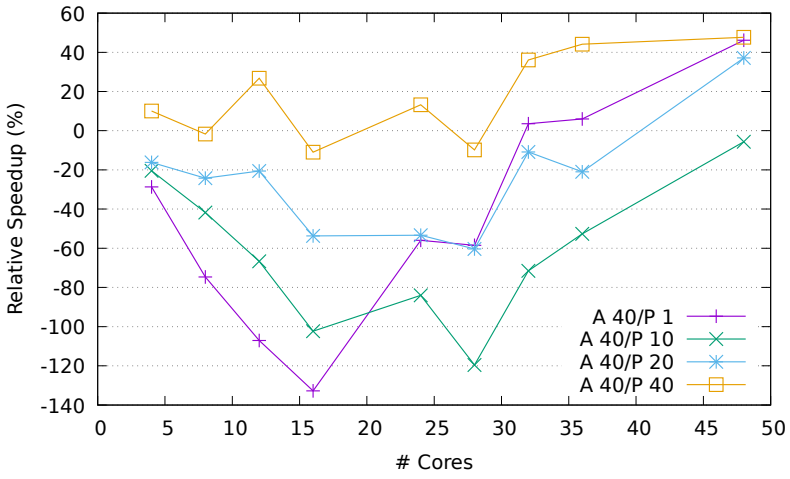
Another aspect of investigations of the experiments we carried on concerns the effect of state size in this model, compared to the dynamics of the approximated rollbacks solution. Indeed, from what we discussed in this chapter, shrinking the state size of a logical process should result in a less evident impact of the approximated rollbacks solution as the cost of log and



(A) *Approximated* $\chi = 10$



(B) *Approximated* $\chi = 20$



(C) *Approximated* $\chi = 40$

FIGURE 5.10: *TBC epidemic model - relative performance of the approximated vs precise rollbacks.*

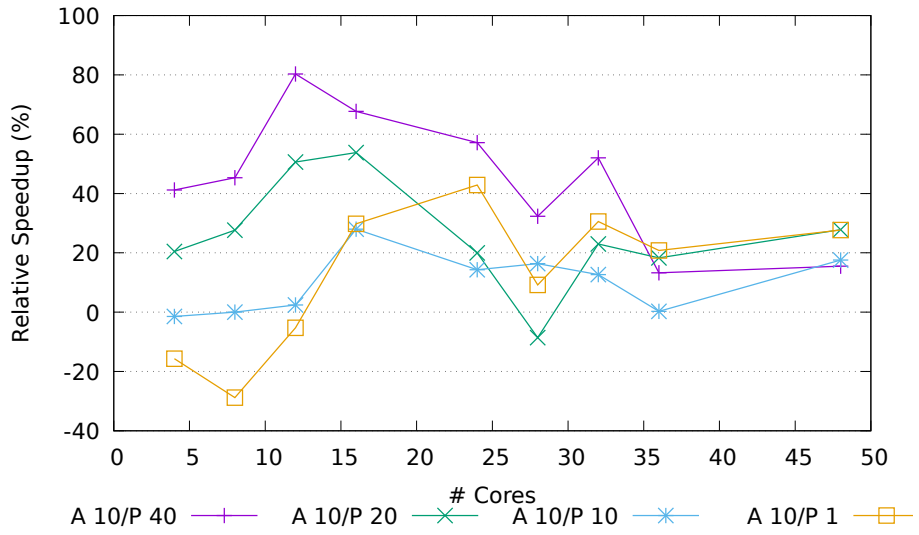


FIGURE 5.11: *TBC model's relative performance of the approximated ($\chi = 10$) vs precise rollbacks with scaled down workload*

restore operations of *core* and *non-core* portions becomes negligible: if the state is composed by a tiny amount of data, we won't get any significant improvement by splitting up a fraction of that already small amount of information into reconstructable/non-reconstructable data. In particular, we significantly reduced the load at each LP to make them manage an average of 20 agents rather than 1000 of previous results. As a matter of fact, we proved this behavior by running the approximated scheme with $\chi = 10$, the results of which are plotted in Figure 5.11. From the chart, it is evident that the resulting relative speedup is less favorable compared to the one depicted in Figures 5.10a - 5.10c.

As a final experiment, we also chose to measure the accuracy of simulation outcomes in runs enabling the approximated rollbacks scheme. We recall that the simulation results could be affected by the logic introduced in the RA callback as it could bring to significant shifts in the population's characteristics, ultimately affecting the spreading of the virus. In Figure 5.12, a plot of the percentage of agents in each state for each simulated day

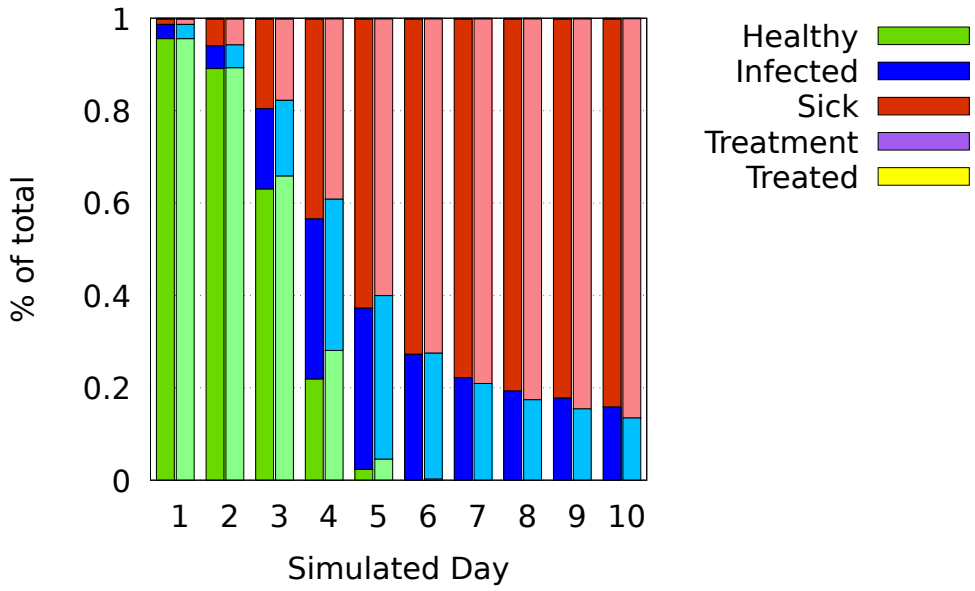


FIGURE 5.12: *TBC model's results accuracy of the approximated (lighter) vs precise (darker) rollbacks.*

is presented. We recall that the state of an agent can be either healthy, infected, sick, under-treatment, or recovered and that the *core* portion of the state of an LP contains the counter of the currently present agents belonging to each class and their characteristics. Also, we show the simulation results for runs configured with $\chi = 10$.

Analyzing the depicted results, we can observe that the stochastic divergence introduced by the approximated scheme is reduced. In fact, the difference in the number of agents belonging to the different states is comparable between the two checkpointing approaches. This can be considered a proof of the fact that the approximated rollbacks technique is a viable solution to provide meaningful simulation results if we accept a reasonable error range in the produced outcomes. Furthermore, combining these results with the relative performance increase we outlined in previous charts, we can confirm that if the final modeler is mainly interested in trends of the overall simulated study, the approximated rollbacks approach is viable.

CHAPTER 6

Enhanced Agent-based modeling

Never send a human to do a machine's job.

— Agent Smith, The Matrix

Generally speaking, when dealing with simulation, modelers usually have to face the description of complex systems and complex interactions between entities residing into them. Indeed, evaluating or deriving aggregated system dynamics equations could represent an onerous task when working with a computer system. In relation to these issues, the Agent-Based Modeling System (ABMS) paradigm includes system-wide features into the agents' abstractions and interactions and represents a more viable technique to simulate behaviors or emergences of higher-order patterns, such as traffic congestions, network compositions, terroristic organizations, etc. Overall, there are several reasons for which ABMS is considered a powerful approach in the simulation area. First off, the specification of the agents can be decoupled from the environment in which they will operate, resulting in a more straightforward way of developing a model (and possibly re-using agents' specifications). Also, agents' interaction can provide complex behavioral patterns, with the result that even *emergent behaviors* can be exploited and analyzed. The ABMS approach is also suited for adoption in several areas: many works in the literature provide advanced models dealing with neural networks, evolutionary algorithms, etc.([GT00], [MHH07]): this proves

that the agent-based modeling can be even more powerful and realistic than classical model definition approaches.

When designing an ABM system to simulate real-world scenarios, three main issues need to be taken into account:

Topology specification: in order to provide flexibility in terms of movement of the agents, an ABMS description should offer many different and possibly complex topologies. Complex graph-based environments also enable more interesting interactions between agents to be described or to emerge. Moreover, the topology definition should not be limited to real-world environments since virtual scenarios involving many layers of information might be necessary to model synthetic situations. Finally, the topology specification's flexibility should also reside in the capability of agents to dynamically modify the topology itself.

Agent specification: the most common description of an agent is provided through simulation states. Specifically, this involves the set of variables and structures that maintain their interaction with the environment and other agents, along with their own evolution over time. The state of an agent results to be crucial not only because it allows to identify it among the whole system uniquely, but also because it drives the behavior of it throughout the simulation.

Interaction specification: agents need to communicate with each other in order to perform basic actions related to the evolution of the simulation. Their interaction could depend on their current state, the status of the environment they're settled in, and the presence or absence of certain classes of agents in the surrounding areas.

