Sapienza
Università di Roma

On Techniques to Handle Risk in Speculative Parallel Discrete-Event Simulation

Facoltà di Ingegneria
Ingegneria Informatica (XXXV cycle)

Andrea Piccione
ID number 1422045

Advisor
Prof. Alessandro Pellegrini

Co-Advisor
Prof. Bruno Ciciani

Academic Year 2022/2023

Thesis not yet defended

On Techniques to Handle Risk in Speculative Parallel Discrete-Event Simulation
PhD Thesis. Sapienza University of Rome

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

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

Abstract

In this thesis, the traditional concepts of risk and aggressiveness associated with speculative parallel/distributed execution of discrete-event simulations are revisited and re-examined, introducing a more general framework of interpretation. It is shown how these concepts, intrinsic to the speculative nature of parallel/distributed executions, can and should be addressed in competitive simulation engine implementations. The spectrum of possibilities related to the exploitation of risk and aggressiveness is explored from both a methodological and a technological point of view, on which strong emphasis is placed. We illustrate techniques that improve performance while providing several guarantees to model developers. In particular, it is shown how it is possible to combine different synchronisation algorithms to avoid thrashing phenomena and provide transparent support to the programmer that avoids known problems or significant performance drops. In general, revisiting the concepts of risk and aggressiveness opens up the possibility of repositioning concepts from the literature into a broader utilisation framework, which can be exploited in future lines of research to produce increasingly high-performance simulation systems.

Contents

Chapter 1

Introduction

Since ancient times, people have used simple models to understand and predict the behaviour of natural phenomena. For example, ancient Egyptians used models of the Nile River to predict flooding, and ancient Chinese used models of the lunar calendar to predict eclipses. Those were the first examples of *simulations.* Following the definition in the *Merriam-Webster* dictionary, a simulation is "the imitative representation of the functioning of one system or process by means of the functioning of another".

As technology advanced, so did the complexity of simulations. In the 17th century, scientists such as Galileo and Isaac Newton used mathematical models to understand and predict the motion of objects. By the 19th century, scientists and engineers used increasingly sophisticated simulations to understand and design complex systems, such as steam engines and telegraphs.

Until then, simulations were carried out on *pen and paper* or with the help of mechanical devices, which limited the range of tractable models. The advent of electronic computers in the 1950s and 1960s made it possible to run much more complex simulations. By the 1970s, computer simulations were used in various industries, including manufacturing, finance, and transportation.

The advent of the personal computer in the 1980s made it possible for individuals and small organizations to run simulations, leading to a significant increase in the number of them being run. Additionally, the development of dedicated computing infrastructures, such as supercomputers, has made it possible to run large-scale simulations with great detail and accuracy.

Today, computer simulation is an integral part of many industries and scientific research fields. With the continued advancements in computer technology, it is almost certain that computer simulation will continue to play an increasingly important role in helping us understand, design, and optimize complex systems.

The fundamental component of a simulation is its *model.* A simulation model is a mathematical (or, in general, formal) representation of a real-world system or process designed to mimic its behaviour over time. It is created using a set of equations, algorithms, and rules that describe the system's components, their interactions, and the mechanics governing their behaviour. In essence, the simulation model is what is actually being evaluated and run during a simulation.

Choosing the right *modelling paradigm* is an essential step in formulating a simulation model. A modelling paradigm is the approach used to represent the modelled system or process. Different modelling paradigms have distinct advantages and limits in their capabilities to express the characteristics of the system under study, and each is suited to different types of systems and scenarios.

The choice of the modelling paradigm can significantly impact how a simulation is run by affecting the types of algorithms and techniques available for its execution. Additionally, its choice can also impact the complexity and computational requirements of the simulation, or affect the ability to analyze and interpret the simulation results. In this work, we will primarily focus on the *discrete-event modelling* paradigm, but we will show that many findings also hold, in general, for simulation applications.

Discrete event simulation (DES) is a modelling paradigm that describes the behaviour of a system over time by mapping it to a series of distinct events that occur at specific points in time. These events mark an instantaneous change in the system's state, and the simulation's progression is based on the scheduling and sequencing of these events. This means that the simulation advances to the next state only when a new event is processed; no changes are assumed to occur in between events.

In DES models, this is not a limitation since the values of variables between two events are usually irrelevant for system dynamics or can be trivially computed in case of necessity. With this approach, the simulation time can repeatedly jump directly to the timestamp of the next scheduled event.

A discrete event simulation model can be simulated using a simple algorithm that utilizes a priority queue. The algorithm repeatedly extracts the next event from the queue, processes it causing an advancement in the simulation state, and then inserts any new events generated during the processing back into the queue. These basic operations are illustrated in figure 1.1.

Discrete event simulations are deemed useful because they provide a way to model complex systems that involve many interacting components and changing conditions over time in a way that is relatively easy to understand and analyze. This justifies their use in a wide range of applications, for example:

- Logistics: simulating the flow of products through a factory or supply chain to optimize production schedules and inventory management.
- Transportation: simulating the movement of vehicles, people, or cargo through a transportation network to optimize routes, traffic flow, and capacity.
- Healthcare: simulating the operation of hospitals and clinics to evaluate staffing and resource allocation, and to plan for emergency scenarios.
- Finance: simulating the behaviour of financial markets and portfolios to evaluate the performance and risk of investments.
- Defense and security: simulating the operation of military units, weapons systems, and emergency response teams to evaluate readiness and response plans.

As technology advances, the demand for simulation performance continues to rise. Simulation studies often require many runs of increasingly large and complex models, and applications based on DES are no exception. However, due to the physical limitations of silicon devices, it is impossible to increase the processing

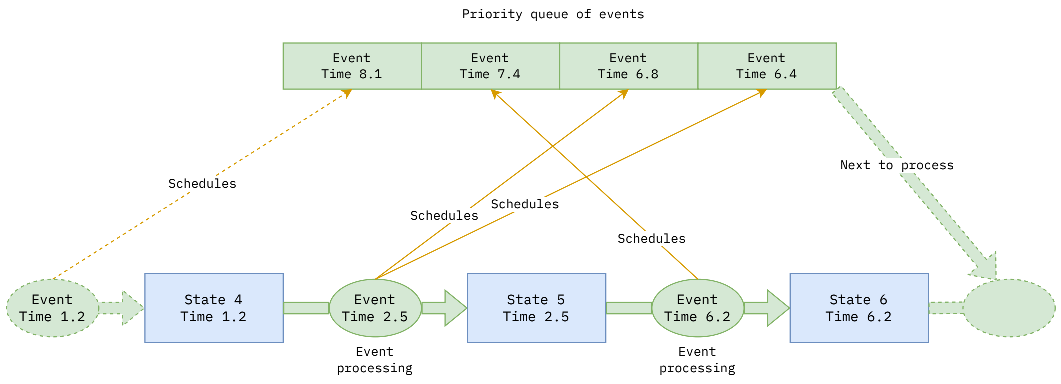


Figure 1.1. The operations of a sequential Discrete Event Simulation

power of a single CPU core significantly. This is why multi-core CPUs are now commonly used in desktop computers, smartphones, and even Internet of Things (IoT) devices. High-performance compute nodes use multiple multi-core CPUs, and supercomputers comprise many such servers, possibly hundreds of them.

Parallelization of simulations is, therefore, a crucial concern. When multiple runs of the same simulation are needed, for example for model calibration [170], executing them simultaneously on the same machine is feasible. However, some applications may require the outcome of a single simulation run in the shortest amount of time, or the simulation may be too large to fit in a single computer's memory or too slow to run on a single machine. These scenarios render necessary the parallelization of a single simulation run.

For DES applications, the approach to parallelizing these simulations is through the use of Parallel Discrete Event Simulation (PDES) techniques [38]. This involves dividing the simulation model's state into smaller, autonomous components called Logical Processes (LPs). These LPs can then be partitioned across multiple computing units and processed in parallel. Each LP can update its state and schedule new events, possibly to other LPs, independently of the rest of the simulation. However, indiscriminately executing events in parallel would yield incorrect results because an LP could execute an event with a particular timestamp before receiving a message with a lower timestamp from another LP; it is, therefore, necessary to ensure that events are executed in a synchronized manner to avoid any *causality violations*. This is achieved through the use of synchronization methods between the LPs. At this point in writing, we can simplistically divide synchronization algorithms into two distinct categories: *conservative* and *optimistic*. The former guarantees that no portion of the simulation state is ever incorrect. The latter admits transient situations of incorrectness that are corrected a posteriori by reconstructing a previous correct state.

Pure conservative methods use the concept of *Time Windows*. With this approach, the execution of an LP is temporarily paused if the next event to be processed has not been verified as correct. For example, in YAWNS (Yet Another Windowing Network Simulator) [92], each LP can only execute its pending events as long as their timestamps fall within a periodically computed virtual time window. This window starts at the minimum timestamp of the last executed event among all LPs and lasts for a period of virtual time equivalent to the *lookahead* value. The lookahead is the minimum amount of virtual time delay that an LP can schedule for a new event and is determined by the specific characteristics of the model. For example, in a computer network simulation, the lookahead can be the minimum simulated link latency. Without the lookahead information, it is not possible to use purely conservative methods. Research has devised several variants and improvements of conservative synchronization methods. Still, they all share the same fundamental property: an LP would execute and schedule (in parallel) only the events executed and scheduled by the corresponding sequential simulation.

On the other hand, optimistic synchronization methods do not try to maintain this property. A noteworthy example is the *Time Warp* protocol [59]. This approach allows causality violations to occur, but when inconsistencies are detected, the simulation is restored to a previous state to ensure system consistency. An incorrect order of event execution is identified when an LP tries to execute an event with a timestamp smaller than its last executed one. This event, known as a *straggler message*, causes the LP to roll back to a previous safe state and undo the effects of incorrectly executed events. These events are then re-executed in the correct order of timestamps. Additionally, events generated during the inconsistent simulation trajectory are undone by sending *anti-messages* to other involved LPs. The simulation state is committed periodically when LPs reach consensus on a consistent

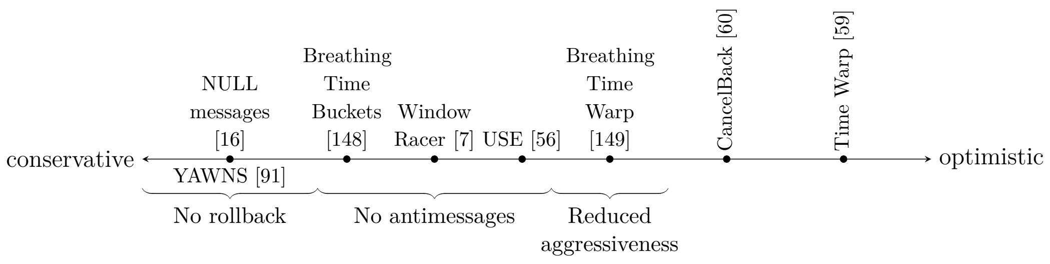


Figure 1.2. Spectrum of PDES Synchronisation Protocols.

virtual time—the so-called *Global Virtual Time*.

Put differently, conservative methods require lookahead information but not rollback support, while optimistic methods require rollbacks but not lookahead.

As a matter of fact, the two just-described PDES synchronization algorithms are at opposite ends of a much more complex spectrum worth exploring. Effectively, the field of PDES has developed various algorithms that prioritize different simulation properties, possibly giving up some, but not all, of the conservative guarantees. Research has shown that there's no such thing as the *best* synchronization method. The best method can only possibly be defined as a function of several factors such as the simulated model, the available computing resources, and many other considerations [134].

Reynolds et al. [134] have proposed an interesting distinction between *risk* and *aggressiveness* of these algorithms, where aggressiveness refers to the likelihood of executing events out of order, and risk relates to the possibility of producing and transmitting inconsistent events. In this framework, conservative methods are considered non-risky and non-aggressive, while the Time Warp protocol is deemed risky and aggressive.

Many variants to the Time Warp protocol have been proposed in the literature, showing that optimisation possibilities are many, as shown in Figure 1.2. For example, the *cancelback protocol* [60] makes it possible to limit aggressiveness if certain portions of the simulation are too far from the commit horizon and the system's memory pressure is too high. *Breathing Time Buckets* [148] is a protocol allowing events to be sent only if they are guaranteed to be valid, thus being risk-free, at the cost of drastically reduced aggressiveness. This protocol was extended in *Breathing*

Time Warp [149], allowing only events generated by those events that are closest to the commit horizon to be sent in order to reduce the probability of rollbacks. In *USE* [56], on the other hand, aggressiveness is controlled by using a global queue on a single processing node shared between all worker threads and adopting non-blocking policies for its management. In this way, a temporary binding between simulation entities and worker threads is realised to improve performance by attempting to reduce the impact of rollback operations.

As has been shown when comparing optimistic and conservative algorithms [13], the performance of optimistic algorithms depends heavily on the nature of the specific simulation model executed. For example, it was shown in [2] that the same simulation model could benefit from different levels of optimism depending on its configuration. Jefferson [61] also showed how it is possible and desirable to realise a simulation platform that can exploit different synchronisation protocols simultaneously by performing mode switches at runtime. In this way, it is possible to work on the various attributes of the simulation to maximise performance while limiting the possible adverse effects of optimism.

In general, the various synchronization protocols that have been proposed in the literature focus on event generation, management, and execution to control and limit risk and aggressiveness, for the purpose of improving simulation performance. We believe that aggressiveness and risk are only two facets of a more significant risk concept.

1.1 Thesis objectives and Contributions

In summary, the goal of this work is to present a comprehensive framework for interpreting and managing the risks and aggressiveness associated with PDES techniques. We have two primary objectives in this regard.

The first objective of our work is more methodological and is implied throughout the thesis. Our aim is to convince the reader that risk in parallel simulation is a complex concept that is not solely related to the choice between conservative and optimistic synchronization. Instead, it encompasses many aspects which, admittedly, sometimes are not taken in first consideration. The properties of the simulation algorithm, the characteristics of the simulated model, and even single specific implementation details of the simulation engine, all contribute to determining the position of a simulation run in the vast *risk space*.

The second objective is more technical, where we demonstrate various methods to manipulate and exploit the level of risk. For example, in Chapter 7, we illustrate how to reduce risk to increase fidelity with respect to serial simulation executions, while in Chapter 6, we show a technique to exploit risk to achieve potentially optimal tradeoffs in accuracy and performance.

The content of this thesis is partially based on the following publications that appeared in international conferences and journals:

[1] Piccione, A., Principe, M., Pellegrini, A., and Quaglia, F. (2023b). Approximated rollbacks: Effective state management in stochastic speculative pdes. *ACM Transactions on Modeling and Computer Simulation*. Under review.

[2] Piccione, A., Andelfinger, P., and Pellegrini, A. (2023a). Hybrid speculative synchronisation for parallel discrete event simulation. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '23, New York, NY, USA. ACM. To appear.

[3] Piccione, A. and Pellegrini, A. (2023). Practical tie-breaking for parallel/distributed simulations. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '23, New York, NY, USA. ACM. Under Review.

[4] Piccione, A. (2022). Comparing different event set management strategies in speculative pdes. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '22, page 55–56, New York, NY, USA. ACM.

[5] Andelfinger, P., Piccione, A., Pellegrini, A., and Uhrmacher, A. (2022). Comparing speculative synchronization algorithms for continuous-time agent-based simulations. In *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 57–66. **Winner of the Best Paper Award**.

[6] Pimpini, A., Piccione, A., and Pellegrini, A. (2022). On the accuracy and performance of spiking neural network simulations. In *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. **Shortlisted for the Best Paper Award**.

The author also contributed to the following publications, which, in varying degrees, have served as inspiration for the findings presented in this thesis:

[1] Du, X., Pimpini, A., Piccione, A., Meng, Z., Siguenza-Torres, A., Bortoli, S., Knoll, A., and Pellegrini, A. (2023). Autonomic orchestration of in-situ and in-transit data analytics for simulation studies. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '23, New York, NY, USA. ACM. Under Review.

[2] Piccione, A., Bernardinetti, G., Pellegrini, A., and Bianchi, G. (2023). Is your smartphone really safe? a wake-up call on android antivirus software effectiveness. In *Proceedings of the Italian Conference on Cybersecurity*, ITASEC '23, Bari, Italy. CEUR-WS.org.

[3] Pellegrini, A., Di Sanzo, P., Piccione, A., and Quaglia, F. (2022). Design and implementation of a fully transparent partial abort support for software transactional memory. *Software: Practice and Experience*, 52(11):2456–2475.

[4] Pimpini, A., Piccione, A., Ciciani, B., and Pellegrini, A. (2022). Speculative distributed simulation of very large spiking neural networks. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '22, page 93–104, New York, NY, USA. ACM.

[5] Piccione, A. and Pellegrini, A. (2020). Agent-based modeling and simulation for emergency scenarios: A holistic approach. In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–9.

[6] Principe, M., Piccione, A., Pellegrini, A., and Quaglia, F. (2020). Approximated rollbacks. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of*

Advanced Discrete Simulation, SIGSIM-PADS '20, page 23–33, New York, NY, USA. ACM.

[7] Piccione, A., Principe, M., Pellegrini, A., and Quaglia, F. (2019). An agent-based simulation api for speculative pdes runtime environments. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '19, page 83–94, New York, NY, USA. ACM.

Finally, as part of the commitment to the research community, the author has also been involved in the reproducibility initiative, for which the following reports have been produced:

[1] Piccione, A. (2020). Reproducibility report for the paper: Optimizing discrete simulations of the spread of hiv-1 to handle billions of cells on a workstation. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '20, page 79–81, New York, NY, USA. ACM.

1.2 Reference Implementations and Benchmarks

All the algorithms and methods that we present in this thesis have been implemented in ROOT-Sim [103], a parallel/distributed discrete event simulator. The source code of the different implementations can be found online[1]. Moreover, whenever possible, we have opted in the Reproducibility of Computational Results supported by the ACM. The interested reader can refer to the original papers for additional details on the specific implementations.

We used a combination of synthetic benchmarks and real-world applications to experimentally evaluate our proposals. These benchmarks are utilized throughout this thesis, so we will provide a brief overview of them here.

1.2.1 Synthetic Models

As a first testbed application, we have used the classical PHold benchmark [36]. It is a synthetic model that creates a series of events exchanged between simulation

[1] `https://github.com/ROOT-Sim/core`

entities. Its synthetic workload allows for the controlled adjustment of simulation parameters, such as the number of entities, events' delay, and distribution of the workload across the overall simulation.