Thanks to its easiness in development and its capability of modeling

complex systems in a more natural way, ABMS is gaining interest at a higher and higher pace. However, when dealing with the actual materialization of such systems on top of a computing platform, many concerns (possibly experienced at the same time) may arise, such as:

- Non-linear and dynamic behaviors in the system of interest, presenting high uncertainty and a conspicuous amount of stochasticity. This is a specific context in which a single run of a simulation model is not enough to extract statistically meaningful results, and only averaging results on a repeated number of runs can lead to better estimations. Although this kind of scenario can take place, in general, in many stochastic models, unfortunately, while a single ABMS run could be feasible, a large number of ABMS simulations might be not.

- Enhanced policies driving the interaction and decision-making of the agents, based, e.g., on cognitive or psychological processes. This kind of requirement, typical of micro-simulations, may entail higher computation demands.

- Enlarged set of explanatory variables in the state of single agents. This situation, again typical of micro-simulations, can lead to an increased amount of data to maintain, becoming unfeasible when the overall number of agents increments.

- Large number of involved agents or incredibly vast environments. For instance, a larger model could disclose emergent behavior linked to the number of interactions between agents. It could be the case that smaller models can't produce such a result because of interactions' magnitude. These cases can either generate unfeasibilities related to insufficient memory capacity of computing nodes or to extended (potentially indefinite) execution times of simulation runs.

Therefore, even if, on the one hand, the ABMS formalism can cope with the handling of complex systems, on the other hand, *what-if* analysis can experience performance penalties preventing researchers from finding the needed insights. Additionally, concerning speculative PDES, modelers may be required to manage an increased amount of implementation details. In fact, although speculative discrete event simulation models turn out to be very well suited for the requirements of an ABM system (e.g., the impulsiveness of events can be embraced in the mode in which agents should interact), some critical operations such as the handling of rollbacks can represent an unviable burden for developers. Considering a high number of domains relying on ABMS and parallel/distributed computing platforms, it is required not to demand the final programmer to tackle implementation-related details, as the ABMS framework should hide this complexity, letting the developer concentrate only on model-specific dynamics.

For these reasons, this chapter describes a semantically-rich API that allows a simple and effective implementation of ABMS models on top of PDES platforms. In particular, it has been devised to address *i)* the peculiarities of massively parallel/distributed architectures *ii)* the aforementioned requirements of ABMS, and *iii)* bridge the gap between the users of ABMS run-time environments and the architectural capabilities of modern computing infrastructures.

6.1 The enriched ABMS APIs

The APIs that will be presented in this section were devised, as mentioned, to transparently provide ABM features on top of PDES platforms, taking into account both programmability and efficiency. A core characteristic of these APIs is represented by the fact that they are stateful by design: the

run-time environment must keep track of a set of data structures to maintain the evolution of the simulation. To provide a comprehensive set of functionalities, the problems mentioned above related to ABM were specifically tackled. Therefore, the APIs were split in three different areas of concern, namely agents' modeling, agents' interaction, and topology definition. The three different subsets of APIs will be discussed separately in the following subsections. Note that the APIs definitions are presented as signatures of a possible C implementation, but by no means this means that they are limited to this specific programming language.

6.1.1 Modeling agents

The fundamental aspect to be introduced before discussing the provided APIs regarding agents definition lies in the way in which agents are handled by the run-time environment. Taking inspiration from [CPQ15] and [PQMSCG16], in our ABM system, each logical process models a single region of the environment, while agents are represented by either data structures or objects, depending on the programming language the APIs are implemented with. This choice was specifically made in order to simplify the implementation of agents' interactions dramatically: an agent is handled via variables which is a concept that even a non-experienced developer can quickly master.

With this premise in mind, the process of uniquely identifying an agent becomes straightforward. Indeed, we introduced the `agent_t` type, which can be considered a system-wide unique identifier (UID). A variable of this type needs to be associated with an agent's data structure, and its globally unique value can be computed relying on the fact that the simulation kernel instances involved in a simulation run are already commonly identified in the system itself (e.g., via an MPI rank in a parallel/distributed scenario).

Hence, a viable value can be computed basing on a per-instance monotonic counter, such as the *Cantor pairing function*

$$\frac{(K+c)\times(K+c+1)}{2}+c \tag{6.1}$$

Here, K is the simulation kernel instance ID, and c is the current value of the monotonic counter. Even if this is only an example, this approach can be suitable as it relies only on integer arithmetic, and can be optimized in multithreaded environment via atomic facilities (e.g., read-modify-write instructions). Since the lifecycle of an agent can be shorter than the duration of the whole simulation, it is necessary to handle the `agent_t` type in an efficient manner. In fact, as will be later discussed, agents can be dynamically added or dismissed, involving possibly costly operations regarding data structures managing them. Therefore, a straightforward suggestion would be to use hash maps to reduce the latency of retrieving metadata representing an agent.

On the other hand, the final user can exploit any kind of arbitrarily complex data structure to store agents' state, provided that they maintain all the explanatory variables allowing the agents to make decisions or change their behavior. Depending on the size of different families of agents the model of interest involves, the developer is allowed to define as many data structures as needed in order to fulfill the requirements of each agent (e.g., in the limit case, each agent might be described by its dedicated data structure type).

The description of the APIs related to agents' management are provided in the following:

`AddAgent(unsigned user_data_size)`: Provides the creation of a new agent in the system. The return value is an `agent_t` variable, identi-

fying the newly created agent (resembling return values of databases' CRUD operations). The parameter `user_data_size` specifies the size of a buffer that the run-time environment is required to dedicate to store data associated with the new agent. Note that, since this API targets speculative environments, the creation of a new agent can be subject to rollbacks. Therefore, the run-time platform needs to bind the buffers associated with the new agent's representation to the event causing this agent's creation and the LP which originally executed this event. A viable implementation of such a rollback handling can be achieved making use of dynamically allocated memory. In fact, if the allocation of an agent is redirected to the memory dedicated to an LP (i.e., a region), then rolling back the state of the LP would entail restoring a consistent situation concerning the creation of an agent as well. Clearly, this will imply that for a limited period, the agent "belongs" to the simulation state of the region within which it is residing. However, this turns out to be a desirable situation since all the agents residing in the same region (i.e., LP) will observe a consistent simulation snapshot while simulating events accessing and/or modifying their structures. Moreover, given that agents are identified by a UID, if the generation rule is similar to the one presented in 6.1, then it is not required to handle the roll back of the UIDs. Indeed, the UID related to the rolled back agent won't be associated with any agent in the restored (i.e., corrected) simulation track.

`RemoveAgent(agent_t agent)`: This is the counterpart of the previous API. If the model logic at some point determines that an agent is no longer needed to be part of the simulation, this function is the entry point to delete it. The function accepts as parameter the `agent_t` which uniquely identifies the target agent. After the completion of

this API, the semantic is that all subsequent API calls referring to the deleted `agent_t` should fail. Again, we denote the fact that even the deletion of an agent is an operation subject to rollbacks. With respect to the buffers keeping the state of the agent, the same consideration made for the `AddAgent()` API can be applied. However, a deeper discussion is required for the correct handling of the identifier of the agent. Regardless on how the agents' identifiers are maintained (e.g., via fast hash-maps), the run-time environment needs to provide the following behavior on deleted agents: *i)* every action involving a deleted agent should fail and *ii)* if a call to `RemoveAgent()` is rolled back, then the UID associated with the deleted agent must become valid again. A possible solution to the second point can be provided by putting in place a sort of cache of deleted agents. Whenever a call to `RemoveAgent()` is issued, a flag determining whether an agent is retracted can be set. The UIDs of the agents retracted during the execution of an event can be maintained into a dedicated buffer kept into the event's structure. In this way, whenever a rollback occurs, all the retracted agents can be safely reintroduced in the system by simply clearing their retraction flag, and as a consequence, their `agent_t` identifier will be safely recycled too. Retracted agents can be permanently deleted upon a GVT round computation: along with the garbage collection operations performed after computing the *committment horizon* (we recall that this is a. standard operation in traditional Time Warp systems), the simulation platform can inspect and safely free the retracted agents' buffers.

`GetAgentData(agent_t agent)`: Given the UID of an agent, this API function returns (the pointer to) the memory buffers dedicated to this agent's state. Obviously, the size of such a memory buffer should be

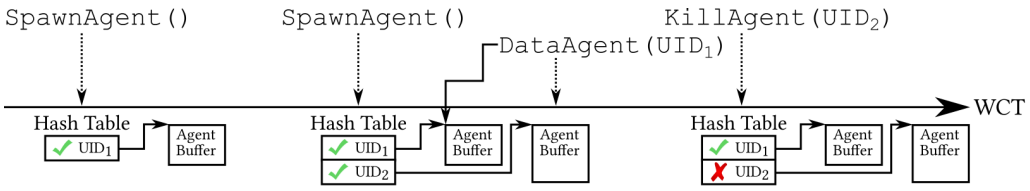


FIGURE 6.1: *Example of agents' management timeline*

at least `user_data_size`, as specified in the `AddAgent()` function. This API results can be useful as the state of an agent can have a structure defined by the simulation model developer. We remark that the memory buffers retrieved with this API need to be managed by the simulation platform from its creation (i.e., upon an `AddAgent()` call) to their dismissal (that is, after a successful GVT computation round).

An example of the lifecycle of an agent is presented in figure 6.1.