We have also relied on a variant of PHold, called *PHold Memory*, which has been developed according to the specification in [161]. In PHold Memory, in addition to the conventional busy loop used to simulate the delay of a simulation event, each LP allocates a sparse state consisting of several linked data structures at the startup of the simulation. These data structures are synthetic and maintain a member indicating their size and a buffer filled with random data. During the execution of an event, the LP examines its sparse state and updates the contents of a subset of the buffers by writing additional random data. With a certain probability, one of the linked data structures is released from the simulation state of an object and sent to a random recipient as an event-message payload. Upon receiving such an event message, the destination LP installs a copy of the data structure in its simulation state and maintains its contents according to the logic described above.

PHold Memory enables capturing additional characteristics of simulation models related to memory utilisation. In particular, an LP can have a different state layout and content from other LPs (thus mimicking a scenario with several LPs representing various real-world entities). Furthermore, data exchange mimics the traditional behaviour of PDES applications, according to which LPs exchange information. Finally, the overall size of the simulation model (for all its objects) is kept constant.

1.2.2 Real-world Models

Regarding real-world applications, we have relied on two variants of agent-based Susceptible-Infected-Recovered (SIR) [67] models that supports the study of the spread of epidemics in large populations.

The first model focuses on tuberculosis (TBC). The model was developed at the Barcelona Supercomputing Center (BSC) [87] to study the spread of TBC in the Barcelona area. The model is based on agents (individuals) circulating in an area represented by several LPs. Each object models a region of the geographical area of interest, and the presence of agents in the region is recorded through the information stored in the state of the LP that models the region. Specifically, the object maintains a record for each agent currently residing in the corresponding region. The different agents model individuals who may be in one of five possible states based on the dynamics of TBC infection: *healthy*, *infected*, *sick* (i.e. with active TBC), *under treatment* and *cured*. The status variables of individuals refer mainly to their status in the TBC infection cycle and the time spent in these phases. Other individuals' parameters are age, native or immigrant origin, possible risk factors (e.g. smoking) and possible immunosuppression (mainly AIDS). Once a person has been infected, the presence (or absence) of lung cavitation is also considered.

For our second benchmark, we utilized a simulation model that examines the spread of contagion in an epidemic, which was initially introduced in [2]. This model extends the conventional agent-based formulation of the susceptible-infected-recovered (SIR) model [77]. To enhance understanding of these models, we will provide further background on memory-less stochastic processes in Section 3.2, where we will discuss the challenges in simulating them.

Chapter 2

Risk in Parallel Discrete-Event Simulation

Before delving deeper into the topic of risk in PDES, we need to establish and clarify some foundational concepts. This chapter will present a straightforward yet effective formalisation of discrete event simulation models. We will then comprehensively analyse parallelisation techniques and their associated trade-offs. In addition, we will examine our concept of risk in greater detail.

2.1 Discrete Event Simulation Fundamentals

In modelling discrete event systems, it is quite common to rely on rich formalisation, such as Discrete Event System Specification (DEVS) [174]. In the present writing, a different approach will be taken: a more implementation-centric formalisation will be adopted, which is deemed sufficient for the specific goals of this study. In particular, we will show that the naive definition of DES given in Chapter 1 already hides some problems that this work will instead try to address.

Definition 2.1.1 (Discrete-Event Simulation Model)**.** A DES model M is defined by the quadruple $\langle S, E, \delta, e_0 \rangle$ where:

- S is the set of possible model states.

For example, in a simulation of a point in a 2-D space, S could be the vector space $\mathbb{R}^4$, with each state configuration being a 4-dimensional vector, with the first two components representing the position and the latter two representing the velocity of the point.

- $E = \mathbb{R}_{\geq 0} \times D$ is the set of possible events.

  Event timestamps are represented as non-negative real numbers, and D contains the admissible event payloads. In addition, we define the function $time(e) : E \mapsto \mathbb{R}_{\geq 0}$ that computes the timestamp associated with an event.

  For example, in a computer network simulation [75], D could be $\mathbb{N}^2$, with each contained pair representing a network packet; the first component would indicate the size in bytes while the second component would encode its content.

- $\delta(s, e) : S \times E \mapsto S \times \{(e_0, e_1, \ldots, e_n) \mid e_k \in E\}$ is the model transition function.

  The transition function δ takes the current model's state and an event as input and computes the next state and a sequence of new events intended to be processed in the future.

- $e_0 \in \{0\} \times D$ is the initialisation event.

  We also assume that $\forall s_k, s_j \in S : \delta(s_k, e_0) = \delta(s_j, e_0)$. Basically, δ ignores its first argument when computed for the initialisation event. We call this computation *simulation initialisation.*

  This definition is a short-hand to avoid explicitly defining an initial state and a sequence of initially scheduled events.

In this context, the simulation initialisation process is able to deal with startup from arbitrary simulation states. Specifically, if the simulation must start from a certain state s_k and $e_i, \ldots, e_j$ are the events to initially generate, we can simply impose:

$$\forall s_j \in S : \delta(s_j, e_0) = (s_k, (e_i, \ldots, e_j)) \tag{2.1}$$

The intuition behind this formulation choice is that the initial state s_k is computed in the initialisation, which can possibly enclose warm start, cold start or even arbitrary calibration activities.

The simulation initialisation operation can also encode, for example, model configuration activities. In case of stochastic models, also the configuration of pseudo-random number generators can have an effect on the final outcome of the model. With this formalization, we assume that pseudo-random number generators (PRNGs) are part of the model specification. Consequently, PRNGs internal state is part of the simulation state and their initial seeding is handled by the simulation initialisation as well. This formalisation encapsulates away the variability associated to a model, with the aim of focusing on the correctness and reproducibility of a single simulation run.

With these considerations in mind, to capture an execution of a DES model, we can use the following;

Definition 2.1.2 (Simulation Trajectory)**.** A sequence of simulation states and events $T = (s_i, e_{i+1}, s_{i+1}, \dots, e_j, s_j)$ is a *simulation trajectory* if it satisfies the following two conditions:

- $\forall k \in \{i, \dots, j-1\} : time(e_k) \leq time(e_{k+1})$

  the sequence of events $(e_{i+1}, \dots, e_k)$ is ordered by non-decreasing timestamp.

- $\forall k \in \{i, \dots, j-1\} : \delta(s_k, e_{k+1}) = (s_{k+1}, (e_l, e_m, \dots, e_n))$

  the states sequence $(s_i, s_{i+1}, \dots, s_j)$ is generated by applying the transition function on the sequence of events $(e_{i+1}, \dots, e_j)$.

Given a single transition $\delta(s_{k-1}, e_k) = (s_k, (e_l, e_m, \dots, e_n))$ contained in a simulation trajectory T, we define as out_k the sequence of generated events $(e_l, e_m, \dots, e_n)$. Now, given a simulation trajectory $T = (s_i, e_{i+1}, s_{i+1}, \dots, e_j, s_j)$, we define the following two multi-sets:

- the n_{th} *processed events multiset* defined as $proc_n = \bigcup_{k=0}^{n} \{e_k\}$

- the n_{th} *generated events multiset* defined as $gen_n = \bigcup_{k=0}^{n} out_k$

Note that with this formalization it's possible to *arbitrarily* construct simulation trajectories. Given a sequence ordered by timestamp of events $e_1, \ldots, e_n$ picked arbitrarily from E, we can be evaluate it over δ starting from an arbitrary state s_k. We would then get this simulation trajectory T_*:

$$T_* = (s_k, e_1, s_{k+1}, e_2, \ldots, e_n, s_{k+n} \tag{2.2}$$

Clearly, when executing M with the intuitive semantics of a correct simulation, T_* may never occur — in this case we would call T_* an *inadmissible* trajectory. Put differently, we can use the following definition to determine whether a certain trajectory is correct with respect to the definition of a certain simulation model M.

Definition 2.1.3 (Simulation Correctness)**.** Finally, a simulation trajectory $T = (s_0, e_1, s_1, e_2, s_2 \ldots)$ is a *correct simulation* of model M if it satisfies:

- $(s_0, out_0) = \delta(s, e_0)$

  Note that s is irrelevant, given the properties of the simulation initialisation.

- $\forall e_i \in T : e_i \in (gen_{i-1} - proc_{i-1})$

  Each event in the trajectory has been generated in the past and has not already been processed.

- $\forall e_i \in T, \forall e_{ext} \in (gen_{i-1} - proc_{i-1}) : time(e_{ext}) \geq time(e_i)$

  Given an event contained in the trajectory, every other event that could have been processed has a higher or equal timestamp.

This formalisation is close enough to the operation of a simulation engine. We must now distinguish between *logical time* and *wall-clock time*. The former refers to the internal time associated with the simulation model, which is indicated by the timestamps of the events. In contrast, the latter refers to the actual amount of time the software engine requires to perform the computations involved in the simulation.

Our formalisation states that a simulation must commence with its initialisation and must process solely those events produced during the execution of the model.

Proper handling of events is fundamental, particularly concerning the logical time at which they occur. We enforce that the events with the earliest timestamps be considered first to ensure that all the events may be processed without incurring causality violations. This approach aligns with the proposal in the seminal paper in [73], which can produce a conservative-optimal Time Warp simulation and will be a fundamental building block to parallelise DES, as we shall discuss in Section 2.2.

By our definition, a transition function's continuous generation of new events can result in an indefinitely-long simulation trajectory. However, in practical terms, simulation practitioners are usually only interested in the computation until a certain logical time t specified in advance. Beyond that point, the computation may be terminated.

We also note that if a model admits a correct simulation, the transition function cannot generate events with a lower logical timestamp than the processed event when evaluated over an event in the simulation trajectory. This makes sense since violating this property would mean that a future state can influence a past state, which is absurd for the totality of models imitating the real world.

A different formulation of the transition function may have avoided this problem altogether. Unfortunately, this is not feasible in practice; a simulation engine cannot impose and, most importantly, verify constraints on arbitrary model code.

The definition of the DES model introduced earlier highlights other problems and limitations that must be tackled. The two most significant ones are:

- Multiple, possibly correct simulations are allowed when multiple events have the same timestamp. In practice, what should a simulation engine do in such cases?
- The transition function of a correct model is expected to be well-behaved during the simulation trajectory. However, if, for practical reasons, some parts of the model execution are approximated, for example, by computing speculative trajectories, what can we expect from the simulation engine?

These problems arise from the interaction of general simulation algorithms with arbitrary model implementations. In practice, a model implementation may not even deliver correct results if its events with the same timestamps are executed in some *unexpected* order. This is another facet of risk in simulation: indeed, a *risky* simulation engine may ignore the occurrence of same-timestamp events or assume that the model is always well-behaved, regardless of the events processed, even if they are possibly outside the correct simulation trajectory.

It is worth mentioning that additional engineering limitations need to be considered when transitioning such a formal simulation model into a computer program. One such limitation is the limited precision of the floating point representation of real-valued timestamps [47]. Although this may appear to be a separate issue from the formalism of simulation, as we will show, it can significantly impact the simulation performance and results.

2.2 Parallelizing DES

Simulation studies constitute increasingly sophisticated processes involving large-scale models and numerous simulation experiments, often computationally intensive. To support the execution of these models, relying on highly parallel and distributed High-Performance Computing (HPC) systems is imperative. HPC simulations make use of high-performance computing systems, such as massively parallel supercomputers, which enable unprecedented levels of accuracy in models and make previously intractable problems tractable by exploiting the enormous available computing power. Such simulation systems can allow for effective what-if analysis, as alternative scenarios can be explored through multiple concurrent simulations.

Indeed, if from the point of view of computing resources, we are now able to exploit exascale architectures [28, 29] thanks to extraordinary worldwide results [71, 41, 83, 141], it is evident that the reference implementation simulation engines should be aligned to the requirements of the new architectures, something that has already been done for the previous transition iteration towards petascale systems [163]. To efficiently do so, the above-mentioned formalisation of the DES model must be readapted to make model implementations friendly to exascale simulation engines. Conveniently enough, the traditional partitioning strategy proper of PDES [38] can be leveraged for this purpose.

To efficiently simulate large models, it is essential to partition the model's state into distinct *Logical Processes* (LPs) to achieve this. Each LP represents an independent unit comprised of its own internal state, and the simulation can be advanced by evaluating the behaviour of all LPs. Through the partitioning of the model into LPs, inter-LP communication and events computation can be easily distributed on the computing elements of a parallel machine.

Partitioning the model into LPs may require effort from the model developer. However, in many cases, the partitioning can be done naturally by modelling each system's independent *physical process* as a single LP. Physical process refers to a clearly identifiable, independent object, system, or process in the original model. For example, each server and router could be modelled as a separate LP in a computer network simulation. In some cases, determining the optimal partitioning of the model into LPs is not so obvious. For instance, in the case of a traffic simulation [172], the most intuitive partitioning may be individual vehicles. Still, the most efficient partitioning in terms of performance usually involves operating on the road network, with LPs being roads and intersections [98].

Therefore, selecting LPs in a simulation model can be an interesting task, as it involves a trade-off between simulation fidelity, computational efficiency, and ease of programmability. A finer partitioning of the model into LPs may result in more detailed simulations but also carry a higher event handling overhead. On the other hand, coarser partitioning may result in less accurate simulations, with more limited space for parallelisation but lower event handling demands. Nevertheless, in this thesis, we assume that the models have been partitioned into a reasonable set of LPs, as our focus is on the simulation algorithms themselves.

The previously established formalisation of a discrete event model can be expanded to accommodate multiple Logical Processes (LPs) rather than a single, "monolithic" state. A model is now comprised of n LPs $(LP_0, LP_1, \ldots, LP_{n-1})$, each identified by its own unique identifier and defined by the quadruple $\langle S^k, E^k, \delta^k, e_0^k \rangle$. Therefore, for a given LP_k we have that:

- S^k is the state space of the logical process.

- $E^k = \mathbb{R}_{\geq 0} \times \{k\} \times D^k$, where D^k is the set of possible event payloads.

  This is the set of events that the LP can process; the first component of the triple represents the event timestamp, while the second component encodes the LP identifier. Compared to the previous formalisation, we have that $E = \bigcup_{j=1}^{n} E^j$. We note that if $\forall k E^k = E$, then the model is uniform, i.e. all LPs can process the same set of events. While this is common for many models, a proper general-purpose simulation engine cannot base its organisation on this assumption.

- $\delta^k(s, e) : S^k \times E^k \mapsto S^k \times (e_0, e_1, \ldots, e_n) \mid e_k \in E$.

  This is the transition function of LP_k. It only consumes events destined for LP_k, but can also generate events destined for other LPs.

- e_0^k is the initialisation event of LP_k.

In a similar way, we can redefine the concept of simulation trajectory. First, we define a simulation trajectory as a set of per-LP trajectories, where each LP trajectory is a sequence $T_k = (s_i^k, e_{i+1}^k, s_{i+1}^k, \ldots, e_k^k, s_k^k)$ that satisfies the following two conditions:

- $\forall j \in \{i, \ldots, k-1\} : time(e_j) \leq time(e_{j+1})$

  the sequence of events $(e_{i+1}, \ldots, e_k)$ is ordered by non-decreasing timestamp.

- $\forall j \in \{i, \ldots, k-1\} : \delta(s_j, e_{j+1}) = (s_{j+1}, (e_l, e_m, \ldots, e_n))$

  the states sequence $(s_i, s_{i+1}, \ldots, s_k)$ is generated by applying the transition function on the sequence of events $(e_{i+1}, \ldots, e_k)$.

A global simulation trajectory T can be derived from the various T_k as follows. The function $time(\cdot)$ used above can be defined as $time : E \mapsto \mathbb{R}$, which induces a *strict weak order* $\prec_t$ on E by setting $e_1 \prec_t e_2$ if and only if $time(e_1) < time(e_2)$. Then, we can define T as:

$$T = \bigsqcup_{k=0}^{n-1} T_k \tag{2.3}$$

where $\sqcup$ denotes the *disjoint sequence union*, an extension of the sequence union operator [130] that preserves the strict weak order $\prec_t$ of all elements belonging to T_k in T.

Note that this definition is not able to generate a unique trajectory T from the various T_ks, because the weak total order defined above implicitly defines the equivalence $e_1 \sim e_2$ if and only if $time(e_1) = time(e_2)$. This means that multiple admissible simulation trajectories T can be obtained from a finite set of per-LP timelines T_k. This problem has a severe effect on the implementation of a parallel simulation runtime environment, with an impact also on the guarantees it can provide to the user and the modeller. These issues will be discussed in more detail in Chapter 7.

Equation (2.3) defines the simulation trajectory without explicitly defining the concept of *correctness*, which was clearly outlined in Definition 2.1.2 for sequential simulations. While it is possible to adapt the sequential version of correctness, this approach still results in multiple admissible global timelines T. The most effective way to define the correctness of T is to compare it with the timeline $\bar{T}$ generated from a sequential execution of the same model. This concept is similar to the notion of *linearizability* introduced by Herlihy and Wing in [54].

Referring indirectly the concept of linearizability, we can provide an informal definition of correctness for parallel executions as the existence of an equivalent correct serial execution. Therefore, as long as a parallel simulation produces a trajectory that can be generated by a correct serial engine, we can consider it to be correct.

Later in this work, we'll show that the existence of a correct simulation trajectory in the model is a necessary but not sufficient condition for the correctness of a parallel DES run. The existence of a correct simulation trajectory is only sufficient to guarantee correctness of a sequential simulation. Depending on the kind of parallel simulation algorithm, more conditions must be imposed on the behaviour of model M.

Thus, if we have a global simulation trajectory T of a DES model consisting of $LP_0, LP_1, \ldots, LP_{n-1}$, we can conclude that if T was generated by a correct serial simulation, then the individual trajectories $T_0, T_1, \ldots, T_{n-1}$ created by selecting, in order, the events designated for $LP_0, LP_1, \ldots, LP_{n-1}$ from T must also be correct. We note that in this case though, the correctness of an individual LP's trajectory at any given point in logical time will depend on the past defined on T which may include other LPs.

Therefore, a significant problem arises in the way of coordinating the execution of the LPs to maintain the illusion of a single, sequential simulation. This coordination is achieved through the use of *synchronisation algorithms*.

A synchronization algorithm is responsible for ensuring that the LPs are executed in a manner that preserves the causality of events. This means that simulation events occur in the same order as they would in a sequential simulation, thereby achieving correctness according to our informal definition.

In addition, using a suitable synchronisation algorithm reduces communication overhead between LPs, leading to a more efficient and scalable simulation. Without a properly designed algorithm, the communication overhead required for mimicking the results of a sequential simulation would significantly impact performance; a naive simulation engine would have to synchronise its processing units every time a new event is processed.

Therefore, we will now introduce the two main classes of algorithms that allow the parallel execution of a discrete event simulation model.