6.1.2 Modeling agents' interactions

We split the APIs describing interactions into two subsets: the first one is dedicated to the functions offering the possibility to make agents interact within the same (or neighbouring) portion of the environment (i.e., the same LP), and the second one refers to function dealing with agents' mobility.

Supporting agents' decisions

The APIs governing the interaction of an agent with the current residing region are described in the following:

`CountAgents()`: A foundational operation that can be of interest for an agent is knowing the number of agents which are close to it. Since an LP manages a single region of the overall environment in our design, the number of agents can be easily retrieved by querying the LP state

via operations on fast per-LP hash tables storing the UIDs of currently residing agents. This API can be beneficial to study agents' emergent behaviours.

`IterAgents(agent_t *agent_p)`: This API function supports the operation of scanning through all the agents which are registered into a certain region. Specifically, subsequent calls to this API will fill the `agent_p` parameter with the id of the "next" agent residing in the region (resembling the implementation of an iterator). Note that there is no strict requirement on the ordering according to which the UIDs of the agents are returned while scanning: it is left to the custom implementation of the run-time system to devise how agents are stored in the region. In our case, since the hash table data structure guarantees no ordering, the order in which UIDs are returned coincides with scanning the whole table.

`RegisterNeighborInfo(void *neighbor_data)`: As anticipated, the interactions between agents become essential to study emergent behaviors. When an agent, other than the region in which it resides, becomes aware of the surrounding environment, it can compute even more complex interactions. While, on the one hand, traditional sequential or time-stepped simulators could allow accessing data residing in different portions of the environment immediately, PDES runs impose additional challenges. Indeed, given the speculative nature of the advancement, LPs modelling regions might have reached at any given moment different simulation time instants, and handling such a kind of shift can be cumbersome for the system. As an example, consider an agent A residing in a region associated with LP_k, which is in the process of deciding whether to move to either a region mapped to LP_x or LP_y. If the rationale that A uses to select the destination

region is the crowdedness of the target, then it will need to access updated and consistent data from all of the possibilities. To make this informed decision, A needs first to collect the number of agents registered in LP_x and LP_y, a logic that in order to be implemented in traditional PDES needs to be split into multiple events targeting the different LPs. Therefore, due to the optimistic processing of events, LP_x and/or LP_y might be forced to roll back, which will insert an additional burden to the run-time environment (and this becomes even more problematic if the events are simultaneous). Hence, `RegisterNeighborInfo()` addresses these problems by implementing a sort of publish/subscribe protocol between the involved LPs. Provided that the model developer can define, at simulation startup, the portion of the state of an LP that can be broadcast to other LPs, this function allows to subscribe to it and retrieve a local copy of that data, which needs to be superseded every time a new version of data is installed. In this way, an event that needs to access data from a neighbour can inspect a local copy of its state portion. Regarding updates toward this local copy, two additional points need to be stressed:

1. The run-time system needs to understand whether the accessing event will modify the observed memory area or not. Depending on the actual implementation of the platform, this can result in a costly operation. In our implementation, modification detection is carried out via a fast hash function comparison of memory buffers associated with the accessed data. Different solutions may entail the usage of OS facilities such as `mprotect`, which are inherently more costly.

2. The run-time system needs to handle consistency of the values observed by neighbours. Indeed, updates to the subscribed mem-

ory portions must be transparently sent to the subscribers, and if the actual event triggering state transitions is rolled back, also the updates towards the subscribed memory should be undone. This is a crucial aspect since if the event executed by a subscriber LP reads a retired value, the execution of this event needs to be annihilated too, according to the traditional Time Warp protocol. In our implementation, we cope with this situation with the usage of *control messages*, which are events exchanged at the level of the run-time system, which are included in the receiving LP's event queue but never delivered to simulation models' handlers. The logic associated with these events is precisely that of updating the local copy of remote portions of the simulation state once the timestamp associated with them is reached in the simulation. Thus, annihilating such control messages via anti events cancels updates that have not yet been processed, while relying on the rollback operation allows restoring previous consistent snapshots in case remote state updates are retracted.

`GetRegionData(direction_t direction, unsigned int *region_id, void **data_p)`: After subscribing to some specific region's data, this API allows to retrieve the data from the registered LP. The `data_p` pointer makes the data of this neighbouring LP accessible. In order to select the region of interest, the `direciton_t` parameter can be used to navigate the selected topology. Finally, in the `region_id` pointer, the run-time should store the id of the LP currently handling the portion of the environment targeted by `direction_p`.

Supporting agents' movements

In many classical agent-based models, updates in agents and the environment happen globally (e.g., [Rey87], [Gar70]). This entails that it is simply impossible to produce definitive inferences by observing only a subregion of the model. Clearly, this kind of approach is incompatible by design with speculative PDES data partition-oriented systems, which are the ones in which the Time Warp protocol best fits. To support such a programming model without placing a high burden on the final model designer, the concept of *visit* needs to be introduced. An agent maintains a set of regions to be visited, which can be modified at any time, creating a *visit list* which allows defining the order in which regions belonging to the topology of interest will be reached during the agent's lifetime. Whenever an agent takes a decision, it is free to modify this list arbitrarily. If an agent does not have a large lookahead, then the *visit list* is simply reduced to the next region to visit. Note that the visit list can even be used to build a set of pending logical tasks at run-time, which are considered interactions too. Therefore, any agent has a versatile way to describe how to explore and interact with the surrounding environment, and this can be easily adapted to many real-world problems. From an implementation point of view, the *visit list* can be realized either by relying on linked lists or on resizable arrays. While the first is a viable solution, in our implementation, we opted for the latter, as it can provide significant performance improvements thanks to memory locality (and, moreover, can be easily migrated around in a distributed environment). This being said, the following APIs describe how agents can act with and react to the environment via the usage of *visit lists*:

`EnqueueVisit(agent_t agent, unsigned region, unsigned event_type)`: Inserts at the end of the *visit list* of an agent the region to be reached. Whenever this visit is triggered, the LP identified by

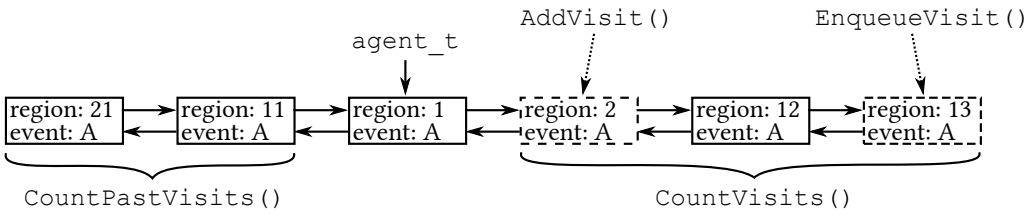


FIGURE 6.2: *Example visit management with respect to current region*

`region` is hit by the specified event of type `event_type`

`CountVisits(const agent_t agent)`: Retrieves the size of the *visit list*. Note that past visits, namely regions that have already been visited, are not included in this return value: the function `CountPastVisits(const agent_t agent)` was specifically devised for this purpose. An example of visit management is presented in figure 6.2.

`GetVisit(const agent_t agent, unsigned *region_p, unsigned *event_type_p, unsigned i)`: While executing the logic associated with the model, it could be of interest to inspect an agent's future visits list to, e.g., understand whether the choice of moving to a specific region is still valid or not. Therefore, this API will inspect the `i`-th entry of `agent`'s future list and store into the `region_p` pointer the id of the LP associated with the visit entry, along with the type of event which will be scheduled when this visit is triggered, into `event_type_p`.

`GetPastVisit(const agent_t agent, unsigned *region_p, unsigned *event_type_p, simtime_t *time_p, unsigned i)`: This is the counterpart of the previous API. Suppose a user is interested in getting information about past visits to a region. In that case, this function will behave as the previously described API returning the information referring to the passed `simtime_t` time. However, with

respect to the run-time system implementation, we emphasize that this API function could generate a non-minimal memory footprint. In fact, if agents move quickly around the available environment, the past visit list may grow indefinitely. Since the semantic of this API allows to retrieve any past visit, also those associated with a timestamp falling before the committed portion of the simulation (namely, before the last GVT round) are eligible. Therefore, the system may not be allowed to prune last visits even when garbage collection operations are put in place. We leave to the implementation of the simulation platform this concern, as different decisions can be honoured in order to cope with specific scenarios: if inspection of all past visits from the beginning of the simulation is required, then no memory associated with visit lists can be reclaimed; on the other hand, if past visits are interesting to some fixed degree than some more sophisticated logic can be undertaken to recycle resources.

`AddVisit(agent_t agent, unsigned region, unsigned event_type, unsigned i)`: Allows to insert a new visit right before the `i`-th entry in the *visit list* of the `agent`.

`SetVisit(const agent_t agent, unsigned region, unsigned event_type, unsigned i)`: Allows to perform modifications on the `i`-th entry of the `agent`'s *visit list*.

`RemoveVisit(agent_t agent, unsigned i)`: Deletes the `i`-th entry of `agent`'s *visit list*, thus the agent won't move in that direction anymore.

`ScheduleNewLeaveEvent(simtime_t time, unsigned int event_type, agent_t agent)`: This API function allows to tell the run-time system the intent of an agent to move towards the next area (according to its visit list) according to the passed `time` simulation

time. Due to the speculative nature of the run-time these functions are to be ported, this function delivers a "guess" for the simulation time at which the next hop of the agent will occur: this is because it is not guaranteed that the simulation trajectory will continue in the expected direction. Indeed, the semantic of this function is that it tries to make the agent move: if, for instance, the agent is removed from the simulation employing a call to `RemoveAgent()`, the firing of the leave event will never take place. Note that subsequent calls to this API can be issued for an agent before the execution of the event. If subsequent calls take place, new invocations will override previous move guesses. The only constraint related to this function is that the specified `time` needs to be in the future with respect to the current simulation time at which the call takes place (traditional Time Warp systems enforce this). Again, in our implementation, we support such a kind of API, making use of control messages, which are placed in the event queue of the LP calling `ScheduleNewLeaveEvent()` so that when its clock reaches the passed `time`, then an event is sent to the targeting LP, namely the one associated with the next region to be visited. This event will piggyback as payload the data related to the current agent. Similar to the other APIs described so far, this function call should be rollbackable or, more precisely, the leave event can be rolled back. For the sake of efficiency, we implemented the rollback support via the usage of the retraction flags mentioned earlier, which are memorizing whether an agent should be kept or removed into a specific region and allowing to free its related memory upon a successful GVT round. However, an additional point needs to be stressed regarding the rollback operations related to this API. Indeed, since we allow the possibility to issue multiple times calls to

API function	Description
`AddAgent`	Create a new agent
`RemoveAgent`	Delete an agent
`GetAgentData`	Retrieve an agent's information
`CountAgents`	Retrieve the number of agents in the neighborhood
`IterAgents`	Return an iterator on the agents in the current region
`AddVisit`	Add entry at specific location in the visit list
`RemoveVisit`	Delete entry at specific location in the visit list
`SetVisit`	Modify a specific entry of the visit list
`RegisterNeighborInfo`	Subscribe to a region information
`GetRegionData`	Retrieve information of a specific region
`EnqueueVisit`	Add a region to an agent's visit list
`CountVisits`	Return the number of entries in the pending visit list of an agent
`GetVisit`	Get an entry from an agent's visit list
`ScheduleNewLeaveEvent`	Initiate the movement of an agent towards the next entry in its visit list

TABLE 6.1: *A summary of the agents' interactions APIs*

`ScheduleNewLeaveEvent()`, the same UID entry in the hash table keeping the retraction flags should be associated with various incarnations of the same agent. To provide so, this data structure can be expanded with a stack: agents are placed in descending timestamp order into the stack, allowing the simulation events to find on top of the stack the newest incarnation of the agent. In case of a rollback occurrence, the agent's incarnations that have been rolled back can be popped and freed. Therefore, the top of the stack will point to the last consistent snapshot of the agent representation.

Table 6.1 summarizes all the APIs dedicated to agents' interactions.

6.1.3 Modeling topologies

As previously hinted, a comprehensive topology description must provide, along with extreme simplicity, two main features:

a) High degree of expressiveness, as ABMS paradigms should be exhaustive in abstracting real-world and synthetic scenarios in all their means.

b) The capability of arbitrarily and dynamically change their structure. This can be a fundamental feature as the physical environment can be modified either by agents or by changes related to the evolution of agents, for example, in disaster recovery scenarios. However, supporting this possibility in a speculative run-time environment is a challenging task.

Regarding the second point, we rely on an initial topology specification to be issued at simulation startup. This means that model developers are free in the generation complex environment, for example, in terms of the shape of regions, number of corners, number of neighbourhoods and so on. In the following enumeration, we list the fundamental topologies that should be supported by the APIs we envisage:

- **Square**: A classical grid in which each LP models a square cell. Depending on the specification, either four or eight neighbours can be reached from each cell (except, obviously, from the boundaries). The overall environment can therefore be either a square or a rectangle.

- **Hexagon**: Similar in spirit to the previous one, with the main difference that six neighbours can be reached from each cell (not placed on the borders).

- **Ring**: Every LP is mapped to a region that can only be reached by the adjacent one, resembling a mono-directional ring. There is only one movement direction.

- **Bid-Ring**: This is an extension of the Ring topology in which agents can move back and forth from a region, having two possibilities in their movement directions.

- **Torus**: This topology is a further extension of the Bid-Ring one. In particular, this is a 3D ring in which agents can move horizontally, vertically or a combination of them, for a total of four or eight possible directions.

- **Star**: Here, a central LP is connected to all other LPs. If an agent desires to move from one cell to another, it must pass through the central LP.

- **Fully Connected Mesh**: This is an expansion of the Star topology in which every LP is connected to any other LP. Even if this can be considered a complete environment in terms of possible movements, at the same time, it introduces the highest burden on the run-time environment due to high costs in communications between LPs.

- **Graph**: Resembling the well-known data structure, this is a generic weighted and directed graph. Each region is connected to a portion of all the LPs, being each connection (i.e., an edge) characterized by either a weight or a probability. In this case, the initial configuration of the topology should define the set of reachable nodes and the associated weight or probability.

Figure 6.3 presents a graphical representation of the topologies mentioned above.

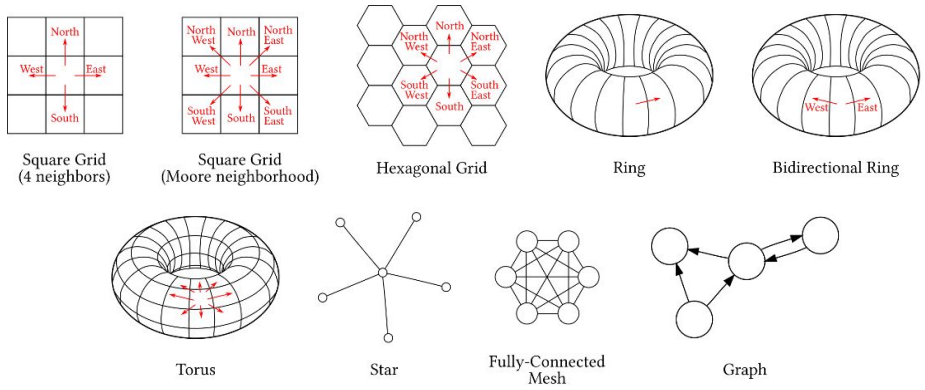


FIGURE 6.3: *Graphical representation of fundamental topologies.*

Note that the last described topology is the most well suited for changes that the model can issue at run-time. In particular, a graph topology could have forbidden connections (for example, obstacles in rescue scenarios) represented by an edge presenting an infinite weight or a zero probability. In this way, by relying on the APIs of the topology, modellers can modify these weights or probabilities at run-time to alter the connections across environmental regions. Changes to the topology, however, require to be rollbackable. Indeed, it may be the case that events scheduled at specific LPs observe a no-longer consistent topology after an inconsistent situation is restored. Hence, the run-time environment must associate the event reshaping the topology with the topology's incarnation at that specific simulation virtual time. In his way, changes can be undone via a procedure similar to the one explained for the agents' interactions APIs. The specifications of the APIs managing the topologies manipulations are presented in the following:

`NeighborsCount(unsigned int region_id)`: Given the UID of a region (LP), returns the number of cells that can be reached from it. This can be especially useful on boundary regions, which could present limitations in terms of agents' movements. In other words, this API

allows retrieving the reachability degree of a region.

`RegionCount()`: Returns the total number of active cells. This API can be useful in situations in which agents are required to respond to observation and sense of the surrounding environment.

`SetValueTopology(unsigned int from, unsigned int to, double value)`: This API is specifically designed for graph-based topologies. In particular it allows to update the `value` associated with the edge connecting `from` and `to` regions. Therefore, this is the entry point for dynamic topology modifications and, as hinted, could be rolled back at any time. Our implementation relies on control messages, using a solution similar to the one presented in the previous section.

`GetReceiver(unsigned int from, direction_t direction)`: Returns the id of the neighbour which would be reached by moving in the specified `direction`, being `from` the source region. This is the API that allows model developers to code movements of agents. Movements, of course, are governed by the topology itself and from the possibilities exposed by the API definition. For instance, a region can lead to up to eight neighbours in a square grid environment, as in the Moore neighbourhood topology.

`FindReceiver()`: Given the stochastic nature of some topologies, we provide this API which picks a random neighbour. In a grid-based environment, the probability distribution to be considered should be uniform. In contrast, in graph-based topologies, the distribution should consider the probabilities (or weights) associated with edges.

`FindReceiverToward(unsigned int to)`: Allows to return the next

API function	Description
`NeighborsCount`	Return the number of currently reachable regions
`RegionsCount`	Return the total number of regions
`SetValueTopology`	Update the value associated with an edge
`GetValueTopology`	Return the value associated with an edge
`FindReceiver`	Pick a random neighbor region
`FindReceiverToward`	Return the next region to visit in the path to the destination
`ComputeMinTour`	Return the shortest path from source to destination as an array of LPs to visit

TABLE 6.2: *A summary of the topology API*

region towards which a agent should move in order to reach the specified `to` destination. This API should account for the minimum-cost path, being this cost either the number of moves or the sum of the weights of the encountered edges. In our implementation, we relied on Dijkstra's pathfinding algorithm, with a small enhancement related to the caching of the computed *Minimum Spanning Tree*. Caching takes place in a per-LP data structure, which is flushed upon a change in the topology.

`ComputeMinTour(unsigned int source, unsigned int dest, unsigned int result[RegionsCount()])`: Returns a collection (or an array) maintaining a list of LPs to visit in order to reach `dest` from `source`. This is an enhancement of the previous API and allows to prepare a tentative schedule for the actions of an agent in the imminent future, a mechanism similar to the *visit list* described in the previous section.