2.2.1 Conservative Synchronisation

Conservative synchronisation manages event processing among different LPs such that the state of the simulation always remains consistent. The process is considered "conservative" because it ensures that computations are always carried out without taking any risk, i.e. they are only performed when it is proven safe to do so, even if this implies that the simulation may not be as efficient as possible.

An essential requirement of conservative synchronisation is the *lookahead*, which refers to the amount of future time that the simulation is permitted to foresee when executing events. Essentially, the lookahead sets a guaranteed minimum interval of logical time before the next event is allowed to occur on an LP. The purpose of the lookahead is to ensure that events are executed in the correct order without explicitly synchronising the simulation after each processed event.

For example, the YAWNS algorithm [94, 91] is a conservative synchronisation algorithm that uses a window mechanism to ensure the consistency of the simulation among multiple LPs. The CMB algorithm [16], on the other hand, relies on NULL messages to notify connected LPs that no new event will be scheduled up to a certain simulation time—this approach requires knowing LP interconnections beforehand.

The YAWNS window mechanism works by periodically defining a time window, a logical time interval in the simulation, within which events can be executed. When an LP executes an event that would advance its local time beyond the end of the time window, it must wait until the next time window begins. The size of the time window is determined by the lookahead value, while its start is computed as the minimum timestamp reached by the LPs in the previous windows. This value is called Global Virtual Time (GVT) [84].

All the computations done during event processing are final and contribute to the simulation result. This desirable property comes at the cost of periodic blocking synchronisation. Indeed, whenever a time window is nearing its completion, processing units need to wait for the slowest one, usually using a barrier primitive. The computation of the GVT also requires another synchronisation step.

This algorithm is a worthy choice if the model exposes a *large enough* lookahead value; conversely, if the lookahead value is too small, the synchronisation activities between time windows will become frequent enough to result in an overhead that would outweigh every other benefit.

2.2.2 Optimistic Synchronisation

Optimistic synchronisation [59] is a different approach to synchronising the processing of events in parallel discrete event simulations. Unlike conservative synchronisation, which requires LPs to wait for each other before executing events, optimistic synchronisation allows them to proceed with their events even if they may lead to inconsistent states, with the assumption that the simulation will eventually converge to a consistent state.

The critical aspect of optimistic synchronisation is the utilisation of a rollback mechanism. This mechanism enables LPs to undo their computations if an inconsistency is detected a/posteriori. An inconsistency can occur when a LP has to execute a *straggler event*, i.e. an event with a lower timestamp than its last executed event.

The advantage of optimistic synchronisation is that it allows for event processing to potentially go further in a given span of wall-clock time compared to conservative synchronisation. Temporary deviations from the correct simulation trajectory are acceptable because, if needed, the rollback mechanism can restore the simulation to a previous, presumably correct, state.

Implementing optimistic synchronisation algorithms requires the support of a rollback mechanism, which can impose significant overhead on the system, both in terms of performance, memory utilisation and implementation effort [59, 35]. The rollback mechanism may require LPs states to be periodically checkpointed [104], or, in the alternative, the model code may need to be instrumented or manually modified to implement reverse computation capabilities [15].

In addition to the implementation costs, optimistic synchronisation algorithms can also result in sub-optimal performance if the rollback mechanism is overly utilised. This can occur in models with tightly connected components [4] or highly dynamic and difficult-to-balance simulation loads. These factors can cause the rollback mechanism to be abused, resulting in a situation in which most of the computing time is spent in checkpoint and restore operations.

2.3 The Notion of Risk

In Chapter 1, we briefly presented the concept of risk and aggressiveness in PDES. As anticipated, we believe that these two concepts are actually two facets of the same phenomenon, which for backward compatibility with the literature, we will generically call *risk*. Risk, revisited according to the framework of this thesis, is not to be understood as a phenomenon to be steered clear of but rather a simple dynamic intrinsic to speculative execution environments that must be carefully managed but which, at the same time, can be exploited. In particular, it can make it possible to reduce the time required to obtain results from one (or a set of) simulations by working on how far these results deviate from what would be obtained–with more significant time cost–by running a sequential simulation. In particular, we define the notion of risk as follows.

Definition 2.3.1 (Risk in Speculative Discrete-Event Simulation)**.** Risk is the natural propensity of a runtime environment for speculative simulation to *attempt* to maximise throughput by sacrificing the accuracy of results in a manner as compatible as possible with the modeller's expectations.

We note immediately that this loose definition of risk fuses together the concepts of aggressiveness, accuracy and risk identified in the seminal paper [133]. Furthermore, it clearly links the concept of the execution environment with the modeller since we believe that the development of any synchronisation algorithm, runtime optimisation, or API should be designed with the ultimate goal of simulation that we introduced at the beginning of Chapter 1 in mind: to enable the understanding and prediction of the behaviour of natural phenomena in the simplest possible manner. The author has a strong conviction that without the centrality of the model and the performance execution of models, any scientific approach will be of little use.

As mentioned above, our definition of risk is rather loose, leaving room for multiple interpretations, both conceptual and implementational. This is done on purpose since the boundaries of the concept of risk can be relatively broad, and similarly its effects. Indeed, we claim that dealing with risk in parallel simulation is a worthwhile endeavour even if it has *the potential* for causing:

- correctness issues, such as deadlocks, crashes, and inconsistent outputs, which may result from the corruption of the simulation process [93].
- poor simulation performance due to excessive memory usage, wasted work, inefficient CPU utilisation, and excessive wait times for shared resources [36].
- divergence from the results of the corresponding sequential simulation, i.e. producing an output which is correct and coherent with the model but not reproducible by the *theoretically equivalent* sequential simulation [99].

Although the concept of sacrificing performance in exchange for some degree of risk may seem enticing, this idea of risk in parallel simulation may not seem reasonable. One could argue that:

- correctness is a binary property; something is either correct or not. Synchronisation protocols are provably correct; therefore, a correctly implemented simulation should never encounter such an issue, while an incorrect one is to be discarded.
- if enough processing cores are used, a parallel simulation will always be faster than a sequential one, or at least will not be slower.
- it's redundant to mention *divergence* as a separate concept since a parallel simulation that produces different results than a sequential one would be incorrect, therefore, useless.

Unfortunately, transitioning from theoretical concepts to practical objects is not always straightforward. In the real world, we are not dealing with abstract algorithms and properties but with concrete code implementations, real computer architectures, and real needs.

Regarding performance, experience and research teach that simply using more cores in a parallel simulation may not always improve performance and that specific models may even perform worse than the sequential counterpart if their specific characteristics are not *carefully* taken into account [2].

As we implied previously, developing a parallel simulation that exactly matches the results of the sequential simulation is a challenging task and may require an unnecessary trade-off in performance. Indeed, a divergent simulation can still be helpful in certain practical situations [120]. Even a completely *non-deterministic* simulation—producing different outputs without any change in the input—can suffice for some applications.

Additionally, as our proposed formalisation highlights, a reliable DES model must only behave correctly along its *true* simulation trajectory. The outcome of any other trajectory is not necessarily defined. This challenges high-performance optimistic PDES engines that may process inconsistent or out-of-order events. To address this, executing the model's code inside a sandbox may be necessary to ensure correctness [23], although this can significantly impact performance or versatility. Failure to take precautions can result in undefined behaviour, such as divisions by zero, heap corruption, or deadlocks, for example, when processing events that require resources not-yet-released by some not-yet-delivered events [93].

In a way, this notion of risk in parallel simulations can be compared to financial risk. Just like in finance, the user must make informed decisions based on their available resources, in this case, computing power, and their willingness to tolerate risk, which corresponds to the requirements of the simulation in terms of accuracy and performance. Risk is a multidimensional concept that depends on the market—in our analogy, the simulation model. Similar to financial investments, it is possible to use all the available resources and underperform by making poor decisions, just as it might be more advantageous not to use all available resources and allocate them elsewhere.

In the next Chapter, we will present two compelling use cases for two very different instances of large-scale simulations that we believe would greatly benefit from the considerations about *risk* we have discussed.

Chapter 3

Case Study: The Effect of Risk in Real-World Simulations

To further highlight the importance and opportunities of carefully managing risk, in this Chapter, we present two case studies that exemplify the potential consequences if proper caution is not taken. We have selected as use cases two different real-world simulation models that exhibit very interesting behaviour and come from very distant domains, namely *epidemics* and *machine-learning/brain simulation*.

These use cases are interesting because, depending on their configuration, they have a behaviour that is easily parallelizable by traditional Time-Warp-based simulations, or they can easily thrash them. Moreover, concerning the brain-simulation model, the events have a very fine grain, and the interconnections can be extremely dense, making it hard to simulate optimistically.

Overall, this Chapter introduces some early results obtained on the path to the construction of the techniques presented in this thesis and showcases some of the building blocks that will be exploited in later chapters to manage and handle risk.

3.1 Spiking Neural Networks

Spiking Neural Networks (SNNs) have grown significantly in recent years due to their versatility and expressiveness [43]. This particular type of Artificial Neural Network (ANNs) is helpful in various domains, including neuroscience, medicine, artificial intelligence, and psychology, as they highly accurately mimic biological neural networks. Additionally, compared to traditional neural networks, SNNs have a lower energy consumption when implemented in hardware, which has led to the development of neuromorphic chips [48, 113, 118]. These chips aim to strike a very favourable balance between energy efficiency and performance in large-scale parallel computing systems.

Studying the behaviour of SNNs is typically done through simulations because analytical treatment is only possible for specific, simplified cases [11]. In SNN simulations, neurons are modelled individually, and they interact with each other through the exchange of spikes. The changes in a neuron's state can result in the emission of a spike delivered to connected neurons. To maintain consistency, a neuron's state can only be updated after it has received all spikes with smaller timestamps.

SNNs are more complex to work with than traditional ANNs because they encode information in a temporal domain, known as the spike train [9]. The output of an SNN is a series of impulses that convey information through their timing rather than a set of values computed impulsively, as in other ANNs. Additionally, the connections between neurons can be arbitrary, and a single neuron can receive multiple spikes in a short simulation window, making SNNs a part of the continuous-time tightly-coupled models family, which is challenging to simulate [132, 44, 3].

In order to handle the complexity of this type of simulation, the continuous time is discretized, and conservative methods are used for time-stepped simulations [65, 94, 32]. This involves monitoring the state of the simulation at set intervals and determining if new events should be generated based on the observed conditions. The advantage of this approach is that it allows for using existing conservative methods to efficiently support the simulation on multiple hardware instances [153, 70, 19, 173].

However, the conservative time-stepped approach is not without limitations. Firstly, simulation algorithms for SNNs generally use a fixed time-step for both simulation and integration, which can result in approximate results, i.e. the simulation may miss some spikes, even when subthreshold dynamics (the neuron state evolution in the absence of spikes) are linear [51]. Secondly, the time-stepped nature of SNN simulation algorithms has favoured using *easy to compute* neuron models, for instance, by numerical integration with the Euler method. Finally, reducing the time step to improve the accuracy of results would result in significant performance penalties. Recent studies [117, 115], however, have shown that the use of Parallel Discrete Event Simulation (PDES) methods [35] with optimistic synchronization [59] offers improved performance and accuracy. PDES can efficiently skip time intervals during which there are no interactions between neurons and capture the exact simulation time at which a neuron may spike. Additionally, an optimistic synchronization algorithm such as Time Warp [59] can capture the parallel nature of SNNs, where groups of neurons may spike at different times.

Here, we evaluate the performance and accuracy of modern time-stepped simulation algorithms [42] by varying the time step values. Additionally, we compare these outcomes with a state-of-the-art simulation algorithm for SNN, which utilizes the Time Warp synchronization algorithm [115]. By working on the risk concept introduced in this thesis, these results show how it is possible to obtain speculative simulations that provide superior performance results to traditional techniques that have been studied for decades, particularly for large networks.

In particular, we intend to show that, although possibly anti-intuitive, the significant risk associated with using the time-stepped approach in simulations may be higher than that obtainable with speculative DES. Specifically, even a basic model can exhibit a substantial divergence between the sequential and timestep-integrated simulations. Therefore, the simulation risk is not exclusively related to the choice between conservative and optimistic synchronization. Every design decision in a simulation is a tradeoff between performance, correctness, and accuracy. In the context of spiking neural networks, we contend that the conventional approach is being employed *without* the necessary considerations of correctness and accuracy.

Now, we introduce some more background on SNNs, which is necessary for a better understanding of the work.

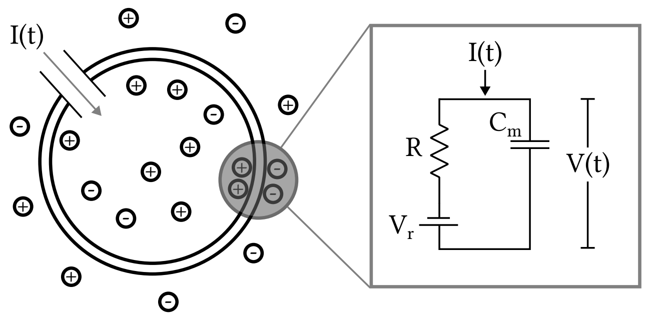


Figure 3.1. A neuron is enclosed by the cell membrane (the circle). When it receives a positive input current $I(t)$, it increases the electrical charge inside the cell. The cell membrane acts like a capacitor in parallel with a resistor, which is in line with a battery of potential V_r.

3.1.1 Background on Spiking Neural Networks

SNNs utilize *spiking neurons* that communicate via synapses, transmitting signals in the form of spikes. Spiking neurons and their connecting synapses are *stateful* in nature. These neurons fire only when their *membrane potential* surpasses a specific threshold value, leading to the generation of a spike. This spike is propagated to other neurons connected to the neuron, causing their membrane potential to increase or decrease over time. However, the spike passes through synapses, which have a weighted and time-varying nature introducing a *transmission delay.*

Spiking-neuron models are inspired by the behaviour of biological neurons, which respond to and communicate through electrical signals, allowing them to be modelled as circuits. The plasma membrane properties of neurons give rise to a membrane capacitance C_m, and the potential $V(t)$ at time t across the membrane triggers the propagation of an action potential when it reaches a threshold value of V_{th}. Without stimuli, the membrane potential returns to a resting value V_r. Following the generation of an action potential (also known as firing or spiking), an enzyme restores the neuron to its resting state over a period of time, known as the refractory period τ_{ref}, during which the membrane does not depolarize and the neuron is unable to spike.

In order for the membrane potential of a neuron to increase, it requires an input current I. This current is typically the sum of the stimuli $I(t)$ received from the neuron's presynaptic neurons, as well as any external current I_{ext} that may be applied (e.g. for experimental purposes). To clarify, the presynaptic neurons are those whose outputs are received by the considered neuron, while postsynaptic neurons are the ones that receive spikes from the neuron.

The *subthreshold dynamics* of a neuron refer to its behaviour before it reaches the threshold value for firing a spike. In other words, it describes the membrane potential changes and current flows that happen when the neuron processes incoming signals but does not generate an output spike. A neuron model usually expresses them as a set of differential equations.

The most commonly-used neuron model in large SNN simulations is the *Leaky Integrate and Fire* (LIF), which is depicted in Figure 3.1. The subthreshold dynamics of the neuron are described by Equations (3.1), where $V(t)$ is the membrane potential and $I(t)$ is the current flowing inside the neuron:

$$\begin{gathered}\frac{\mathrm{d}V(t)}{\mathrm{d}t} = \frac{-V(t) + V_r}{\tau_m} + \frac{I(t) + I_{ext}}{C_m} \\ \frac{\mathrm{d}I(t)}{\mathrm{d}t} = -\frac{I(t)}{\tau_{syn}}\end{gathered} \tag{3.1}$$

The membrane time constant and the synaptic time constant are represented by the positive constants τ_m and τ_{syn}, respectively, as described in [9]. Further details on the meaning of all neuronal parameters can be found in this reference.

Postsynaptic neurons receive spikes with a delay in virtual time, and the synapse model determines the effect of the spike in terms of the potential delivered to the postsynaptic neuron. In most PDES simulations, a simple synapse model called *jump synapse* is used, which has a fixed transmission delay t_{trans} and weight w. When a spike is transmitted through a jump synapse, it immediately increases the postsynaptic neuron's $V(t)$ by w.

The *exponential synapse*, also known as *instantaneous raise/exponential decay synapse*, is a more sophisticated synapse model in which spikes cause an instantaneous increase in the neuron's input current rather than acting directly on the membrane potential. The current's effects are then gradually applied over time to the membrane potential, similar to the charging of an electronic capacitor, while its intensity decreases exponentially. This implies that the neuron may spike at some future time. If no analytical solution for spike timing is available, it must be

Algorithm 1 Time Stepped Simulation Algorithm.

```
1: $t = 0$
2: while $t < t_{end}$ do
3:     for each neuron do
4:         process incoming spikes
5:         advance neuron dynamics by $\mathrm{d}t$
6:         for each neuron do
7:             if $V(t) > V_{th}$ then
8:                 reset neuron membrane
9:                 for each connection do
10:                    send spike
11:                end for
12:            end if
13:        end for
14:        $t \leftarrow t + \mathrm{d}t$
15:    end for
16: end while
```

determined numerically and enqueued as a future event.

3.1.2 Simulation Algorithms for SNN

This section illustrates the inner workings of the two SNN simulation algorithms we consider in the assessment we are carrying out for the present case study.

Time Stepped Simulations

The majority of SNN simulations, such as those implemented by the well-known NEST [42] and Brian [150] simulators, depend on time-stepped algorithms, the high-level pseudocode for which may be found in Algorithm 1.

This kind of simulation approach is simple. Indeed, all neuron state updates are evaluated periodically by processing the incoming spikes. These spikes increase membrane potential, which is again evaluated numerically in the interval $\mathrm{d}t$. After updating all neurons' states, the simulation algorithm checks which (if any) have a membrane potential V_m that has reached the spiking threshold. If so, spikes are sent from each ready-to-spike neuron to the respective postsynaptic neurons. In order to account for synapse delays, the typical strategy is to rely on some sort of future event queue, typically implemented as a circular array [9], that allows keeping track of what spike should be delivered to what neuron at what time(step) in the future.