Table 6.2 depicts a summary of the exposed topology APIs.

6.2 A performance assessment

In order to provide a complete evaluation of the viability of the proposed APIs, we considered a total of six models to be used for testing the behaviour of our implementation: three of them are synthetic, while the other half mimic real-world applications. The models of interest are the following:

- *Stupid Model*: A number of insects (also called *bugs*) are placed in a toroidal topology, and they are allowed to move freely inside it. Each region of this area produces food with a particular production rate, and each bug (i.e., agent) that enters a region eats the food inside it, and grows in size accordingly. A bug can only move to a non-occupied cell, and it selects as destination region the cell with the highest food amount. Moreover, with a certain probability, a bug can either die or reproduce. If it reproduces, then five new agents are spawned in the surrounding cells (provided they're empty; otherwise, fewer amounts are created). The termination condition is triggered when a bug reaches a specific maximum size. More information can be found in [RLG05].

- *Segregation*: This model recreates a social situation to show how segregation among people happens, despite the agents (i.e., people) are not characterized by racism inclinations. Therefore, an agent is identified by some information such as race, ethnicity, economic status, etc. Multiple populations (that is, possibly heterogeneous groups of agents) are placed at random positions within the environment at simulation startup. Depending on the percentage of similar agents surrounding him, an agent can be satisfied with his current position or not. Therefore, agents are free to move to the location they prefer within topology's boundaries. For further details, we refer to [Sch78].

- *Sugarscape*: This is a model similar in spirit to the *Stupid Model*. In a grid environment, every region contains a different amount of sugar. Agents move around the grid targeting the nearest cell with the highest amount of sugar in it. Whenever an agent eats, the sugar is metabolized, and they produce pollution. As in *Stupid Model*, an agent can either reproduce or die, according to a determined probability. Moreover, agents can both trade or borrow sugar from companion agents, and they can either transmit diseases or generate immunity to them. The work in [JME97] describes the model in more details.

- *Terrain-Covering Ant Robots (TCAR)*: This model tries to recreate a (natural) disaster situation in which rescue robots are employed to map the area and provide first aid ([KL01]). The agents' first action when an accident occurs is to explore the whole region they're part of. The terrain is modelled as an undirected graph, which is helpful to account for obstacles generated by the disaster itself, and ant robots are allowed to move in any direction. While exploring, each ant releases pheromones, which is a sign of the fact that it passed through a particular region. This is implemented via a node counting algorithm, which increments the number of visits for every cell whenever an agent moves into it. Therefore, each ant moves to the next cell by selecting the region encountering the smallest visit counter value.

- *Robot Explore*: A set of agents are required to explore a topology. This is the model which was explained in section 4.2.1.

- *Tuberculosis*: Models the spread of tuberculosis infections. As this was already discussed in section 5.3.2, we will not provide here further details.

6.2.1 Experimental analysis

As far as performance data is performed, we carried on experiments on a subset of the models above. In particular, we report data for the *Stupid Model* and *Segregation* for the synthetic cases and for *Tuberculosis* and *TCAR* for the real-world scenarios. The test-bed platform we selected for the evaluation is made by a cluster of three medium-size heterogeneous servers interconnected via 1GB dedicated ethernet links. All of the nodes are running Linux 4.9.0 on top of an Opteron^{TM} processor: two of the three nodes are, however, equipped with 32 cores and 64 GB of RAM, while the remaining one mounts 48 cores and 128 GB of RAM. Also, OpenMPI 3.1.3 has been selected as the communication framework between nodes. We also varied the number of threads from 3 to 112, and that the MPI assignment policy has been set so as to ensure that threads were distributed on the cluster in a round-robin fashion: this is a worst-case scenario for the selected environment since we always incur in communication cost, especially when the number of threads is low, even though a single machine could support communication by relying on shared memory.

Table 6.3 shows the configurations of the models that we selected to carry on experiments. Moreover, the performance evaluation focuses on the speedup of the parallel/distributed runs over a sequential simulation. Note that sequential runs are extremely optimized, as they rely on a fast O(1)

Model	**# LPs**	**Involved Agents**
Stupid Model	4,096	1 [1]
Segregation	10,000	5,000
TCAR	3600	48
Tuberculosis	1024	400,000

TABLE 6.3: *Configurations of the models for ABMS APIs benchmarking*

scheduler based on calendar queues (as described in [Bro88]). Figure 6.4 reports the performance result for the real-world simulation models, namely TBC and TCAR. For the latter one, from the data, we can understand that performance experiences a drop when exploiting a number of concurrent threads overcoming a specific threshold (namely, around 40). This is an expected result as, considering the configuration shown in table 6.3, there is a high imbalance between the overall amount of agents and the number of regions composing the topology. Therefore, given the distributed nature of the runs, the probability that LPs observe skews in simulation time is high and, in turn, affects the final efficiency of the simulation with approximately a 15% drop. Note, however, that this phenomenon is not dependant on the API used to implement the model; rather, it would have appeared even without using the proposed structure. Therefore, we can note that the proposed API implementation is resilient to this unfavourable scenario, up to a significant quantity of distributed threads.

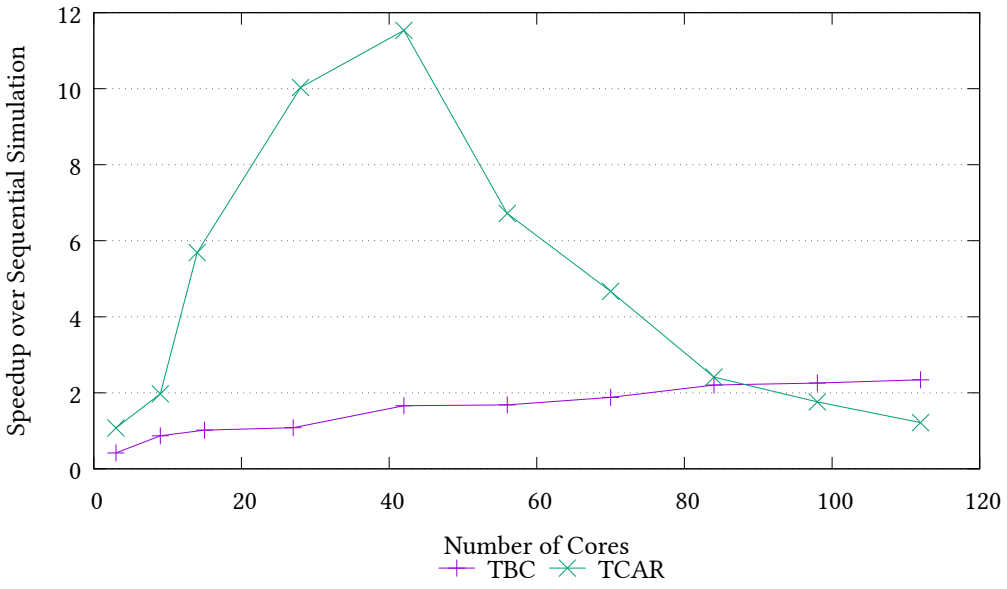


FIGURE 6.4: *ABMS API Results with the real-world models*

For the TBC model, we stress the fact that two factors increase the roll-

back probability: *i)* this is the model with the highest degree of parallelism because every thread controls a reduced amount of LPs (the minimum case is 9 LP per thread) and *ii)* the overall number of agents in the simulation, compared to the size of the environment, is quite large.

Despite the escalation of rollbacks, the proposed API is, in this case, able to provide a speedup, although minimal. We recall that this is a case in which resorting to our proposal allows making hard problems feasible due to the possibility to, e.g., overcome the memory wall thanks to the distribution of the load among all possible nodes in the cluster. For instance, the sequential run of this model has shown a memory footprint of 16 GB, which couldn't be bearable when using off-the-shelf commodity hardware.

As far as the synthetic models are concerned, we show in figure 6.5 the performance data regarding *Stupid Model* and *Segregation*. Interestingly, both models deliver a linear speedup, even though event granularity is relatively small while the amount of events exchanged between LP is non-negligible. Indeed, due to the `RegisterNeighbourInfo()` API, control messages to neighbouring LPs are transparently sent upon any update of a region, and the number of agents receiving those updates is quite high (especially in the *Segregation* case, according to table 6.3). The reason behind the experienced linear speedup is that the number of explanatory variables used to describe an agent is reduced, thus the simulation states are minimal. Therefore, the scale of the model can be significantly increased (up to 5 thousand in the *Segregation* case), with the result of better handling events when the number of cores/threads increases. These results suggest that if these models, which are very simple by their definition, are used as a skeleton to build a more complicated logic, then the benefits gathered from distributed PDES exploiting an increased computing power can be non-minimal. As a last note, in figure 6.6, we show the performance eval-

uation of runs performed on the largest node of the cluster regarding the real-world models. Considering that we do not pay communication costs in this case, the trends of the curves are perfectly in line with the ones of figure 6.4, involving a distributed scenario. Therefore, the proposed API and its reference implementation are also resilient to network delays.

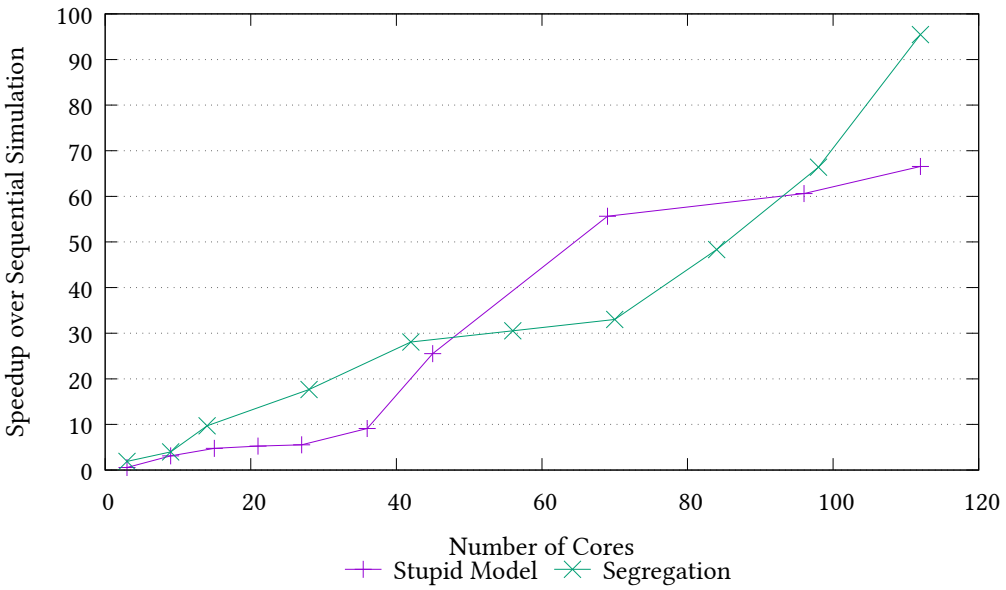


FIGURE 6.5: *ABMS API Results with the synthetic models*

Effects on programmability

Since the final scope of the ABM layer (and, overall, of this thesis) was to provide an efficient and easy to use set of APIs, a final note about the effects on programmability that this solution provides is to be discussed. We remind that assessing those effects is not easy, especially in contexts involving multiple domains models. For this reason, we compared the classical implementations with ones relying on the proposed API. To give an idea of how our ABMS solution can be viable to reduce the burden of model development, for each of the models described above, we compared the Lines Of Code (LOCs) of the original implementation with the LOCs of the new

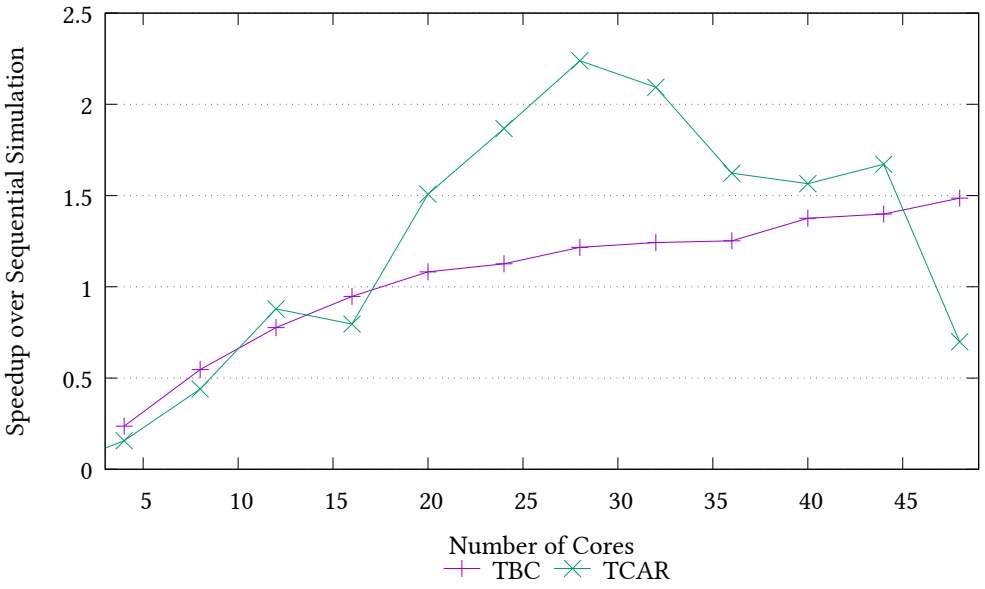


FIGURE 6.6: *ABMS API Results with a single node, in the real-world models scenario*

implementation exploiting the presented APIs. From table 6.4 we can understand that, except for the *Segregation* case, the line count is constantly smaller in the case of the new implementation, even for models which are not mimicking real-world scenarios (that is, very simple models). This is an indication of the fact that the API is somehow effective at capturing the requirements of ABMS, reducing the cost on model developers, with the final result that they can concentrate on model dynamics and, at the same time, benefit from the performance speedup offered by distributed PDES.

Model	Original Implementation	Using our API
Stupid Model	189	138
Segregation	83	110
Sugarscape	1072	152
TCAR	228 [2]	103
Robot Explore	500	332
Tuberculosis	1,115	654

TABLE 6.4: *LOCs for the different ABMS model Implementations*

CHAPTER 7

Conclusions

In this thesis, we faced the issues related to the ease of use and enhanced exploitation of CC resources, particularly in the scientific area of simulation. The interest in literature and the challenges the entire world needed to tackle in the last two years proved the crucial role simulation can play in forecasting real-world scenarios. Therefore, easier and wiser employment of the nowadays parallel and distributed (cloud) resources turns out to be essential to reach the expected performance targets. After this dissertation, experts from many scientific areas can leverage *i)* a transparent mechanism that makes distributed memory accesses, taking place along speculative PDES trajectories, occurring as "invisible" to the final developer, with the result that the burden of exploiting distributed clusters is reduced; *ii)* a semantically rich API to let the speculative PDES features be effectively used to execute Agent-Based Modeling scenarios, with the primary purpose of providing an easy-to-use interface to modelers and let them focus on model-specific peculiarities rather than PDES intrinsics; *iii)* a methodology that allows modelers to have a higher degree of freedom in determining portions of the simulation state to be saved/restored whenever a causality inconsistency occurs, in order to properly tune speculative simulation runs performance basing on the evolution of the data at run-time.

In Chapter 4 we have presented a synchronization protocol and a ref-

erence distributed middleware implementation to support the deployment of PDES simulation models implemented for shared memory on clusters of (Cloud) resources. The middleware transparently intercepts memory accesses to the state of different simulation objects by models' event handlers. Suppose the simulation objects are running on a remote node. In that case, the middleware enforces a synchronization protocol that transparently transfers causally-consistent memory pages, therefore implementing a form of transparent speculative distributed shared memory. A decision model determines, at run-time, the best-suited amount of pages that should be prefetched upon the first remote memory access to reduce the likelihood that additional synchronization is required to enhance the overall performance. Our experimental evaluation has shown that this strategy is effective under differentiated workloads and memory-access patterns. Therefore, this proposal can be a viable solution to support the transparent deployment of PDES models on clusters of (Cloud) resources, without a significant burden on the model developer to explicitly code a large number of memory-based interactions across the simulation models.

In Chapter 6, we have introduced an API specification for ABMS in Time Warp run-time environments. This API has been shown to be effective at implementing very different agent-based models compactly and expressively. Furthermore, an implementation of our API has also shown that it is possible to obtain non-minimal speedup also in straightforward (toy) models. Therefore, we consider this an important step ahead at disclosing the power of PDES to domain experts, which should not be exposed to the complexity of speculative synchronization.

Finally, the presented approximated rollbacks technique, described in Chapter 5, allows saving/restoring only a subset of the state of a simulation object, relying on a user-defined state-reconstruction function to possibly

guess the non-restored (missing) portions of the state. Our experimental assessment has shown that this approach can provide non-negligible performance improvements in local and distributed simulation environments. We have also assessed the impact of the approximated rollback technique on the statistical goodness of the outcome of the simulation, in particular for the case of an epidemic phenomenon of the tuberculosis disease. These results further show the viability of our proposal.

All of the proposed innovations were implemented on top of the ROOT-Sim parallel/distributed general-purpose simulation engine and are entirely open-source to be shared both for improvements and utilization by experts in the field. Some of the produced papers were also awarded the reproducibility ACM badges[1], proving that the introduced results are accurate and permanently available, replicable, reusable, and available for repurposing in any further researches.

[1]https://www.acm.org/publications/policies/artifact-review-badging

Bibliography

[AST07] T. Andres F. Campolongo J. Cariboni D. Gatelli M. Saisana A. Saltelli, M. Ratto and S. Tarantola. *Introduction to Sensitivity Analysis*, chapter 1, pages 1–51. John Wiley & Sons, Ltd, 2007.

[ATCL09] Heiko Aydt, Stephen Turner, Wentong Cai, and Malcolm Y. H. Low. Research issues in symbiotic simulation. pages 1213–1222, 12 2009.

[ATLO17] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory M.P. O'Hare. Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Computer Science Review*, 24:13–33, may 2017.

[BC18] Andrei Borshchev and Nikolay Churkov. Anylogic cloud: cloud-based simulation analytics. In *Proceedings of the 2018 Winter Simulation Conference*, WSC, pages 4245–4245. IEEE Press, 2018.

[BCJL13] Peter D. Barnes, Christopher D. Carothers, David R. Jefferson, and Justin M. LaPre. Warp speed: Executing time warp on 1,966,080 cores. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, page 327–336, New York, NY, USA, 2013. Association for Computing Machinery.