The time complexity of this simulation algorithm can be easily computed. The first inner loop accounts for neuron state updates. If there are n neurons in the network, the loop has an $O(n)$ cost. Considering that the physical time of the simulation is divided into intervals of the same size, the cost is $O(n/\,\mathrm{d}t)$ per unit of physical time. Concerning the second inner loop, if we call f the average firing rate of neurons per physical-time unit, assuming that, on average, each neuron is connected to s other neurons, the cost is $O(fns)$. Under general assumptions, it cannot be stated which of the two components impacts the overall cost more. We can therefore conclude that the cost of this algorithm *per physical-time unit* is:

$$O\left(\frac{n}{\mathrm{d}t} + fns\right). \tag{3.2}$$

In this computation, we have assumed that activities related to the computation of neuron dynamics and spike delivery are negligible. However, depending on the specific used neuron model and the complexity of the topology, it might not always be the case. Anyhow, Equation (3.2) indicates that the overall cost of the time-stepped simulation grows with the network's size and the simulation's precision, which is exactly one of the key points we assess experimentally in this Chapter.

An additional issue with the time-stepped simulation algorithm is that spike timings are aligned to a grid defined by the time steps. Therefore, the final result approximates the actual behaviour of the network, even when the numerical methods used to compute differential equations can provide much more accurate results. Similarly, since the check on the threshold is carried out only at the time steps (see line 7 in Algorithm 1), some spikes may be missed. This is the second key point that we assess experimentally in this Chapter.

Speculative Discrete-Event Simulations

For speculative DES executions, we consider the organization of the model discussed in Section 2.2. According to the simulation algorithm described in [115], each neuron is mapped to a single LP.

Spikes are represented by messages, which are delivered to the destination LP. Since a single neuron can be connected to multiple neurons, injecting one spike event for each destination LP could easily thrash the simulation due to the significant time

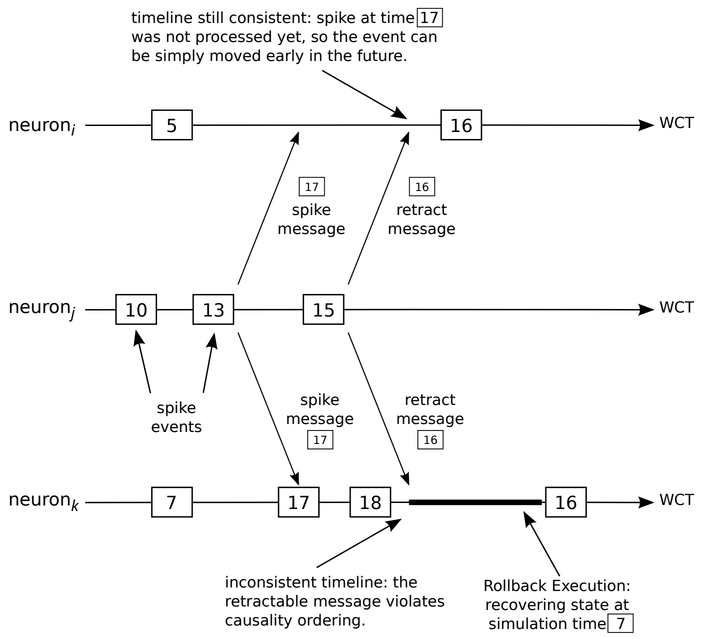

timeline still consistent: spike at time 17 was not processed yet, so the event can be simply moved early in the future.
neuron$_i$
5
16
WCT
17 spike message
16 retract message
neuron$_j$
10
13
15
WCT
spike events
spike message 17
retract message 16
neuron$_k$
7
17
18
16
WCT
inconsistent timeline: the retractable message violates causality ordering.
Rollback Execution: recovering state at simulation time 7

Figure 3.2. Retractable Spikes Naïve Scheme. A neuron can decide to change the time of a spike already injected in the system. Receiving neurons might have to roll back part of their execution.

spent on event management. Therefore, cross-neuron communication is supported by a form of publish/subscribe events: a spiking neuron will inject a single instance of the spiking event into the system. All destination neurons will subscribe to the events generated by the source one, and the underlying runtime environment will deliver *a copy* of the spiking event, thus significantly reducing the burden on the messaging subsystem.

According to the traditional Time Warp protocol, events are executed independently of their safety. It means that a destination neuron could receive a spike after the simulation time at which it had to be processed. In this case, the state of the neuron is rolled back to a previous time instant, and execution is resumed from a consistent snapshot. During rollback execution, inconsistently-generated spikes are undone by generating so-called antimessages. An antimessage reception could cause additional cascading rollbacks.

Given the nature of the spikes, it is impossible to consistently predict the spiking time given the current state of the neuron. Indeed, a more accurate spiking time could be determined after the neuron receives an upcoming spike. A simple solution at the model level could be to inject in the system *tentative* spikes, i.e. events that could be associated with some per-neuron epoch counter. Every time the neuron state is updated due to the receipt of an incoming spike, the new spiking time could be re-computed. A new spiking event (superseding the previous one) could be injected into the system. Anyhow, this naïve approach is unlikely to scale due to the large amount of extremely-fine grained simulation events that typically cause poor performance in Time Warp simulations [36]. The performance degradation of this scheme stems from the strict decoupling between the model and the runtime environment in Time Warp simulations. In this scenario, the model cannot inform the runtime environment that a tentative spiking event should be removed from the system. The model can only logically discard it once it is delivered for execution.

For this reason, in [115], the authors introduce the concept of *retractable events*, i.e. events that the model can mark as *tentative*. Logically speaking, this support allows implementing tentative spike events, according to the scheme depicted in Figure 3.2. After a neuron has sent its spikes, a new event could update the time of said spikes. This is the case, e.g., of new spikes being delivered to the neuron, which consequently charges faster, thus reaching the threshold V_{th} earlier. In this case, the neuron can inform the receiving neurons that the spike should be dealt with earlier. At the destination neurons, if the spike has not been processed yet, the event is moved earlier. Conversely, if it has already been processed, a traditional rollback operation will restore the neuron state to a consistent timestamp, and the new spiking time will be considered in the simulation.

The problem with this naïve approach is that the total number of rollbacks can still be high. Therefore, a straightforward optimization is to deal with retractable events locally at a single neuron. A neuron locally determines its next firing time and schedules to itself a *tentative spike-firing event.* This tentative firing event is managed as a regular firing event (i.e., the neuron sends the spikes to all destination neurons upon receiving it) if no change in the firing time occurs. Conversely, if the neuron model determines a new timestamp for the firing event, the runtime environment will accordingly act on the message queue. In particular, if the firing event is not yet processed, it will simply be moved to the appropriate new firing time. In this way, the number of events injected into the system and the total number of rollbacks is significantly reduced, as the destination LPs will only receive a spike at the accurate firing time; after that, the firing neuron has correctly received all presynaptic stimuli.

3.1.3 Experimental Comparison of the Approaches

Concerning time-stepped simulations, we rely on the NEST simulator [42], while for the discrete-event simulation, we rely on the implementation presented in [115], which is based on ROOT-Sim [103].

Experimental Setup

The performance experiments were run on an AWS m5.8xlarge machine with 32 vCPUs. These machines are based on Intel Xeon® Platinum 8175M processors, running Ubuntu 20.04.3 LTS, on kernel version 5.13.0-1025-aws. Each experiment was run with 32, 24, 16, 8, and 4 worker threads. NEST only has data points for 16 or more worker threads due to a limitation not allowing more than 2^{27} synaptic connections per worker thread; as such, only ROOT-Sim was run on 4 and 8 workers.

The standard benchmark we have used to run the performance experiments is inspired by a study on signal propagation in linear integrate-and-fire (LIF) models [164]. This benchmark [9] considers current-based (CUBA) synaptic interactions in a network of 300,000 LIF neurons, separated into two populations of excitatory and inhibitory neurons, forming 80% and 20% of the neurons, respectively. All neurons are connected randomly using a connection probability of 2%. The CUBA model is simulated for 10 seconds of simulation time with each simulator while varying the simulation precision. In NEST, this is achieved by selecting a resolution value. In ROOT-Sim, the time tolerance is currently built into the model and can be selected deliberately as long as the hardware constraints allow it.

We have built a synthetic network model consisting of 1,000 neurons for accuracy experiments. The network is acyclic, divided into four layers: Input, L1, L2, and Output. A network topology scheme is found in Figure 3.3. The Input layer comprises 100 excitatory neurons, which receive a constant input current of $1800pA$.

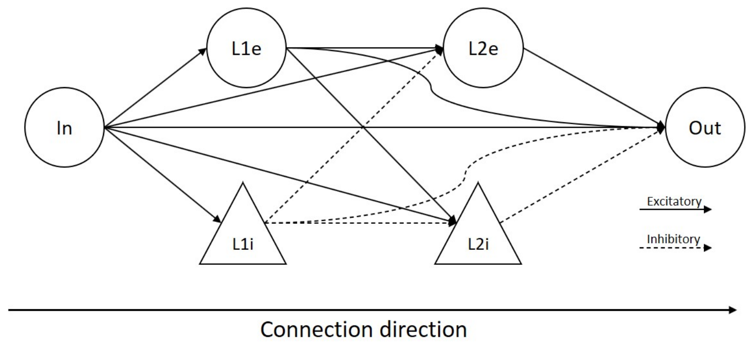


Figure 3.3. Schema of the Synthetic Model.

Layers L1 and L2 both comprise two populations of 100 excitatory (L1e/L2e) and 100 inhibitory neurons (L1i/L2i). The Output layer consists of 100 neurons. The output neurons' spikes are observed and compared with the ground truth to determine simulator accuracy. Synapses all have fixed weights of $200pA$ with a delay of $1.5ms$ when excitatory and a weight of $-600pA$ and a delay of $0.8ms$ when inhibitory.

The network topology and relevant parameters (initial membrane potential, input current, synaptic weight, synaptic delay) are generated with a script into a configuration file, which then is loaded by the models of each simulator, as well as by the script that computes the ground truth, leading to the very same topology and initial conditions for every single neuron in all three cases.

Table 3.1. Neurons and populations parameter specification.

Populations and inputs

Name	Input	L1e	L1i	L2e	L2i	Output
Population size	100	200	200	200	200	100

Neuron Model

Name	Value	Description
τ_m	10 ms	Membrane time constant
τ_{ref}	2 ms	Absolute refractory period
τ_{syn}	0.5 ms	Postsynaptic current time constant
C_m	250 pF	Membrane capacity
V_{reset}	-65 mV	Reset potential
V_{th}	-50 mV	Fixed firing threshold

Table 3.2. Connectivity map for the generated topology.

		to					
		In	L1e	L1i	L2e	L2i	Out
from	In	-	0.292	0.192	0.049	0.237	0.169
	L1e	-	-	-	0.106	0.254	0.438
	L1i	-	-	-	0.409	0.250	0.309
	L2e	-	-	-	-	-	0.491
	L2i	-	-	-	-	-	0.225

Computing the Ground Truth

As noted earlier, the network chosen for the accuracy evaluation is acyclic, and, conveniently, there is a simple algorithm able to compute its behaviour. Given such an acyclic network, we compute a topological order of the neurons $n_0, n_1, ...n_k$; then, necessarily, the behaviour of a neuron n_i will only depend on the behaviour of neurons $n_0, n_1, ..., n_{i-1}$. That implies that once a simulation time limit t has been selected, it is possible to simulate the neurons one by one, starting from n_0 through n_k feeding the output from the neurons to the correct postsynaptic ones. Since there is no analytical closed-form solution for the spike times for the LIF neuron used in this network, we still have to resort to numerical methods. We are not concerned with performance in this case. Therefore, we implemented a Python script carrying out the described computations with an error factor of $10^{-9} ms$.

Accuracy and Performance Results

We report in Tables 3.4 and 3.5 the results obtained running the synthetic model on ROOT-Sim and NEST, compared to the ground truth results obtained according to the method described in Section 3.1.3. We have set the timestep/accuracy factor to 0.1 (Table 3.4) and 0.001 (Table 3.5). The results report the spikes obtained in the simulation's first 10 ms. We provide the spiking time and the ID of the neuron

Table 3.3. Synaptic parameter specification.

Name	Value	Description
w_{exc}	200 pA	Excitatory synaptic strengths
w_{inh}	-600pA	Inhibitory synaptic strength
d_e	1.5 ms	Excitatory synaptic transmission delays
d_i	0.8 ms	Inhibitory synaptic transmission delays

Table 3.4. Spiking Times for ROOT-Sim and NEST (timestep/error: 0.1). For each result, we provide the spike time (*ms*) and the neuron number in brackets. The results relate to the first 10 *ms* of simulated time.

Spike No.	Ground Truth	ROOT-Sim	NEST
1	2.999 (900)	3.046 (900)	3.200 (900)
2	3.556 (977)	3.615 (977)	3.900 (975)
3	3.598 (975)	3.630 (975)	3.900 (950)
4	3.787 (970)	3.771 (970)	6.200 (912)
5	5.843 (953)	5.955 (953)	6.300 (952)
6	—	6.215 (927)	—
7	—	6.667 (923)	—

that generated the spike in the Output layer for each spike.

By the results in Table 3.4, we observe that, for both simulators, the accuracy is not high. In particular, ROOT-Sim generates spikes at the correct neurons, but the difference in spiking times is between 1% and 2%, in a significantly reduced simulation time. Interestingly, this model generates two additional spurious spikes. Conversely, NEST has a higher error (up to 60%), but more interestingly, it induces spikes at the wrong neurons, except for the first one. The number of spikes is anyhow correct. These results are expected. Indeed, given the nature of the synthetic model, it is clear that a low resolution is unlikely to provide accurate results due to the strong interaction between excitatory and inhibitory neurons.

The results with a higher resolution, provided in Table 3.5, show that the results based on ROOT-Sim deliver much higher accuracy. Conversely, NEST results show that two spikes are missing, spikes are induced at the wrong neurons, and the accuracy is still low (with an error ranging from 3.7% to 80%). One could wonder how the two examined simulators may deliver a different accuracy even when using the same value for the time-step/error. As mentioned in Section 3.1.2, the sources

Table 3.5. Spiking Times for ROOT-Sim and NEST (timestep/error: 0.001). We provide the spike time (*ms*) and the neuron number in brackets for each result. The results relate to the first 10 *ms* of simulated time.

Spike No.	Ground Truth	ROOT-Sim	NEST
1	2.999 (900)	2.999 (900)	3.110 (900)
2	3.556 (977)	3.556 (977)	3.795 (975)
3	3.598 (975)	3.598 (975)	6.486 (952)
4	3.787 (970)	3.787 (970)	—
5	5.843 (953)	5.842 (953)	—

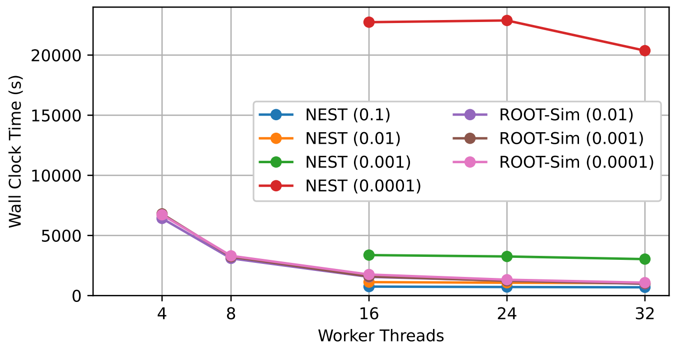


Figure 3.4. Performance Comparison.

of inaccuracy are essentially two. Common to both algorithms, the first is due to computational inaccuracy in spike timings: minor deviations can cause postsynaptic neurons to emit or miss a spike when they should not have. The second source of accuracy loss is specific to NEST only, and it is due to how spike detection works. With the default settings, a spike is detected only if the firing threshold potential is overcome at one discrete time step. In other words, NEST assumes that a neuron can never overcome the firing threshold if it has not done so at the beginning and the end of the time step, which can lead to missing a spike in some edge cases, even with a single neuron.

As for performance evaluation, we can refer to Figure 3.4, where both simulators used the standard current-based (CUBA) synaptic interactions benchmark [9] to simulate 10 seconds of physical time, with varying degrees of accuracy. NEST only has data for 16 or more workers because it does not allow for more than 2^{27} synaptic connections per worker thread. While NEST outperforms ROOT-Sim in terms of speed for low-resolution values (10^{-1} and 10^{-2}), when running with a resolution of 10^{-3}, the performance dramatically degrades, leaving the edge to ROOT-Sim, even when the latter runs on eight workers. With a resolution of 10^{-4}, the NEST time-to-solution is significantly larger, with the best configuration (using 32 worker threads), taking over $20,369$ seconds to complete, while ROOT-Sim took $1,076$—this is 18,92x. This result is expected, as multiplying the resolution tenfold also multiplies the number of calculations needed. It is reasonable to expect higher resolutions to be practically unfeasible for sizeable networks.

Increasing simulation resolution appears to have a minimal impact on ROOT-Sim's performance, allowing it to be increased almost at will without the risk of running into prohibitive time costs.

3.2 Epidemic Models

Many natural phenomena can be modeled as memoryless stochastic processes in which transitions occur randomly with exponentially distributed delays in continuous time. Such systems are present in several fields, including biochemistry, genetics, ecology, epidemiology, and social sciences.

In this section, we will explore two distinct methods for simulating epidemiological models based on memoryless stochastic processes that employ the Next Reaction Method (NRM)[44], a variant of Gillespie's Stochastic Simulation Algorithm (SSA)[45]. The two methods under consideration differ in their exploitation of risk and offer different accuracy and performance tradeoffs.

NRM was originally developed to simulate complex biochemical processes involving numerous concurrent chemical reactions. The method utilizes a *stochastic race* mechanism to determine the next reaction, where reactions compete based on randomly generated delays. The algorithm follows a discrete-event approach, where potential reactions are represented as timestamped events. The simulation repeatedly selects and executes the reaction with the nearest timestamp and adjusts the rates of other reactions to reflect any changes.

The application of this methodology to epidemiological models is based on recent results in [55, 167], which showed that it is possible to apply this method to agent-based models and demographic studies, respectively. Here, stochastic races select which possible agents' transition will happen next. During each transition, individuals can access arbitrary portions of the simulation state and affect the transition rates of other agents.

In our study of the impact of risk on simulation outcomes, this class of models is of particular interest. Traditionally, optimistic algorithms face challenges when simulating these models since each transition may involve instant read and write access to the states of agents located on different processors. Correctly handling these accesses requires linking the progress of the source and destination agents in simulation time, which may result in rollbacks or idle times. Additionally, Time Warp's assumption of completely event-driven interaction among LPs requires that every read and write access between agents be reflected by an event exchange, which may incur a significant cost.

One potential workaround to this issue involves discretizing simulated time and utilizing established conservative methods for time-stepped agent-based simulation. However, as demonstrated in the previous example about spiking neural networks, this approach may lead to poor accuracy-performance tradeoffs. As a matter of fact, prior research has shown that reproducing the results of even a simple cellular automaton with rate-driven transitions necessitates the use of tiny time steps [70, 19, 173].