[Bel90] Steven Bellenot. Global virtual time algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 122–127, January 1990.

[BJCH09] David W. Bauer Jr., Christopher D. Carothers, and Akintayo Holder. Scalable time warp on blue gene supercomputers. In *2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 35–44, 2009.

[Blo15] Jakob Blomer. A survey on distributed file system technology. *Journal of Physics: Conference Series*, 608:012039, 05 2015.

[Bro88] R. Brown. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Commun. ACM*, 31(10):1220–1227, October 1988.

[Bru95] David Bruce. The treatment of state in optimistic systems. In *Proceedings of the 9th workshop on Parallel and Distributed Simulation*, volume 25 of *PADS*, pages 40–49, New York, NY, USA, 1995. IEEE Computer Society.

[Bry77] Randal Bryant. Simulation of packet communications architecture computer systems. mit-lcs-tr-188. 01 1977.

[BS93] H. Bauer and C. Sporrer. Reducing rollback overhead in time-warp based distributed simulation with optimized incremental state saving. In *[1993] Proceedings 26th Annual Simulation Symposium*, pages 12–20, 1993.

[CDM+12] Gennaro Cordasco, Rosario De Chiara, Ada Mancuso, Dario Mazzeo, Vittorio Scarano, and Carmine Spagnuolo. A framework for distributing agent-based simulations. In Michael Alexander, Pasqua D'Ambra, Adam Belloum, George Bosilca, Mario Cannataro, Marco Danelutto, Beniamino Martino, Michael Gerndt, Emmanuel Jeannot, Raymond Namyst, Jean Roman, Stephen L. Scott, Jesper Larsson Traff, Geoffroy Vallée, and Josef Weidendorfer, editors, *Proceedings of Euro-Par 2011: Parallel Processing Workshops*, Lecture Notes in Computer Science, pages 460–470. Springer Berlin Heidelberg, 2012.

[CLY+11] Li-li Chen, Ya-shuai Lu, Yi-ping Yao, Shao-liang Peng, and Lingda Wu. A well-balanced time warp system on multi-core environments. In *2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, pages 1–9, 2011.

[CM79] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, 1979.

[CN13] Nicholson Collier and Michael North. Parallel agent-based simulation with Repast for High Performance Computing. *SIMULATION*, 89(10):1215–1235, oct 2013.

[CP99] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. . Efficient optimistic parallel simulations using reverse computation. 9(3):224–253, July 1999.

[CP10] Christopher D. Carothers and Kalyan S. Perumalla. On deciding between conservative and optimistic approaches on massively parallel platforms. In *Winter Simulation Conference*, WSC, pages 678–687. IEEE, 2010.

[CPJ+18] Biagio Cosenza, Nikita Popov, Ben Juurlink, Paul Richmond, Mozhgan Kabiri Chimeh, Carmine Spagnuolo, Gennaro Cor-

dasco, and Vittorio Scarano. OpenABL: A Domain-Specific Language for Parallel and Distributed Agent-Based Simulations. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2018.

[CPQ15] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. RAMSES: Reversibility-based agent modeling and simulation environment with speculation support. In Sasha Hunold, Alexandru Costan, Domingo Ginenéz, Alexandru Iosup, Laura Ricci, María Engracia Gómez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L. Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander, editors, *Proceedings of Euro-Par 2015: Parallel Processing Workshops*, PADABS, pages 466–478. LNCS, Springer-Verlag, 2015.

[CPQ17] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. Transparently mixing undo logs and software reversibility for state recovery in optimistic pdes. *ACM Trans. Model. Comput. Simul.*, 27(2), may 2017.

[CS89] K. M. Chandy and R. Sherman. Space-time and simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 53–57. SCS, 1989.

[CT92] Wentong Cai and Stephen J Turner. *An Algorithm for Distributed Discrete-event Simulation–: The" carrier Null Message" Approach*. University of Exeter. Department of Computer Science, 1992.

[DM14] Gabriele D'Angelo and Moreno Marzolla. New trends in parallel and distributed simulation: From many-cores to cloud computing. *Simulation Modelling Practice and Theory*, 49:320–335, 2014.

[DSQC+15] Pierangelo Di Sanzo, Francesco Quaglia, Bruno Ciciani, Alessandro Pellegrini, Diego Didona, Paolo Romano, Roberto Palmieri, and Sebastiano Peluso. A flexible framework for accurate simulation of cloud in-memory data stores. *Simulation Modelling Practice and Theory*, 58(2):219–238, July 2015.

[FD97] A Fabbri and L Donatiello. SQTW: a mechanism for state-dependent parallel simulation. Description and experimental study. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 82–89, 1997.

[Fie06] Frank Fiedrich. An hla-based multiagent system for optimized resource allocation after strong earthquakes. pages 486–492, 2006.

[FKK+06] D. Fox, J. Ko, K. Konolige, B. Limketkai, D. Schulz, and B. Stewart. Distributed multirobot exploration and mapping. *Proceedings of the IEEE*, 94(7):1325–1339, 2006.

[Fuj90] Richard M Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[Fuj99] R.M. Fujimoto. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proceedings Thirteenth Workshop on Parallel and Distributed Simulation. PADS 99. (Cat. No.PR00155)*, pages 46–53, 1999.

[G91] Kaushik Ghosh and Richard M . Parallel discrete event simulation using space-time memory. In *Proceedings of the International Conference on Parallel Processing*, pages 201–208. CRC Press, 1991.

[Gar70] M Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, 1970.

[GFS93] Kaushik Ghosh, Richard M. Fujimoto, and Karsten Schwan. Time warp simulation in time constrained systems. *SIGSIM Simul. Dig.*, 23(1):163–166, July 1993.

[GIOY16] B. Kaan Gorur, Kayhan Imre, Halit Oguztuzun, and Levent Yilmaz. Repast hpc with optimistic time management. In *Proceedings of the 24th High Performance Computing Symposium*, HPC '16, pages 4:1–4:9, San Diego, CA, USA, 2016. Society for Computer Simulation International.

[GT00] Nigel Gilbert and Pietro Terna. How to build and use agent-based models in social science. *Mind & Society*, 1(1):57–72, mar 2000.

[HCS06] Mike Holcombe, Simon Coakley, and Rod Smallwood. A general framework for agent-based modelling of complex systems. In *Proceedings of the 2006 European conference on complex systems*. European Complex Systems Society Paris, France, 2006.

[HNP+13] Danny Hendler, Alex Naiman, Sebastiano Peluso, Francesco Quaglia, Paolo Romano, and Adi Suissa. Exploiting locality in lease-based replicated transactional memory via task migration. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 121–133, 2013.

[HU09] Jan Himmelspach and Adelinde Uhrmacher. The JAMES II framework for modeling and simulation. In *HiBi09 - 2009 International Workshop on High Performance Computational Systems Biology*, 2009.

[Iba13] Hitoshi Iba. *Agent-Based Modeling and Simulation with Swarm.* Chapman and Hall/CRC, jun 2013.

[IPGCMW18] Alonso Inostrosa-Psijas, Veronica Gil-Costa, Mauricio Marin, and Gabriel Wainer. Semi-asynchronous Approximate Parallel DEVS Simulation of Web Search Engines. *Concurrency and Computation: Practice and Experience*, 30(7), apr 2018.

[IS93] Golden G. Richard III and Mukesh Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *12th Symposium on Reliable Distributed Systems, SRDS 1993, Princeton, New Jersey, USA, October 6-8, 1993, Proceedings*, pages 58–67, 1993.

[Iva20] Dmitry Ivanov. Predicting the impacts of epidemic outbreaks on global supply chains: A simulation-based analysis on the coronavirus outbreak (covid-19/sars-cov-2) case. *Transportation Research Part E: Logistics and Transportation Review*, 136:101922, 2020.

[Jef85] David R. Jefferson. Virtual time. 7(3):404–425, July 1985.

[JME97] Robert L. Axtell Joshua M. Epstein. *Growing Artificial Societies: Social Science from the Bottom Up.* MIT Press, 1997.

[KL01] Sven Koenig and Yaxin 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.

[LCRP+05] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. MASON: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.

[LL90] Yi-Bing Lin and Edward D. Lazowska. Determining the global virtual time in a distributed simulation. In Pen-Chung Yew, editor, *Proceedings of the 1990 International Conference on Parallel Processing, Urbana-Champaign, IL, USA, August 1990. Volume 3: Algorithms and Applications*, pages 201–209. Pennsylvania State University Press, 1990.

[LLT07] Michael Lees, Brian Logan, and Georgios Theodoropoulos. Distributed simulation of agent-based systems with HLA. *ACM Transactions on Modeling and Computer Simulation*, 2007.

[LLT08] Michael Lees, Brian S. Logan, and Georgios Theodoropoulos. Using access patterns to analyze the performance of optimistic synchronization algorithms in simulations of MAS. *Simulation*, 84(10-11):481–492, 2008.

[M.90] Richard M. Performance of time warp under synthetic workloads. *Proceedings of the SCS Multiconference on Distributed Simulations, 1990*, 22(1):23–28, 1990.

[MGSF07] Jens Müller, Sergei Gorlatch, Tobias Schröter, and Stefan Fischer. Scaling multiplayer online games using proxy-server replication: A case study of quake 2. HPDC '07, pages 219–220, New York, NY, USA, 2007. ACM.