Physics-inspired models, like Social Force [53] or Intelligent Driver Model [68], which are initially defined in terms of ordinary differential equations, can be simulated through discretization of the logical time dimension, where the time step represents the numerical integration step. This allows the error to be constrained by choosing an appropriate integration scheme [157]. However, determining suitable bounds for the time step size can be challenging, and if the step size is too large, the error can become significant [169].

We note that the minimum delay of any effect propagating from one agent to the next is constrained by the step size. Assuming that transitions occur at rate r, the time until the next transition is exponentially distributed with mean $1/r$. Although a time-driven simulation can imitate such processes, it relies on carrying out a Bernoulli trial at each time step to determine if a transition occurs. But this way, the time step size effectively becomes the lower bound on the time between two transitions. Since simulations often involve the propagation of effects across large numbers of LPs, small time step sizes may be required to adequately represent the dynamics of a continuous-time reference model.

We illustrate this issue on a simple cellular model on a 50×50-cells grid. Cells carry a binary *alive*/*dead* state. Initially, all cells apart from the top-left one are alive. At a rate of 1, each dead cell causes all its Moore neighbours to die simulta-

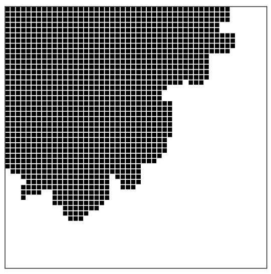


Figure 3.5. Illustration of the cellular simulation on a 50 × 50 grid. In this intermediate state, half of the cells are still alive.

neously. The simulation ends once all cells have died. We compare the simulation end times when executing the model in continuous time as a discrete-event simulation and in a time-stepped fashion. Given the rate of 1, the per-step transition probability in the time-stepped variant with step size τ is computed via the cumulative distribution function of the exponential distribution as $1 - e^{-\tau}$. Figure 3.5 illustrates an intermediate state of an example run in which half of the cells are still alive.

Figure 3.6 shows the evolution of the ratio of dead cells across simulation time. The data was generated by executing 10^6 simulation runs for the continuous reference and each step size. The average simulation end times and their 99% confidence intervals were 17.5±0.004 for the continuous reference, and 36.7±0.005, 22.0±0.004, 18.6 ± 0.004 for the step sizes of 0.5, 0.125, and 0.03125.

These results indicate that, using the time-stepped approach, the mean simulation end time always exceeds the reference, reflecting the cumulative delay introduced by the time steps. Even with the smallest time step size of 0.03125, the

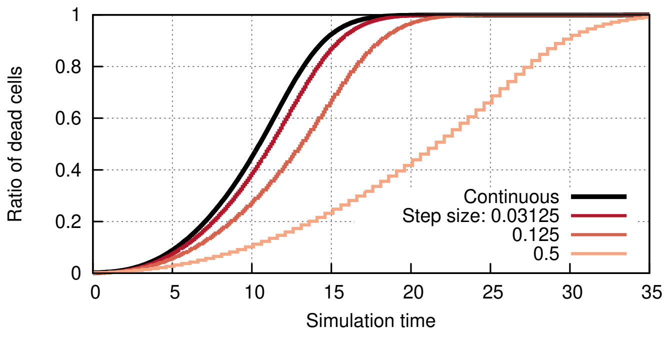


Figure 3.6. Ratio of dead cells over time, showing the time-stepped execution deviation from the continuous reference.

deviation was still 6%. Thus, these findings demonstrate the significant deviation that a time-stepped simulation can introduce when compared to a continuous-time reference simulation. A more thorough analysis of the effects of discretization in time on continuous-time simulations is given in [31]. However, our discussion clearly highlights the effects that risk management, according to our new definition, can have on simulation results. As we will show later, these effects also directly impact simulation performance.

3.2.1 Simulation Algorithms for Epidemic Models

In this section, we present the internal dynamics of the simulation techniques being evaluated in the current case study.

Asynchronous Execution using Time Warp

Simulating large models with tightly-coupled agents on Time Warp typically entails two main classes of events: i) ones implementing the *agent-based logic*, and ii) *artificial events* only required to transition the agent-based model to a Time Warp-based simulation. The main difference between these two classes of events is their granularity. Indeed, it can be expected that the first-class events are coarse-grained because they will implement the interactions among the agents. Conversely, the second-class events are likely fine-grained to the extent that they might only entail reading a single state variable. This aspect is critical for Time Warp simulations, as fine-grained events are likely not paying off the synchronization overhead. Additionally, an interleave of fine- and coarse-grained events could produce a non-negligible skew in simulation time among simulation objects, likely increasing the rollback probability and reducing the simulation's efficiency [36].

The second issue with Time Warp-based simulations directly relates to the high coupling between agents. Classical Time Warp simulations have been shown to be efficient [36] when the runtime environment can capture some degree of parallelism among the different LPs. This is particularly true when only some portions of the model require sharing timestamped information in a particular simulation time window. In the case of tightly-coupled agent-based models, this property rarely holds: the different agents have to continuously access their respective states, e.g. to determine how to interact with each other or collaboratively with the environment. As mentioned, this kind of access typically employs couples of "request”/ "reply" events. The volume of these events could easily make a parallel/distributed simulation based on Time Warp synchronization thrash.

The third issue is still related to "artificial" events to access other agents’ data. As mentioned, these events are simultaneous because they do not produce any actual advancement in simulation time. Nevertheless, if more than one agent is involved in these data exchanges, a cycle of simultaneous events could materialize, a condition hampering the liveness properties of Time Warp-based simulations [61].

There are two primary approaches to address these three main obstacles to an efficient Time Warp-based simulation. The first strategy is related to exploiting regions of interactions among agents. As it has been shown in the literature (see, e.g., [88]), if the goal of an agent-based model is to study emergent behaviour, a good grade of approximation for parallel/distributed simulations is to partition the model into "physical regions". Then, it is possible to differentiate the modelling strategy among inter-region and intra-region interactions, possibly reducing the amount of cross-region interactions. This is especially true if the degree of coupling across regions is low. For the Time Warp-based approach, we map regions to LPs. Within a region, agents can freely interact because we are guaranteed by the synchronization algorithm that all agents will observe the very same simulation time upon each event’s execution.

Conversely, agents belonging to different regions must interact via message passing. Our approximated Time-Warp model allows agents to pick neighbours only within the same region (i.e., the same LP). Conversely, cross-region interactions are modelled by relying on agent migrations: once an agent migrates to a different region, it will build its adjacency graph and start computing infections based on the states of neighbouring agents.

While this solution significantly reduces the number of "artificial" events injected into the system, it does not solve the management of potential transitions. As a candidate solution, an agent could maintain the timestamp of the subsequent reaction within its state. A new discrete event could be injected into the system at the same timestamp to mark a candidate's position upon the simulation timeline for the transition. Upon executing these tentative events, the model could quickly check the involved agent's state to determine whether the transition has to occur or has been overridden by a different transition time. While this approach is quite simple from a modelling perspective, it would inject a large number of highly fine-grained events that would unacceptably hamper the simulation's performance.

Conveniently, this happens to be another use case for *retractable events*[116]. We introduced them earlier to handle future spikes in spiking neural networks and we briefly discussed them in Section 3.1.2.

Synchronous Execution using S^{3}A

Synchronous Speculative Stochastic Agents (S^{3}A) is an optimistic algorithm tailored to the challenges of parallelizing the execution of SSA-driven agent-based simulations [3]. Taking inspiration from the classic Breathing Time Buckets algorithm [147], this synchronous algorithm proceeds in a window-based manner, allowing each agent to advance at most by one transition per window. Having this restriction eliminates the need to maintain a history of the agent's state beyond a single old and new state. Further, the algorithm is able to limit the scheduling overhead by allowing agents to access each other's state directly without mediation in the form of explicit events.

While it has been shown that S^{3}A could substantially accelerate simulations of a large-scale epidemic model with highly dynamic topology, its underlying assumptions make it hard to benefit from the locality in the agent interactions. This is clear in scenarios where agents are segregated, with each agent's interactions limited to its current compartment [22]. If communication is sufficiently local and different model portions are thus sufficiently decoupled, the asynchronous execution of Time Warp may better exploit the model's inherent concurrency.

The S^{3}A algorithm operates under the assumption that sequences of transitions may swiftly and unpredictably affect arbitrary agent states throughout the simulation. To account for this assumption, the algorithm maintains a tight coupling among processors to limit the frequency and cost of rollbacks. This is accomplished through a round-based synchronous approach using global barrier synchronizations. Since the tight temporal coupling among processors may severely limit the exploitation of a model's concurrency, each round must be as inexpensive as possible. The overhead of optimistic algorithms lies in the management of the lists of events, previous states, and antimessages and the cost of rollbacks. In S^3A, state saving is limited to a single old and current state per agent, similar to synchronous update schemes for time-stepped cellular automata [151, 127]. Due to the fine-grained state saving on the level of individual agents, a rollback involves only the inexpensive operation of copying the old state to the current state. Further, the algorithm avoids the need for antimessages by ruling out transitive causality violations altogether.

We describe the high-level operation of the S^3A algorithm based on the pseudocode shown in Algorithm 2. For a more in-depth description, see [3]. Agents are initially distributed to a set of processors. In each round of the algorithm, the processors execute local transitions in non-decreasing timestamp order up to a bound determined as the sum of the global minimum timestamp among all scheduled transitions and a tunable initial window size τ_0. Each transition may involve immediate read/write accesses to other agents. The key idea of the algorithm is to dynamically reduce the window size so that at the end of the round, the window contains only those transitions and accesses that can safely be committed. This is accomplished by guaranteeing that the final window contains the earliest transition or access for each agent according to their timestamps, if any. By definition, these transitions or accesses can never be displaced by another transition or access.

Algorithm 3 shows the dynamic adaptation of the window size during agent access as part of a transition. The variable `earliest_access_ts`, initially set to ∞, records the earliest timestamp of access to the agent. If new access arrives with a timestamp earlier than `earliest_access_ts`, the new access can be carried out, displacing any previously recorded access with larger timestamps by immediately rolling back the agent to its old state. On the other hand, if an earlier access has been recorded previously, the window size is reduced to exclude the current access and its associated transition from the current round. Using a global window rules

Algorithm 2 Main loop of the S^3A algorithm.

```
 1: while !termination_criterion do
 2:     global_bound ← get_global_min_ts() + $\tau_0$
 3:     for each thread in parallel do
 4:         execute transitions earlier than global_bound, dynamically reducing
 5:         global_bound according to Algorithm 3
 6:     end for
 7:     barrier()
 8:     for each agent in active_agents ∪ accessed_agents do
 9:         if agent.min_access_ts < global_bound then
10:             commit transition, enqueue new events
11:         else
12:             roll agent back to the previous state
13:         end if
14:     end for
15:     barrier()
16: end while
```

Algorithm 3 Agent state access in S^3A.

```
 1: procedure AGENT::ACCESS(access_ts)
 2:     agent.lock()
 3:     if access_ts < agent.min_access_ts then    ▷ access is earliest in round so far
 4:         if agent.min_access_ts ≠ ∞ then
 5:             roll back agent
 6:             global_bound ← min(global_bound, agent.min_access_ts)
 7:         end if
 8:         carry out agent state access
 9:         agent.min_access_timestamp ← access_ts
10:     else                                       ▷ access is deferred to a subsequent round
11:         global_bound ← min(global_bound, access_ts)
12:     end if
        agent.unlock()
13: end procedure
```

out transitive effects of displaced or deferred accesses since any previously computed transitions and accesses with timestamps above the window bound are rolled back at the end of the round (Algorithm 2, line 10). Similarly, any newly scheduled transition within a round is deferred to a subsequent round by reducing the upper window bound to its timestamp.

3.2.2 Benchmark Model

We compare the asynchronous and synchronous parallel simulation algorithms using a simulation model of the epidemic spread of a contagion. The model is an extension of the agent-based formulation [77] of the classical susceptible-infected-recovered model. In our model, each agent is situated in one of a configurable number of regions, each initially populated with the same number of agents. Each agent has 8 neighbours chosen uniformly at random within the same region. Hence, the number of regions determines the degree of locality in the agent interactions.

For susceptible agents, the infection rate equals the number of infected neighbours. Hence, agents entering or leaving the infected state must notify their neighbours so their transition to the infected state can be rescheduled according to the changed rate. The transitions to the recovered state and back to the susceptible state occur with constant rates of 1. Two additional transitions introduce dynamic changes to the topology defined by the agents' neighbourhood relations. The first type of transition changes an agent's neighbours within its current region uniformly at random, potentially changing its infection rate or the neighbours' infection rates in the process. The second transition type migrates an agent to another region chosen uniformly at random and links the agent to new neighbours in the selected region. The rates at which these two types of transitions take place allow us to control the degree of computational load and agent interaction within each region on the one hand and the interdependence of transitions across regions on the other hand. Overall, this system resembles epidemic models as used in real-world epidemics studies [49], which aim to capture the effects of the populations' everyday as well as long-distance mobility.

3.2.3 Experimental Comparison of the Approaches

The implementations of the benchmark models discussed above are obtained from the work in [2].

Experimental Setup

We have run our reference implementations on a dual-socket machine, with each socket equipped with an Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz. The total amount of RAM is 256 GB.

We have considered two different values for migration rates, namely 0.16 and

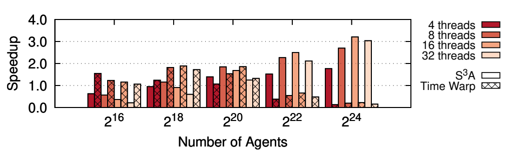
0.01, mimicking highly-variable and more stable scenarios, respectively. As for the number of regions considered in our runs, we have considered 128 and 8192 regions, which account for macro and meso-simulations, respectively. The total number of agents has been varied between 2^{16} and 2^{24}, accounting for medium and large simulation scenarios, evenly distributed across the available regions at simulation startup. We have explicitly discarded all configurations providing less than 128 agents per region at the beginning of the simulations.

In the following, we provide data related to the overall performance of the simulation runs employing both the S^{3}A and the Time Warp algorithms and profiling data to highlight the reasons behind the performance data. All results are averaged over three different runs.

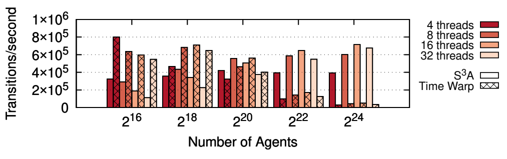
Because we employ entirely separate implementations of Time Warp and S^{3}A, we have identified metrics that allow a fair comparison. We focus on two principal metrics to conduct a performance comparison: the *speedup* over a sequential reference baseline (once again, see [2] for the details) and the *throughput* expressed in terms of committed transitions per second of wall-clock time. We consider only the actual simulation processing time for both metrics, i.e. we do not consider the initial model setup and final cleanup phases. Both the S^{3}A and Time Warp implementations rely on several data structures and subsystems that might require non-negligible time before starting the simulation, but optimizing these phases is beyond the goals of this study.

To study to what degree the performance differences observed in the measurements are fundamental to the synchronization algorithm itself or dependent on implementation-specific overheads, we also instrumented the simulators to isolate the costs of the main tasks. The profiling results shed light on the differences in overheads and enable us to identify avenues for future performance improvements.

The selected profiling metrics are *forward execution time* and *event management time*. The former accounts for the total time spent by both simulators running the model's code, i.e. not considering any housekeeping operation. The latter considers all the housekeeping operations related to event management, such as enqueuing new events, extracting next events, and executing rollbacks.

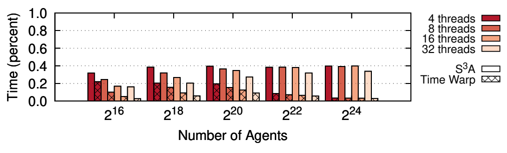


(a) Speedup.

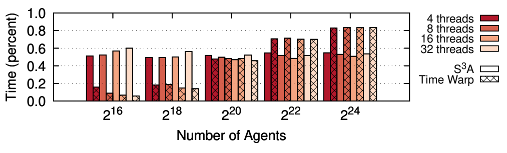


(b) Transitions/sec.

Figure 3.7. Performance Results (Migration Rate: 0.01—Regions: 128).



(a) Time Spent in Forward Execution.



(b) Time Spent in Event Management.

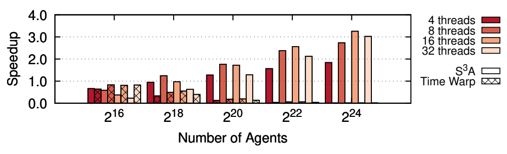
Figure 3.8. Profiling Results (Migration Rate: 0.01—Regions: 128).

For the Time Warp implementation, we also provide classical measures, namely the *efficiency* (in terms of committed events/executed events) and the *rollback length* (i.e., the average number of events that are undone every time that a straggler message is received and a rollback operation is carried out).

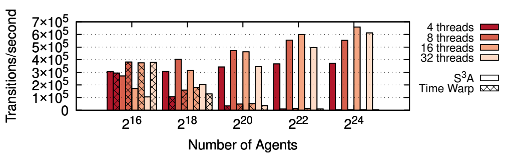
Experimental Results

In configurations with 128 regions, the Time Warp implementation generally delivers poor speedup values. The migration rate and the number of agents play an essential role, anyhow: with higher migration rates (Figure 3.9), the slowdown is generally more apparent than with lower rates (Figure 3.7). Similarly, smaller agent counts deliver better speedups.

Overall, this behaviour is mainly due to the number of agent migrations being high compared to the number of regions. Migrating frequently causes many expensive rollbacks: as shown in Figure 3.15, efficiency decreases when the agent count

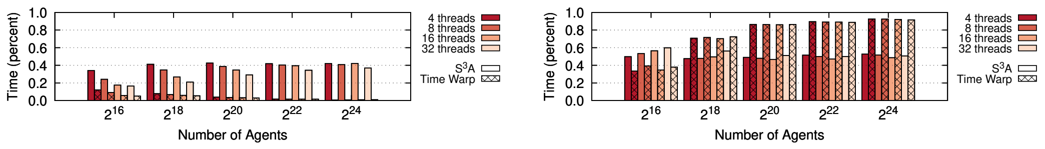


(a) Speedup.



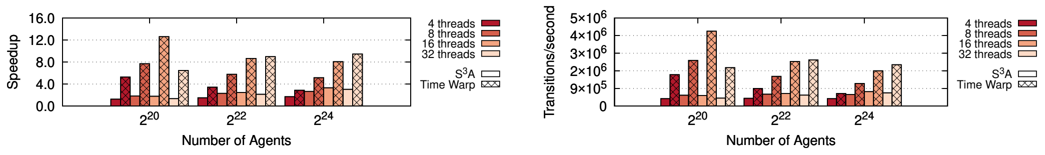
(b) Transitions/sec.

Figure 3.9. Performance Results (Migration Rate: 0.16—Regions: 128).



(a) Time Spent in Forward Execution. **(b)** Time Spent in Event Management.

Figure 3.10. Profiling Results (Migration Rate: 0.16—Regions: 128).



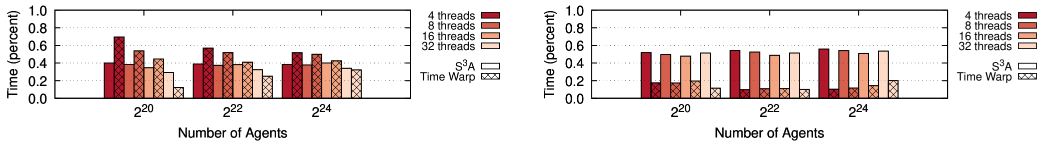
(a) Speedup. **(b)** Transitions/sec.

Figure 3.11. Performance Results (Migration Rate: 0.01—Regions: 8192).

increases. In configurations with 32 threads, efficiency is as low as 20%. At the same time, the rollback length can get as high as 350 undone events per rollback, thus wasting a non-minimal amount of work done in forward execution.

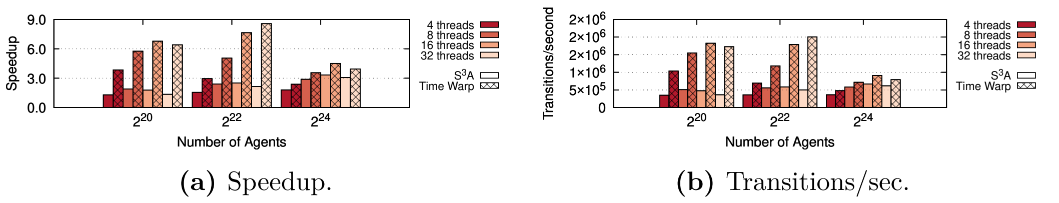
We note that the rollback length directly depends on the checkpoint frequency, which is controlled by an autonomic agent in ROOT-Sim [104]. This agent takes into account several internal parameters to tune the checkpoint interval. The relevant ones for this particular model are the event granularity, which is fine-grained on average, and the size of the LP state, which grows with the agent count. Therefore, the autonomic checkpoint strategy is selecting a large checkpointing interval in an attempt to spend more time in forward processing, trying to favour fine-grained events against costly checkpoints. Conversely, the high number of migrations lowers efficiency due to increased straggler messages being generated: going from 99% to 98% in efficiency translates to a doubled rate of rollbacked messages.

Overall, rollbacks are expensive in these configurations exhibiting few and large regions with many regions interactions. As shown in Figures 3.7 and 3.9, the per-



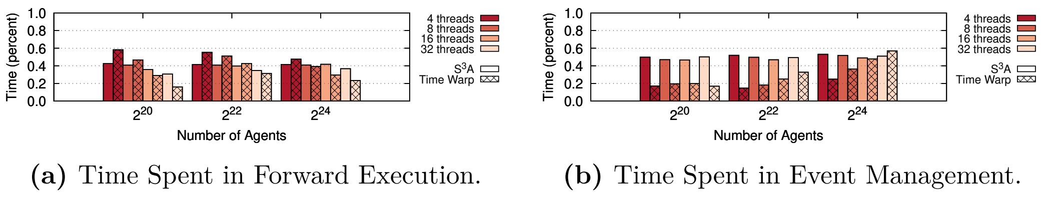
(a) Time Spent in Forward Execution. **(b)** Time Spent in Event Management.

Figure 3.12. Profiling Results (Migration Rate: 0.01—Regions: 8192).



(a) Speedup. **(b)** Transitions/sec.

Figure 3.13. Performance Results (Migration Rate: 0.16—Regions: 8192).



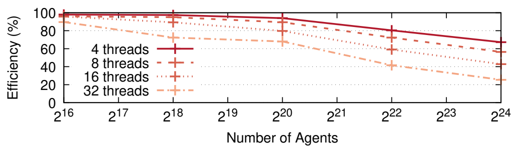
(a) Time Spent in Forward Execution. **(b)** Time Spent in Event Management.

Figure 3.14. Profiling Results (Migration Rate: 0.16—Regions: 8192).

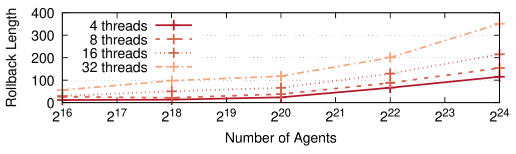
centage of time spent in event management (that also accounts for rollbacks) is significantly high. Rollbacks further disrupt execution by indirectly slowing down other LPs, which would be more likely to generate more straggler events. Time Warp is effectively trashing in this worst-case scenario: the portion of time spent running model code is abysmal. As again shown in Figures 3.7 and 3.9, the percentage of time spent running events is minimal.

Conversely, S^{3}A provides acceptable speedups. The overhead of synchronous rounds and contention on atomic operations with lower agent counts becomes apparent. For example, in Figure 3.7aa, with 2^{16} agents, speedups worsen by increasing the thread count. With 32 threads, each only processes a maximum of $\sim$ 2000 agent updates per round. The results, anyhow, show good scaling in the number of agents, suggesting that S^{3}A performance could grow with larger agents count on larger multicore CPUs. This trend is endorsed in Figure 3.7b, with transitions throughput increasing with both the count of agents and cores. Increasing the migration rates does not impact the speedup results (see Figure 3.9) while transition throughput improves. For S^{3}A, the results highlight that migrations are less computationally demanding, as expected.

With higher region counts, namely 8192, the runtime behaviour of S^{3}A is mostly unchanged—see Figures 3.11 and 3.13. Even the profiling results, shown in Figures 3.12 and 3.14, are almost identical. This result indicates the stability of S^{3}A



(a) Efficiency.



(b) Rollback Length.

Figure 3.15. Time Warp Performance Metrics (Migration Rate: 0.01—Regions: 128).

towards the configuration parameters of the model.

Instead, Time Warp sees a substantial improvement in performance, delivering competitive speedups and transition throughputs. Nevertheless, as shown in Figure 3.11, the trend seems to worsen with the increase in agent count. We conjecture that a too-low agent density leads regions to engage in long incorrect speculative trajectories, leading to infrequent but expensive cascading rollbacks. On the other hand, a too-high agent density gets closer to the pathological scenario shown earlier with 128 regions.

With the migration rate set at 0.16, the best performance spot moves to a higher agent count, as shown in Figure 3.13. Overall, speedups are lower, but these results suggest a bell-shaped speedup behaviour for Time Warp simulations. Interestingly, S^3A performs similarly to Time Warp in the densest configuration. The safer and controlled round-based optimistic execution of S^3A pays off when a misspeculation in Time Warp is either too costly (dense regions states/expensive rollbacks) or frequent (high migration rates/many straggler messages).

3.3 Discussion

The case study presented in this Chapter has considered two very different families of simulation models (SNNs and agent-based epidemics) executed using different synchronization algorithms that exemplify multiple choices in the continuum previously depicted in Figure 1.2.

The results clearly highlight that the factors that affect the performance and accuracy of the results highly depend on the simulated models' inner dynamics. At the same time, the actual lower-level implementation of particular mechanisms provided by the simulation runtime environment is key to identifying well-balanced solutions to exploit risk to improve accuracy and performance.

The results for the SNN simulation have shown that if the high resolution results are pursued, traditional time-stepped simulation algorithms cannot provide results promptly. In contrast, Time-Warp based ones exhibit only a reduced performance penalty. At the same time, with lower time-step values, time-stepped simulation outperforms the speculative PDES' one, but, interestingly enough, the accuracy of time-stepped algorithms still appears to be considerably lower.

Overall, in the case of SNNs, we have shown that the retractability of events alone can allow the runtime environment to improve both performance and accuracy. In the case of epidemic models, depending on their specific configuration, this capability alone may not be enough.

The contrasting assumptions made by the two algorithms used in the latter family of models are the underlying reason for the divergent outcomes. S^3A adopts a strict approach, reducing the local window size and retracting transitions and agent accesses to avoid transitive errors. This approach is appropriate when inter-agent communication is unpredictable, global, and occurs with only small delays in simulation time.

In contrast, Time Warp allows relatively coarse-grained LPs to advance asynchronously in time, which is suitable when the degree of coupling among LPs is low. Our measurement results have demonstrated that Time Warp performs better when there is a high degree of locality in inter-agent communication, breaking the model into loosely coupled LPs. Conversely, S^3A is superior for model configurations with predominantly global communication.

Outside of the two extremes of global and localized communication within small regions, neither of the approaches fully aligns with the properties of the considered model. If there is only a modest number of regions, Time Warp must either divide each tightly coupled region into multiple LPs acting asynchronously or fall back to sequential execution per region. On the other hand, the synchronous execution of S^3A fails to exploit the locality in the inter-agent communication.

To recap, the following conclusions can be drawn from the experimental study presented in this Chapter. For an implementation of a simulation runtime environment to be effective in enabling the modeller to obtain significantly accurate results, it must be able to capture the model's dynamics as much as possible and adapt to the model's peculiar characteristics. Similarly, such a runtime environment must provide highly-optimized support to prevent particular model dynamics from significantly affecting the dynamics of the dynamically adopted synchronization algorithm. This means that it is necessary to push simulation methodologies and techniques to a higher level of adaptability than what has been done so far in the literature, allowing for a dynamic choice between specific execution modes and supports depending on the current model dynamics. In addition, it is clear that depending on the accuracy a modeller expects from the results, different choices can be made, perhaps aiming to improve performance.

The exploration of risk-related possibilities, as we have defined it in this thesis, thus becomes much more multivariate. In the remainder of this thesis, we will illustrate various methodologies, techniques and implementations that allow us to explore these facets of risk, playing with precision in results and performance improvements.

Chapter 4

Literature Survey

The literature offers a wealth of work attempting to control or exploit risk in speculative simulations, according to the general definition provided in this thesis. Conducting an exhaustive survey of all the different proposals and methodological nuances that can be found is definitely beyond the scope of this work. However, we will try to illustrate which main methodologies have been studied in the literature, sometimes without considering their relation with the concept of risk, and show how they fit in effectively with our new interpretation of this concept. For a more comprehensive and thorough discussion of the possible techniques, we refer the reader to [35, 38].

In the discussion, we try to group the proposals into homogeneous sets. However, some of these works often have points of contact with more than one categorisation chosen for this Chapter.

4.1 Optimism Constraining

The first form of risk management arose from the observation that Time Warp performance degrades quickly if the probability of rollback in the execution of a model increases. The incidence of this phenomenon is greater the more skewed the clocks of the individual LPs that make up the simulation.

The LP clock skew directly affects memory consumption as the speculative portion of the simulation trajectory becomes larger. For this reason, in the early years of PDES, when computing resources were scarce, over-optimism could also lead to situations in which a given simulation could become intractable, as the operating system halts process execution due to memory exhaustion. Thus, an early form of optimism control was already identified by Jefferson [60], with the introduction of the *cancelback* protocol. The cancelback protocol involves the generation of artificial rollbacks by the runtime environment to force the retrieval of memory buffers tied to portions of the speculative trajectory (e.g., buffers used for saving states) of those LPs that are too far ahead of the commit horizon (or rather, ahead of the current estimate of the GVT). Indirectly, this protocol tends to contain the risk associated with over-optimism since it prevents the possibility of simulation crashes.

Along the lines of this protocol, the idea of execution throttling was later presented [154, 64]. The aim of these techniques is not so much to prevent risk effects from preventing simulation termination, but rather to avoid performance degradation due to an excessive amount of wasted work. The idea behind these proposals was to suspend the execution of that subset of LPs that strayed too far from the commit horizon. In this way, clocks tend to realign, and the probability of rollbacks decreases, as the probability of an LP being hit by a straggler drops. This is all the more true the larger the model lookahead.

Elastic Time [146] tries to limit the aggressiveness of optimistic synchronisation by noting "pulling back" LPs that got too far from the commit horizon, as constrained by an elastic. This simple throttling approach implements a Near Perfect State Information system, i.e. a situation where all the LPs are not "too far" from the minimum, although there is no requirement for a significant synchronisation effort.

Bounded Time Warp [158] and *window-based throttling* [131] limit optimistic processing by defining a static optimism window. The main problem with these proposals is the possibility that different simulation models (or even individual entities in the same model) may require different window sizes for optimal performance. An incorrectly sized window can drastically reduce performance. *Adaptive Time Warp* [6] controls optimism by forcing a simulation entity that experiences many rollbacks to freeze for a time BW. Determining BW is complex, often leading to configurations that may not maximise speedup.

Furthermore, as discussed in [158, 131], the value of BW may be different for each simulation entity. *Penalty Based Throttling* [131] activates the entities that receive the fewest antimessages, associating them with a penalty that captures the number of antimessages received. In a complex simulation, all simulation entities can have a non-negative penalty at a particular time instant, leading to an excessive reduction in aggressiveness.

Throttling techniques have also been effectively used to support the Next Subvolume Method [63, 25, 166], which is directly related to the use case we have described in Section 3.2.

4.2 LP Scheduling

The work on throttling has connections with several works related to LP scheduling. Indeed, suspending the execution of a group of LPs is tantamount to saying that the scheduler is aware that its choices are related to risk management. Indeed, the choice of the next event to be executed directly connects with the risk incurred by the simulation. The scheduler of the runtime environment is, in fact, the privileged component for managing and exploiting risk. Therefore, more or less explicitly, we find in the literature many scheduling algorithms that improve performance by employing event selection that considers their risk. This section presents some of the key works on this topic. A more extensive discussion, albeit related to a less developed concept of risk, can be found in [126].

The concept of risk-based scheduling is already present in the seminal paper [73] introducing the lowest-timestamp first (LFT) scheduler. Indeed, the basis of this work is the concept that, in a concurrent simulation, it is possible to eliminate the risk local to a processor by selecting for execution, among all the LPs currently being handled by the processor, the one that has the next event temporally closest to the last event processed. This way, LP clocks are only locally aligned, and risk can only come from remote processors.

After this work, many proposals have dealt with heuristics to schedule entities based on their rollback behaviour and productive work. *Useful work* [97] is a performance index based on control theory that enables scheduling policies to control the optimism of a time warp simulation, reduce the rollback frequency, and reduce memory usage and wasted lookahead computation as a secondary effect. These are all facets of the risk concept introduced in Section 2.3.

Aggressiveness/Risk Effects-based Scheduling (ARES) [20] controls optimism by scheduling with a higher priority the simulation entities whose next event has a lower probability of being eventually rolled back. This is done by selecting a set of candidate simulation entities with a low probability of being ultimately undone; then, it chooses an LP among the candidates to minimise the number of new events to be notified.

Share-Everything PDES [56] exploits a short-framed temporary binding between simulation entities and worker threads. The proposal uses a per-node global future event set that can be concurrently exploited to determine the next event closest to the commit horizon—This is another aspect that we will deal with more explicitly in Chapter 7. This approach can reduce the incidence of rollbacks and the generation of stragglers thanks to its controlled aggressiveness.

The body of work on throttling discussed in Section 4.1 also directly relates to LP scheduling. Indeed, some of these works were based on introducing the concept of *windows*. These windows can capture the clock skew, triggering the throttling of too-far-away LPs. Nevertheless, these windows can also be used to decide which LPs to schedule, or how to schedule them. *Breathing Time Warp* [149] combines *Breathing Time Buckets* [148] and Time Warp. While Breathing Time Buckets only allows events to be sent when they are valid (thus with very low aggressiveness), Breathing Time Warp is based on the principle that events closer to the GVT have a lower probability of being cancelled. Therefore, N_1 events close to the GVT are optimistically executed, and N_2 events after that point are executed using *Breathing Time Buckets*. Determining N_1 and N_2, as well as dynamically modifying their values to minimise execution time, are aspects not considered in this work.

Adaptive Bounded Time Windows [95] uses the concept of *useful work*. Windows are sized to maximise speed. *Adaptive Time-Ceiling* [82] is based on a similar concept, although window sizes are chosen from a set of discrete values. The hybrid synchronisation algorithm proposed in the considered paper is related to Breathing Time Warp in its combination of a window-based optimistic synchronisation algorithm with Time Warp.

A fundamental limitation of the proposals mentioned above is that they generally consider uniform simulation models in which the event granularity is a variable that can be ignored. The adaptiveness in this class of approaches is based on dynamically recalculating the length of a simulated time window to include any event suitable for scheduling. However, if the grain of events has a significant variance, all the strategies discussed may lead to suboptimal results. Indeed, selecting for execution an event with an extremely large grain can lead to more work being wasted if the reception of a straggler cancels it out.

This problem was addressed in [125], where a grain-sensitive scheduling algorithm is proposed. The proposed algorithm algorithm schedules larger-grain events with a reduced level of optimism, which effectively reduces the *risk* of wasting substantial CPU power in the event of a rollback.

4.3 Future Event Sets

If the scheduler is a fundamental component of risk management, the data structures supporting its execution are equally as important. Indeed, if the LTF scheduler can manage risk locally, the time required for this management may still generate an imbalance. For instance, the proposal in [73] requires $O(n)$ time (where n is the number of LPs bound to a specific processor) to select the next candidate event for execution. The computational cost of finding the next event falls on the critical path of the simulation execution, so different data structures may have different effects on the clock skew of LPs, with respect to the global simulation.

Many data structures have been explored to reduce the next-event query time—some of them offer (amortised) $O(1)$ access time. Beyond the classical linked list or min-heap, we find the calendar queue [10], the splay tree [144], or the ladder queue [152, 50].

The work in [123] presents a Low-Overhead Constant-Time (LOCT) scheduler that leverages tree-like bitmaps to retrieve scheduled events rapidly. Experiments on multithreaded, shared memory architecture demonstrate that the LOCT scheduler outperforms the ladder queue.

The Non-Blocking Priority Queue in [78, 79] is a data structure with constant time performance that resembles Calendar Queues. This data structure has been shown to offer resiliency to conflicts [80], thus reducing the cost of concurrent event extractions.

In their optimistic parallel simulator [81], Hay and Wilsey [52] investigate the efficacy of leveraging hardware-based transaction memory (TSX) to manage pending events. Their research utilising a multi-set data structure reveals that Hardware Lock Elision beats traditional locking techniques by up to 27%.

As can be seen, the topic of pending-event sets data structure is hot. For a more thorough comparison and discussion of the performance implications of the different data structures, we refer the reader to [27, 34, 128, 136].