[MH95] Horst Mehl and Stefan Hammes. How to integrate shared variables in distributed simulation. *SIGSIM Simulation Digest*, 25(2):14–41, 1995.

[MHH07] Shingo Mabu, Kotaro Hirasawa, and Jinglu Hu. A Graph-based Evolutionary Algorithm: Genetic Network Programming (GNP) and its Extension sing Reinforcement Learning. *Evolutionary computation*, 15(3):369–98, sep 2007.

[MNPQ16] Nazzareno Marziale, Francesco Nobilia, Alessandro Pellegrini, and Francesco Quaglia. Granular Time Warp objects. In *Proceedings of the 2016 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation*, PADS, pages 57–68, New York, New York, USA, 2016. ACM Press.

[Moo06] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, Sep. 2006.

[MSGNCG+15] Cristina Montañola-Sales, Joan Francesc Gilabert-Navarro, Josep Casanovas-Garcia, Clara Prats, Daniel López, Joaquim Valls, Pere Joan Cardona, and Cristina Vilaplana. Modeling tuberculosis in barcelona. a solution to speed-up agent-based simulations. In *2015 Winter Simulation Conference (WSC)*, pages 1295–1306, 2015.

[PAH+94] Richard G. Palmer, W. Brian Arthur, John H. Holland, Blake LeBaron, and Paul Tayler. Artificial economic life: a simple model of a stockmarket. *Physica D: Nonlinear Phenomena*, 75(1):264–274, 1994.

[Pel13] 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. Shortlisted for the Outstanding Poster Paper Award.

[PLM94] Bruno R Preiss, Wayne M Loucks, and Ian D. Macintyre. Effects of the Checkpoint Interval on Time and Space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, jul 1994.

[PPQV16] Alessandro Pellegrini, Sebastiano Peluso, Francesco Quaglia, and Roberto Vitali. Transparent speculative parallelization of discrete event simulation applications using global variables. *International Journal of Parallel Programming*, 44(6):1200–1247, December 2016.

[PQ13] Alessandro Pellegrini and Francesco Quaglia. The ROme OpTimistic Simulator: A tutorial. In *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*, PADABS, pages 501–512. LNCS, Springer-Verlag, August 2013.

[PQ17] Alessandro Pellegrini and Francesco Quaglia. A fine-grain time-sharing time warp system. *ACM Trans. Model. Comput. Simul.*, 27(2), May 2017.

[PQ19] Alessandro Pellegrini and Francesco Quaglia. Cross-state events: a new approach to parallel discrete event simulation and its speculative runtime support. *Journal of Parallel and Distributed Computing*, 132:48–68, October 2019.

[PQMSCG16] Alessandro Pellegrini, Francesco Quaglia, Cristina Montanola-Sales, and Josep Casanovas-Garca. Programming agent-based demographic models with cross-state and message-exchange dependencies: A study with speculative PDES and automatic load-sharing. In *Proceedings of the 2016 Winter Simulation Conference*, WSC, pages 955–966. IEEE, dec 2016.

[PVQ09] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Di-dymelor: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. pages 45–53, 06 2009.

[PVQ15] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1560–1569, 2015.

[PW93a] Avinash C. Palaniswamy and Philip A. Wilsey. Adaptive checkpoint intervals in an optimistically synchronised parallel digital system simulator. In *VLSI*, 1993.

[PW93b] Avinash C. Palaniswamy and Philip A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. *SIGSIM Simul. Dig.*, 23(1):127–134, July 1993.

[QB04] F. Quaglia and R. Beraldi. Space uncertain simulation events: some concepts and an application to optimistic synchronization. In *18th Workshop on Parallel and Distributed Simulation, 2004. PADS 2004.*, pages 181–188, 2004.

[Qua98] Francesco Quaglia. Event history based sparse state saving in time warp. *SIGSIM Simul. Dig.*, 28(1):72–79, jul 1998.

[Qua99] Francesco Quaglia. Combining periodic and probabilistic checkpointing in optimistic simulation. In *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation*, PADS '99, page 109–116. IEEE Computer Society, 1999.

[Qua01] Francesco Quaglia. A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):346–362, apr 2001.

[RAT93] Hassan Rajaei, Rassul Ayani, and Lars-Erik Thorelli. The local time warp approach to parallel simulation. PADS '93, page 119–126, New York, NY, USA, 1993. Association for Computing Machinery.

[Rey87] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21(4):25–34, aug 1987.

[RLAM96] Robert Rönngren, Michael Liljenstam, Rassul Ayani, and Johan Montagnat. Transparent incremental state saving in time warp parallel discrete event simulation. PADS '96, page 70–77, USA, 1996. IEEE Computer Society.

[RLG05] Steve F Railsback, Steve Lytinen, and Volker Grimm. StupidModel and Extensions: A Template and Teaching Tool for Agent-based Modeling. Technical report, Swarm Development Group, 2005.

[RR08] Paul Richmond and Daniela Romano. Agent based gpu, a real-time 3d simulation and interactive visualisation framework for massive agent based modelling on the gpu. In *Proceedings International Workshop on Supervisualisation*, 2008.

[RRCC10] Paolo Romano, Luís E. T. Rodrigues, Nuno Carvalho, and João P. Cachopo. Cloud-tm: harnessing the cloud with distributed transactional memories. *Operating Systems Review*, 44(2):1–6, 2010.

[RTRW98] D.M. Rao, N.V. Thondugulam, R Radhakrishnan, and P.A. Wilsey. Unsynchronized parallel discrete event simulation. In D. J. Medeiros, Edward F. Watson, John S Carson, and Mani S. Manivannan, editors, *Proceedings of the 1998 Winter Simulation Conference*, volume 2 of *WSC*, pages 1563–1570. IEEE, 1998.

[Sch78] Thomas C Schelling. Sorting and Mixing. In *Micromotives and Mmacrobehavior*, chapter 4, page 270. W W Norton & Co, 1978.

[SE98] H.M. Soliman and A.S. Elmaghraby. An Analytical Model for Hybrid Checkpointing in Time Warp Distributed Simulation. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):947–951, 1998.

[SR96] S. Skold and R. Ronngren. Event sensitive state saving in time warp parallel discrete event simulations. In *Proceedings Winter Simulation Conference*, pages 653–660, 1996.

[SRC09] Jianyong Shi, Aizhu Ren, and Chi Chen. Agent-based evacuation model of large public buildings under fire conditions. *Automation in Construction*, 18(3):338–347, 2009.

[ST13] Vinoth Suryanarayanan and Georgios Theodoropoulos. Synchronised range queries in distributed simulations of multiagent systems. *ACM Trans. Model. Comput. Simul.*, 23(4):25:1–25:25, November 2013.

[Sti09] Tim Stitt. *An introduction to the Partitioned Global Address Space (PGAS) programming model*. Connexions, Rice University, 2009.

[STL13] Vinoth Suryanarayanan, Georgios Theodoropoulos, and Michael Lees. PDES-MAS: distributed simulation of multi-agent systems. In *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*, pages 671–681, 2013.

[Sut13] Herb Sutter. The free lunch is over a fundamental turn toward concurrency in software. 2013.

[TPQ+17] Tommaso Tocci, Alessandro Pellegrini, Francesco Quaglia, Josep Casanovas-Garcia, and Toyotaro Suzumura. ORCHESTRA: an asynchronous wait-free distributed GVT algorithm. In *21st IEEE/ACM International Symposium on Distributed Simulation*

and Real Time Applications, DS-RT 2017, Rome, Italy, October 18-20, 2017, pages 51–58, 2017.

[TW04] Seth Tisue and Uri Wilensky. Netlogo: A simple environment for modeling complexity. In *Proceedings of the International Conference on Complex Systems*, ICCS, pages 1–10. NECSI, 2004.

[WCDC+20] Gary Weissman, Andrew Crane-Droesch, Corey Chivers, ThaiBinh Luong, Asaf Hanish, Michael Levy, Jason Lubken, Michael Becker, Michael Draugelis, George Anesi, Patrick Brennan, Jason Christie, C. III, Mark Mikkelsen, and Scott Halpern. Locally informed simulation to predict hospital capacity needs during the covid-19 pandemic. *Annals of Internal Medicine*, 173, 04 2020.

[WP96] D. West and K. Panesar. Automatic incremental state saving. In *Proceedings of Symposium on Parallel and Distributed Tools*, pages 78–85, 1996.

[WRC12] P Wittek and X Rubio-Campillo. Scalable agent-based modelling with cloud HPC resources for social simulations. In *Proceedings of the 4th International Conference on Cloud Computing Technology and Science*, CloudCom, pages 355–362. IEEE Computer Society, 2012.

[YEGL14] Yue Yu, Abdelkader El Kamel, Guanghong Gong, and Fengxia Li. Multi-agent based modeling and simulation of microscopic traffic in virtual reality system. *Simulation Modelling Practice and Theory*, 45:62–79, 2014.

[YP13] Srikanth B. Yoginath and Kalyan S. Perumalla. Optimized hypervisor scheduler for parallel discrete event simulations on virtual machine platforms. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, SimuTools, pages 1–9, March 2013.

[YP15] Srikanth B. Yoginath and Kalyan S. Perumalla. Efficient parallel discrete event simulation on cloud/virtual machine platforms. *ACM Trans. Model. Comput. Simul.*, 26(1):5:1–5:26, 2015.

[Zei19] Bernard P. Zeigler. Theory of modeling and simulation. pages 3–25. Academic Press, third edition edition, 2019.