4.4 Load Balancing

The concept of risk is also related to works dealing with *load sharing* and *load balancing*, such as [8, 46, 82, 95, 105, 162].

In general, these proposals aim to shift the workload associated with individual LPs between the various processors, be they nodes of a distributed simulation or the worker threads of a parallel simulation. The stated objective of these proposals is to improve performance, but in fact, what they do is reduce risk, according to our new definition. Indeed, by migrating workload, the aim is always to avoid portions of useful work being replaced by potentially unfruitful work.

Overall, constraining the effects of risk can reduce the number of rollbacks, thanks to improved exploitation of local synchronisation based on smallest-timestamp first scheduling.

An interesting work along the line of load balancing is [64]. Here, the authors study a combination of LP migration and throttling to combat the over-optimism of Time Warp. The most exciting part of this work is that the authors recognise that throttling and migrations can be effective only if managed together. Their proper combination can avoid side effects—wasted processor cycles for pure throttling and network over-utilisation for migrations. In some sense, this work had already glimpsed that risk is multifaceted concept that should be handled holistically to provide effective simulation environments.

4.5 Hybrid Synchronisation

In some works, hybrid synchronisation approaches have been proposed that add optimism to conservative algorithms or slow down optimistic simulations (for a thorough discussion on the topic, we refer the reader to [24]). SRADS with local rollback [26] and speculative computing [86] optimistically process unsafe events locally, thus confining rollbacks to local entities. Breathing Time Buckets [148] is similar to SRADS but can dynamically vary the conservative time windows based on global and local event horizons. This technique performs poorly under small lookahead conditions, and the GVT must be computed with no in-transit messages.

Bounded lag restriction [76] uses the measure of the minimum distance between entities to decide safe events based on programmer-provided apriori lower bounds between causally related events. The *rollback relaxation* [171] and *unsynchronised parallel simulation* techniques [129] relax causality constraints for memoryless logical processes or completely ignore causality violations for queueing models to reduce the rollback overhead. The final results may be imprecise; in general, these techniques cannot be applied to all classes of simulation models.

In the recent *Virtual Time III* [61] a unification framework between conservative and optimistic synchronisation is proposed. This framework considers conservative algorithms as *accelerators* to Time Warp, given the non-existent overhead introduced to forward execution. Therefore, Virtual Time III enables some simulation entities to execute conservatively while others execute optimistically simultaneously. In this context, some entities are subject to throttling due to the execution in conservative mode.

All in all, these works recognise, more or less implicitly, that the simulation spectrum we previously depicted in Figure 1.2 is apparent and should be leveraged to improve the effectiveness of a simulation runtime environment. This is one of the major contributions of this thesis, which we shall describe in Chapter 5.

4.6 Autonomic Optimisation

The careful reader will have already realised that our view of risk handling is strongly associated with the concept of autonomic computing [66]. Indeed, providing the model with competitive runtime support is impossible without self-optimisation. Interestingly, the PDES literature is replete with self-optimisation protocols, although many of these optimisations focus on very specific aspects of execution dynamics.

One area that has seen a significant proliferation of autonomic self-optimisation models is state saving. Implicitly, these approaches have been aimed at reducing the risk that the overall performance would be reduced by increasing the amount of non-useful work on the critical path of the simulation. Explicitly, they have tried to fine-tune the value of the checkpointing interval depending on the actual execution dynamics of the simulation model.

The approach described in [96] selects the best checkpointing interval by relying on an analytic model based on LP execution time. The underlying assumptions are that the execution of events is *non-preemptive* and that the rollback length is independent of each other. The assumptions of this work can be regarded as weak, especially in the framework of risk we are discussing. Indeed, the non-preemptability of events could be itself a source of risk, because due to transient errors in the speculative trajectory [93] the models' execution might get stuck in a livelock. The case study discussed in Section 3.1 has highlighted that the network topology may bias the rollback length for highly-coupled models.

Under the same assumptions, the work in [135] proposes to observe, in a wall-clock time(WCT) interval, the number of rollback operations and the number of executed events (both committed and uncommitted). A numerical sequence of checkpointing intervals is generated based on these parameters, upon which the optimal checkpointing interval is selected. This scheme does not consider that the execution time of different typologies of events can vary.

This aspect is captured in [143], where the *Event Sensitive State Saving* technique is proposed. This technique emphasises that it is convenient to take a state snapshot when the *granularity* of the next event increases. Then, building on the model in [135], a proper optimal checkpointing interval is selected depending on the most-occurring class of events. This approach, therefore, tries to reduce the coasting-forward time by avoiding reprocessing chains of events containing ones that require a high amount of WCT to be reprocessed.

A different approach is presented in [33], which regulates the checkpointing interval using a heuristic algorithm based on the periodic re-calculation of a cost function accounting for the average amount of wall-clock time to perform a state-saving operation and the cost of the coasting forward phase. This solution, therefore, explicitly adapts the checkpointing interval in the face of rollback frequency.

An additional approach in [121] proposes to observe LPs' event history, considering the variations between the timestamp of two consecutive events, to determine the best moment for taking a snapshot. The approach considers the interval between events in simulation time to determine the most likely point a straggler could hit.

In [122], a cost model is proposed to select the checkpointing position in an optimised way. It is based on a heuristic which tries to minimise the rollback length: the system decides to pay the cost of a checkpoint at a certain simulation-time instant only if the estimation of its possible (future) restore cost is higher.

The work in [104] proposes a fluctuation-resilient approach to both fine-tune the checkpointing interval and select whether it is more convenient to rely on incremental or full checkpointing. Once again, the goal is to reduce the amount of unfruitful work on the critical path, because it directly increases the risk of reducing the overall performance, due to increased rollbacks. We base the autonomic checkpointing strategy used in the experiments of this thesis on this work.

Another decision model can be found in [18]. Here the focus, similarly to [104], is on the mode used to restore a previous consistent state. In particular, the work in [18] mixes state restore based on reverse computation [15] and checkpointing [59].

We discussed the autonomic optimisation of state saving as an emblematic example of the need for autonomic self-optimisation methods for risk management, since we will deal with related issues in Chapter 6. However, the needs for self-optimisation in PDES are more wide-ranging.

4.7 Event Ties

A problematic aspect related to risk introduced in Section 2.2.2 is related to *simultaneous events*, i.e. those scenarios where the runtime environment has to decide upon what is the next event to schedule, from a pool of simultaneous events.

Handling simultaneous events is an essential topic for DES that has observed much attention from the community. Interestingly, in the seminal contribution in [72], the problem of simultaneous events is not considered from the point of view of models that may suffer from non-commutativity in updating states. In fact, concerning simultaneous events, Lamport merely states that *two contemporaneous events do not impact on mutual causality.* However, while not creating causality problems, two simultaneous events may lead to correctness issues with respect to model characteristics, possibly leading to errors in results or crashes in the simulation [93].

In *Virtual Time* [59], the problem of simultaneous events is already apparent in the context of optimistic simulation. Indeed, it is shown how, for concurrency control simulations in distributed databases, it is possible to use the event source to resolve ties between events.

The work in [142] has dealt with simultaneous events demanding additional bits of information from the programmer to break ties. In particular, the authors have replaced the traditional `double` datatype used in ROSS [12] to represent timestamps with a data structure that allows explicit control event ordering in the case of ties. This approach is similar to [137, 69], where a mechanism to sort simultaneous events based on an extension of the timestamp or user-defined priorities was proposed. Differently from these proposals, we can rely on the events' content, thus making the tie-break more transparent. To some extent, the modeller is allowed to specify a priority based on the used event type, although this is not necessary for our solution to work.

A slightly similar path is taken in [85], where the authors extend the concept of virtual time embedding multiple values to preserve determinism over multiple runs. Differently from [142], the authors of [85] also exploit controllable and deterministic random-number generators. Therefore, they also allow exploring multiple ordering of events, thus enabling broader statistical analysis of models. This approach was also envisaged in prior work such as [108, 109, 110], where the importance of studying the outcome of multiple ordering of simultaneous events was highlighted. This contribution is slightly different from the work in [168], where the author claims that it would be correct to present averaged results over all the possible orderings in case of simultaneous events.

In *Virtual Time III* [61], the modeler is required to provide a solution to ties explicitly. The event dispatcher always receives a *zSet* of events, therefore, the model must handle the zSet explicitly and process the events based on the model's logic.

Although this approach is not entirely transparent, it enables models to account for the cumulative effects of tied events, which would otherwise require saving the entire simulation trajectory within the model's state when ties are broken at the simulator level.

Breaking an event tie, when the model developer provides no additional information, is an operation that brings substantial risk. we will deal with this problem in Chapter 7.

4.8 Approximated Simulation Results

One of the effects of our definition of risk still uncovered is related to the possibility of diverging from the results observed in the corresponding sequential simulation. In the literature, we find solutions that explicitly leverage on this possibility to improve the overall simulation performance.

Some solutions are based on *uncertainty* in the occurrence of events [37, 124]. These proposals build a partial order of events that allows the exploitation of temporal uncertainty to reduce the rollback probability. Events are not associated with a single timestamp but rather with an interval. Overlapping events can be reordered to provide equivalent schedules to avoid executing a rollback operation upon receiving a straggler message. A similar objective is pursued through *symbolic execution* [165], in which a group of uncertain cases are jointly simulated in the same run to explore possible configuration parameter intervals. The final effect can be a bias in the evolution of the simulation model trajectory that can ultimately lead to an approximation of the collected statistics compared to a scenario where a complete specification is provided by the model.

Still related to uncertainty, some works [106, 175, 139] have addressed the support for discrete-event simulations under vague information. In particular, these works propose to approximate the simulation by relying on *fuzzy set theory* to handle subjectivity, vagueness or imprecision in estimating activity duration. Probability distributions are replaced or complemented with fuzzy numbers (typically triangular and trapezoidal) that allow for representing uncertainty in activity durations. Events are extracted from the input queue relying on a form of ranking measure [156] that accounts for the fuzziness of the event timestamps. These techniques typically provide approximated simulation results and require ad-hoc simulation support [106] because traditional discrete-event simulation approaches cannot cope with the specific kind of approximation offered by fuzzy sets.

Another form of approximation for stochastic simulations can be found in [159, 17], where the *standard clock* technique is proposed. It is a form of *uniformisation* that simplifies the execution of Markov and semi-Markov and renewal processes, providing a formalism for studying discrete-event simulations. The core idea considers that typical processes modelled in discrete-event simulations entail “beginning” and “end” events to describe the duration of an activity. The approach draws both events from a single distribution, and the “nature” of events is later determined by drawing samples from a uniform distribution. If an event is found unfeasible, it is discarded. Therefore, there is no longer the need for an actual event queue because it is sufficient to determine the next-event timestamp and decide what event it is. While the work in [17] has shown that the approach is suitable for many timestamp distributions, the overall results could diverge from a traditional discrete-event simulation, specifically if multiple random processes are simulated at once.

Other proposals have studied the tradeoff between relaxing strict causality of the events and its effects on performance in (speculative) PDES [129, 57]. Essentially, these solutions skip running some state rollbacks if the effects of processing events out of strict timestamp order are considered acceptable regarding the final statistics computed by running the simulation.

All these works suggest that if the modeller can be satisfied with imprecise results, the notion of risk can be leveraged to reduce the burden on the runtime environment to enforce strict control of the speculative simulation trajectory, and therefore deliver improved simulation performance. We will deal with such an aspect in Chapter 6.

Chapter 5

Hybrid PDES Synchronisation

As we mentioned earlier, the spectrum of possible synchronisation algorithms for PDES (see Figure 1.2) forms a continuum of multiple risk exploitation features. It is, therefore, apparent that going beyond the mere exploration of this spectrum requires cooperation between different algorithms.

This innovative approach stems from the evidence that models with different dynamics lead to different algorithms suffering on the spectrum [13, 2]. The possibility of implementing multiple algorithms within the same simulation environment, allowing them to be dynamically activated as a function of the model's dynamics, brings with it the benefit of avoiding the phenomena of performance collapse of synchronisation algorithms.

This formulation was first theorised in [61], where it is pointed out that it is desirable to be able to combine conservative and speculative synchronisation algorithms. In this way, it is possible to take the best of both worlds, thus improving performance precisely by leveraging the different risk taken by the various algorithmic strategies. To the best of our knowledge, what we propose in this chapter is the first real and practical implementation of what was theorised in [61].

In particular, we will show the strategies required to make different synchronisation algorithms work together. On this journey, we have selected two speculative algorithms that lie rather far apart in the spectrum of Figure 1.2. This hard choice requires proper expedients to allow the algorithms to coexist. While we have used only two, the integration methodology can easily be extended to more algorithms.

5.1 Selected Synchronisation Algorithms

The synchronisation algorithms we picked for integration are the classical Time Warp [59] and Window Racer [7]. Time Warp exhibits the highest degree of risk within the spectrum of optimistic synchronisation protocols.

Conversely, Window Racer is a recent synchronous optimistic synchronisation algorithm for shared-memory architectures. Inspired by Steinman's Breathing Time Buckets (BTB) [149], Window Racer alternates between an execute and commit phase. As in BTB, each worker thread is assigned a portion of the simulation entities. In both algorithms, at the end of the execute phase, a newly determined GVT decides which state transitions can be committed. However, important differences lie in the granularity and policy according to which the new GVT is determined. In BTB, the new GVT is simply the earliest timestamp of any event that crosses thread boundaries. This permits a clear delineation of the execute and commit phases, allowing the execute phase to proceed without any worker thread interaction. However, when simulating systems of entities that interact globally and with short delays, the resulting synchronisation windows can become exceedingly small.

Window Racer alleviates this issue by loosening the entity-to-thread assignment in the execute phase. Algorithms 4 and 5 show Window Racer's main loop and the entity-level locking and GVT negotiation as pseudo code. Each worker thread maintains a *unconditional event list* (uel) holding events guaranteed to be committed at some point throughout the simulation, and an *conditional event list* (cel) holding events generated in the current round's execute phase, some of which may have been generated in error and may never be committed. At the beginning of the execute phase, each worker thread considers the local entities' events in timestamp order. However, any newly generated events are executed as well, regardless of the target entity's thread assignment. Race conditions are ruled out by acquiring a lock on an event's target entity before saving the entity's state, appending the event to a per-entity event list, and executing the event. This allows threads to execute entire chains of dependent events without handing the execution off to other threads or separating the execution into multiple rounds.

Throughout this execution scheme, the threads negotiate the new GVT based

Algorithm 4 Main loop of the Window Racer algorithm.

```
1: global $upperBound \leftarrow +\infty$
2: global $lowerBound \leftarrow -\infty$
3: per-thread $cel \leftarrow$ PriorityQueue( )
4: per-thread $uel \leftarrow$ PriorityQueue( )
5: procedure ProcessWindow( )
6:     while not reached termination criterion do
7:         do atomically:
8:             $lowerBound \leftarrow$ ComputeGlobalMinimumTimestamp( )
9:             $upperBound \leftarrow lowerBound + \tau_0$
10:        done
11:        while GetTimestamp(EarliestEvent($uel \cup cel$)) $< upperBound$ do
12:            $nextEvent \leftarrow$ Pop($thread.uel \cup thread.cel$)
13:            Lock($nextEvent.entity$)
14:            if RegisterEvent($nextEvent.entity$, $nextEvent$) then
15:                $generatedEvents \leftarrow$ ProcessEvent($nextEvent$)
16:                $cel \leftarrow cel \cup generatedEvents$
17:            end if
18:            Unlock($nextEvent.entity$)
19:        end while
20:        $uel \leftarrow \emptyset$
21:        ThreadBarrier( )
22:        for each $entity$ do
23:            if $|entity.event\_list| = 0$
24:             or GetTimestamp(LatestState($entity$)) $< upperBound$ then
25:                $entity.event\_list \leftarrow \emptyset$
26:                $entity.state\_list \leftarrow \emptyset$
27:                continue
28:            end if
29:            Rollback($entity, upperBound$)
30:            for each $event \in entity.event\_list$ do
31:                if GetGenerationTime($event$)$< upperBound$
32:              and GetTimestamp($event$) $\geq upperBound$ then
33:                    $uel \leftarrow uel \cup event$
34:                end if
35:            end for
36:            $entity.event\_list \leftarrow \emptyset$
37:            $entity.state\_list \leftarrow \emptyset$
38:        end for
39:    end while
40:    ThreadBarrier( )
41: end procedure
```

on entity-level straggler events. When a straggler with timestamp t is encountered, the target entity is rolled back to its latest state earlier than t, and the new GVT is updated to exclude the earliest event displaced by the straggler. Through this process, the value of the global variable holding the estimation of the next GVT gradually decreases throughout the execution phase, allowing PEs to immediately cease execution when their earliest event is past the new GVT. The algorithm's name is inspired by the threads' “race” to fit as many events as possible into a gradually closing synchronisation window.

In the commit phase, threads iterate through all local entities. If necessary, the entities are rolled back to the GVT, and any events from their event lists created

Algorithm 5 Entity locking and update of the window bound.

```
 1: procedure RegisterEvent(entity, event)
 2:     Append(entity.event_list)
 3:     if GetTimestamp(event) $\geq$ upperBound then
 4:         return false
 5:     end if
 6:     if GetTimestamp(event) $<$ LatestChangeTimestamp(entity) then
 7:         $refState \leftarrow$ GetEarlierState(entity, event)
 8:         $newUpperBound \leftarrow$ GetTimestamp(refState)
 9:         do atomically:
10:             $upperBound \leftarrow \min(upperBound, newUpperBound)$
11:         done
            Rollback(entity, GetTimestamp(refState))
12:         return true
13:     end if
14:     SaveState(entity)
15:     return true
16: end procedure
```

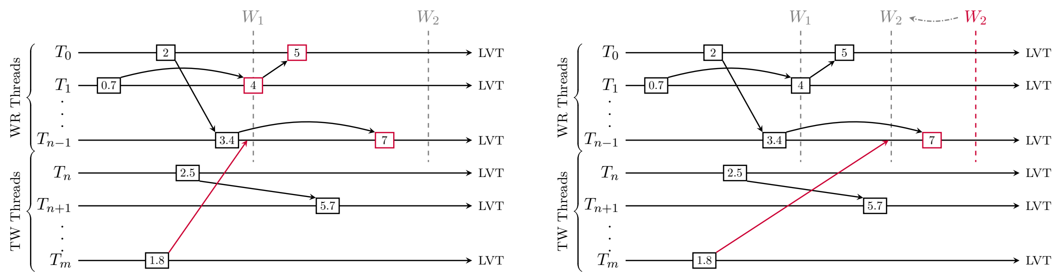
prior to the GVT but with timestamps beyond the GVT are inserted into the threads unconditional event list.

5.2 Hybrid Speculative Synchronisation

The Window Racer algorithm presented in [7] uses n threads to carry out the simulation of k simulation entities cooperatively. Conversely, in the classical Time Warp implementation [59], there is a binding between a worker thread and a set of simulation entities—this binding can be fixed or temporary, as in the case of load-balancing policies such as those presented in [46, 145, 162].

Let us consider for now a single simulation node in which m cores are used to perform processing and housekeeping tasks. Allowing the coexistence of different synchronisation algorithms, such as Window Racer and Time Warp, requires that a number $n \leq m$ of threads, at a given instant in time, performs the tasks required by the Window Racer algorithm while the remaining $m - n$ threads execute Time Warp's activities.

Given that both synchronisation algorithms can execute a complete simulation alone, the integration is trivial if there is no interaction between the simulation entities managed by either algorithm. Conversely, as soon as a simulation entity schedules an event to another entity handled by a thread running the other algorithm—we name it a *cross-algorithm* interaction—, care must be taken to ensure this interaction does not create any inconsistency in the speculative simulation



(a) Straggler Before a Window. **(b)** Straggler In the Middle of a Window.

Figure 5.1. Straggler Messages Invalidating Window Racer Windows.

trajectory. In the following, we describe the methodology to support a correct and efficient integration between the two synchronisation algorithms for cross-algorithm interactions.

5.2.1 Managing Cross-Algorithm Priority Inversion

A cross-algorithm event scheduling might require reconstructing a previous simulation state if the event is a straggler for the recipient. If the destination simulation entity is managed by a Time Warp thread, the scenario poses no harm: a traditional rollback operation can restore the previous consistent state from which to restart execution. Conversely, rollbacks are exclusively local to a window in the original Window Racer algorithm. Therefore, when the processing of a window is completed, the events associated with that window are immediately committed—Window Racer does not need a dedicated algorithm to calculate the Global Virtual Time (GVT) as in the case of Time Warp. A straggler received by a Window Racer thread from a Time Warp one requires additional information to reconstruct a previous consistent state.

There are two cases to consider related to straggler reception by a thread running in Window Racer mode, which demand different management. In the first case (see Figure 5.1a), the straggler hits before the beginning of the current or a previous window. In this case, the solution we adopt considers the entire Window Racer window as an *atomic unit of execution*. All events are undone and the simulation restarts from a previous window.

In the second case, the straggler falls within the current window (see Figure 5.1b). In this case, there is no need to flush the entire window. Indeed, all the events executed before the straggler are still (speculatively) correct. Therefore, in this case, we simply update the window's upper bound, closing the window at the straggler's timestamp. All inconsistent events will be undone.

To support the invalidation of an entire window, we must keep checkpoints also for previous windows, which was unneeded in the original Window Racer algorithm. As a first approximation, to support the cancellation of a window, we could take a snapshot of all simulation entities associated with the n threads executing in Window Racer mode before a new window starts. This approach achieves correct execution: should a straggler be received, window cancellation can be supported by restoring the state of all entities involved. However, this strategy may be suboptimal for several reasons. First, windows may be extremely short, thus requiring many checkpoints. Indeed, as shown in [2], Window Racer performance may suffer if simulation entities are in significant numbers and have a very high message exchange rate.

In the Time Warp literature, however, the problem of selecting a checkpointing interval appropriate to the dynamics of the model has been extensively addressed (e.g., in [33, 2]). These techniques generally involve the possibility of simulation model states being organised in arbitrarily complex data structures, e.g. based on the use of dynamic memory, as in [23, 102]. Therefore, in our integration, the management of checkpoints is entrusted to an *autonomic checkpoint manager* [104] that determines, during the execution of the simulation, which is the most convenient time instant for capturing a snapshot of the simulation state of an entity. This strategy brings about an important change to the Window Racer algorithm: in this way, the need to capture a snapshot before the execution of each event disappears. However, in this way, it is necessary to introduce the execution of a *coasting forward* phase in Window Racer as well to allow the realignment of a simulation state if a rollback occurs. This strategy applies also to window-local rollbacks.

Therefore, it is necessary to properly manage the simulation's *Past Event Set* (PES) to properly allow the reprocessing of events if a rollback occurs and a consistent previous simulation state needs to be reconstructed. Unlike the FES, the PES needs to maintain *per-entity* information, as a rollback affects one single entity. Therefore, a different PES is used for each entity. As a data structure, our integration relies on a doubly-linked list for the PES. The intended use of this data structure is different from the FES since its purpose is to support the efficient execution of coasting forward. This operation is linear by nature: once the first event to be reprocessed by a particular simulation entity has been identified, all subsequent events must be reprocessed in order. Hence, the advantage of using a list of events for each simulation entity. 4 • An important aspect of our integration concerns the calculation of the GVT. Since the simulation involves threads executing the Time Warp algorithm, it is not possible to disregard classical GVT calculation algorithms that determine a lower bound of the real GVT value (e.g., [84, 39, 100, 155]). However, in this GVT calculation, inspecting the PES of all entities is not necessary. Indeed, it is possible to exploit the concept of window atomicity: all entities managed by Window Racer threads enjoy the automatic GVT reduction inherent in the Window Racer algorithm. Therefore, it will be necessary to calculate the GVT reduction between the logical times of all entities managed according to the Time Warp scheme and the initial timestamp of the last correctly processed Window Racer window.

5.2.2 Event Generation and Scheduling Management

Another aspect to deal with in integrating the algorithms is managing the generation of new events and event scheduling activities. In particular, given the different risk of the Time Warp and Window Racer algorithms, we have decided to organise *future event sets* (FES) differently for the threads running the two. For both algorithms, FESs are priority queues implemented using k-heaps, as they have experimentally shown good performance in the case of disparate workloads [111]. In particular, using a k-heap, the extraction of the next event to be executed has worst case logarithmic cost, which is particularly important in the case of the n threads executing the Window Racer algorithm.

Indeed, for the n Window Racer threads, a *single shared FES* is used. While this choice requires access synchronisation (e.g. through spinlocks), the advantage lies in that the n threads can effectively cooperate in the execution of events that fall within the various windows. Conversely, the Time Warp threads use a single FES for all the associated simulation entities. This strategy deviates from classical solutions found in the literature, in which a single queue is provided for each simulation entity. However, as shown in [111], using a reduced number of queues can lead to non-negligible performance benefits. The lack of a per-simulation entity FES requires determining, upon event schedule, what is the proper FES to insert the newly-generated event into. We support this mapping by using a hash function that associates the unique id of an entity with the FES that maintains the events to be processed.

During the execution of events, newly-generated events are subject to different handling depending on their source, destination, and timestamp. Events generated by Time Warp threads are immediately delivered to the relevant FES—of course, if they are stragglers, they will cause a rollback. Nevertheless, if they fall into the current Window Racer window, they determine an update of the upper bound of the window—this is the scenario we already depicted in Figure 5.1b. Conversely, all events generated by a Window Racer thread are inserted in a per-thread *output queue*.

Logically, this output queue has a dual purpose. On the one hand, it buffers events that could be undone should a local rollback to the window cancel the processing of the generated event. On the other hand, it maintains causality information for all those events sent to Time Warp threads: in the event of a window rollback, all those messages will have to be cancelled with anti-messages.

At the end of the execution of the window, all entity events handled by Window Racer threads in the output queue are placed in the cooperative FES of Window Racer threads. Other messages are retained in the output queue until the GVT overtakes the entire window. At that point, the traditional *fossil collection* operation [59] allows the memory buffers to be retrieved. This dualistic use of the output queue allows the different degrees of risk, proper of the two Window Racer and Time Warp algorithms, to be handled correctly.

Algorithm 6 Window Management Algorithm

```
 1: global $windowUpperBound \leftarrow +\infty$                    ▷ Visible to all worker threads
 2: global $pastWindows \leftarrow$ Stack( )                        ▷ Past committed windows
 3: global $FES_{WR} \leftarrow$ PriorityQueue( )                   ▷ Window Racer Future Event Set
 4: per-thread $FES_{TW} \leftarrow$ PriorityQueue( )               ▷ Time Warp Future Event Set
 5: global $rollbackWindow \leftarrow false$                        ▷ If set, an older window will be restored
 6: procedure ProcessWindow( )
 7:     $outputQ \leftarrow$ PriorityQueue( )                       ▷ Events generated in the current window
 8:     while Next($FES_{WR}$) $<$ $windowUpperBound$ OR Next($outputQ$) $<$ $windowUpperBound$ do    ▷ $FES_{WR}$ is accessed atomically
 9:         if Next($outputQ$) $<$ Next($FES_{WR}$) then
10:             $nextEvent \leftarrow$Pop($outputQ$)
11:         else
12:             $nextEvent \leftarrow$ Pop($FES_{WR}$)
13:         end if
14:         $simEntity \leftarrow$ GetEntityOf($nextEvent$)
15:         Lock($simEntity$)                 ▷ Mark the simulation entity as being processed by a WR thread
16:         if GetTimestamp($simEntity.lastProcessedEvent$) $>$ GetTimestamp($nextEvent.time$) then    ▷ Straggler detected
17:             UnLock($simEntity$)
18:             if GetOriginPartition($event$) $= TW$ then
19:                 $rollbackWindow \leftarrow true$
20:             end if
21:             do atomically:
22:                 if $windowUpperBound >$ GetTimestamp($nextEvent$) then
23:                     $windowUpperBound \leftarrow$ GetTimestamp($nextEvent$)
24:                 end if
25:             done
26:             break
27:         end if
28:         $generatedEvents \leftarrow$ ProcessEvent($nextEvent$)    ▷ Calls the model event handler and returns the generated events
29:         UnLock($simEntity$)
30:         $outputQ \leftarrow outputQ \cup generatedEvents$
31:     end while
32:     ThreadBarrier( )                                            ▷ Window is over
33:     if $rollbackWindow$ then
34:         $outputQ \leftarrow \emptyset$
35:         do
36:             $windowToRestore \leftarrow$ Pop($pastWindows$)
37:         while $windowToRestore \geq windowUpperBound$
38:         for each $simEntity$ do
39:             Rollback($simEntity, windowToRestore$)
40:         end for
41:         ThreadBarrier( )
42:         $rollbackWindow \leftarrow false$
43:         return
44:     end if
45:     $pastWindows \leftarrow pastWindows \cup \{windowUpperBound\}$
46:     for each $event \in outputQ$ s.t. GetGenerationTime($event$) $< windowUpperBound$ do
47:         if GetDestinationPartition($event$) $= TW$ then
48:             $FES_{TW} \leftarrow FES_{TW} \cup \{event\}$
49:         else
50:             $FES_{WR} \leftarrow FES_{WR} \cup \{event\}$
51:         end if
52:     end for
53:     $outputQ \leftarrow \emptyset$
54:     for each entity do
55:         Rollback($simEntity, windowUpperBound$)
56:         FossilCollection($simEntity$)
57:     end for
58:     ThreadBarrier( )
59: end procedure
```

Overall, the operations carried out by Window Racer threads are reported in Algorithm 6. The global FES (line 2) is shared among all worker threads cooperating to process the current window (line 3). The Window Racer threads maintain a per-thread output queue (line 4) that is used in conjunction with the FES to determine when the processing of the current window is over and what is the next event to schedule (lines 5–10). As discussed, the output queue is also used to keep track of all the generated events (line 22, 24).

Given the cooperative nature of Window Racer, we must ensure that a single simulation entity is not concurrently executed by two different worker threads. To this end, we employ a locking mechanism based on atomic read-modify-write instructions that ensure that only a single worker thread will take care of an entity if two events are extracted concurrently—lines 12, 14, 23.

If a straggler message is received, the worker thread managing the entity hit by the straggler will reduce the window size (lines 13–19). The update of the upper bound must be done atomically, because multiple threads may be managing stragglers at the same time. In the case of concurrent update, the minimum among all the new tentative values should be stored. Therefore, we rely on a Compare-and-Swap based retry loop, thus implementing a non-blocking update.

The window is completely processed when the next event to be processed is scheduled at a timestamp beyond the window's upper bound (line 5). At this point, the threads should deliver the events still present in the output queue to the FES (lines 27–31). The presence of the synchronisation barrier (line 26) is a legacy of the original Window Racer algorithm, but is not strictly necessary.

5.2.3 Dynamic execution mode switching

An important aspect of managing a hybrid synchronisation mechanism such as the one proposed in this work is the possibility of dynamically switching from one execution mode to another. Indeed, as was shown in [2], depending on the dynamics of the simulation model, different synchronisation modes may prove to be successful. Clearly, in a general simulation, it is possible for these dynamics to change, just as it is possible for different parts of the model to behave differently. Therefore, to

Algorithm 7 Mode Switch from Time Warp to Window Racer

```
global $need\_switch$
global $switching\_thread\_id$
global $warp\_threads\_count$
global $racer\_threads\_count$
procedure DoSwitch( )
    if NOT $need\_switch$ then
        return
    end if
    if ThreadType( ) $= WARP$ OR NOT ThreadId( ) $= switching\_thread\_id$ then
        return
    end if
    ThreadBarrier( )
    if ThreadId( ) $= switching\_thread\_id$ then
        if ThreadType( ) $= WARP$ then
            $warp\_threads\_count \leftarrow warp\_threads\_count - 1$
            $racer\_threads\_count \leftarrow racer\_threads\_count + 1$
            SetThreadType($RACER$)
        else
            $warp\_threads\_count \leftarrow warp\_threads\_count + 1$
            $racer\_threads\_count \leftarrow racer\_threads\_count - 1$
            SetThreadType($WARP$)
        end if
        $need\_switch \leftarrow False$
    end if
    ThreadBarrier( )
end procedure
```

maximise performance, it is desirable that the number n of threads running in a given mode changes during the same simulation.

The different nature of the two synchronisation algorithms considered in this work requires certain precautions to make this transition effective. Let us first consider the simplest case in which a thread executing in Time Warp mode must switch to Window Racer mode. The organisation of FESs described above requires that, in the transition, a Time Warp thread inserts all future events of its FES into the one shared between all threads executing in Window Racer mode.

However, the transition from Time Warp to Window Racer mode must be performed to minimise the invasiveness concerning Window Racer execution. If a Time Warp thread merely modifies the FES of Window Racer threads, artificial rollbacks may be introduced related to this platform-level operation. On the other hand, its windowed nature allows the upper bound to be exploited to discriminate between messages that may generate a rollback and those that certainly do not. The thread that wants to migrate from Time Warp to Window Racer execution, therefore, can adopt the scheme shown in Algorithm 7.

Initially, the migrating thread signals the start of the transition so that the

Window Racer threads are forced to wait at the end of the current window. It then checks the upper bound value of the current window. If this value is less than the time of the next event in its FES, all the events to be processed belong to the *next* window. Therefore, the thread moves its events into the Window Racer FES.

Conversely, if its next event has a timestamp less than the upper bound, we are in the scenario described above in Figure 5.1. In this case, if the thread were to insert new events into the Window Racer queue, it could generate a priority inversion concerning the activities of the n threads executing in Window Racer mode. Therefore, the Time Warp thread in migration appropriately exploits the risk of Time Warp, as shown in Algorithm 7: it will continue to process events in the FES as long as the logical time of the next event does not exceed the upper bound of the current window. The termination of the procedure is guaranteed by the signalling flag, which forces the Window Racer threads to wait for the conclusion of the mode transition.

This migration approach could lead to high costs if not handled correctly. Indeed, the pool of Window Racer threads may have to wait a long time for the transitioning Time Warp thread to complete its realignment. Therefore, when choosing a thread to switch from Time Warp mode to Window Racer mode, selecting a thread that is further along in logical time than the current window is crucial. If no such thread exists, the choice should fall on the one with the next event closest in logical time to the end of the current window. However, we emphasise that this aspect, as well as the choice of when to make the transition and the number of Time Warp threads involved, requires the definition of an *autonomic policy* outside this work's scope.

Switching from Window Racer execution mode to Time Warp requires more care. Threads executing in Window Racer mode have no inherently bound simulation entities: events associated with a given entity can be cooperatively executed by multiple threads in alternation. If a Window Racer thread is to turn into a Time Warp thread, it becomes necessary to manage the output queue discussed above appropriately. Including all current messages in the output queue in the various FESs would not be consistent with its handling as described above. Conversely, we place all events destined for an entity running on Time Warp threads (including the thread performing the mode change) in the corresponding FESs. On the other hand, the output queue is assigned to a Window Racer thread[1], which will carry on its management in a manner consistent with what is described in Section 5.2.2.

5.2.4 Going Distributed

So far, we have focused on managing event processing by considering only one node. Using our hybrid synchronisation scheme in a distributed setup is straightforward.

Indeed, Window Racer was born as an algorithm for parallel but not distributed systems. Conversely, Time Warp is inherently capable of handling the causality violations that may arise from a distributed execution. Therefore, the hybrid scheme we have described can be immediately used on distributed deployments due to the presence of Time Warp.

Considering windows as atomic processing units ensures that, unlike the original proposal in [7], if a straggler message is received prior to a window, it will be cancelled entirely.

In a distributed deployment, therefore, the fact that there are multiple concurrent instances of the Window Racer algorithm running on multiple nodes does not require making these instances aware of the presence of the others. If a straggler message is received by a pool of Window Racer threads, they will work together to restore an earlier state of the window affected by the straggler, resuming execution.

5.3 Experimental Assessment

In this section, we detail the setup and the results of the experimental assessment we carried out for our proposal.

Our analysis was conducted using a machine equipped with two AMD® EPYC™ 7452 processors @ 2.9 GHz, each consisting of 32 physical cores and 64 hyperthreads, for a total of 64 physical cores and 128 hyperthreads—hyperthreads were turned off

[1]We currently pick a random Window Racer thread for this purpose: the envisaged autonomic policy could determine the best-suited target, depending on the current load.

in the experimental assessment. The machine is equipped with 256 GB of RAM. All experimental results are averaged over 20 different runs.

5.3.1 Testbed Applications Configuration

For the PHold benchmark, we have simulated a total of 131,000 simulation entities that have been mapped to a variable number of threads in multiple runs. The entities are divided into a *dense* and a *loose* partition, the former composed of 1000 entities and the latter of the remaining 130,000. When an entity has to randomly select a destination for an event, with a 50% probability it will pick a random destination in the dense partition. Otherwise, it will pick a random destination in the loose partition. This way, we can mimic a scenario where a part of the simulation has a higher load.

The dense partition is assigned for execution to Window Racer threads, while Time Warp threads execute the loose partition. Depending on the number of threads, there could be a significant skew on the logical clocks of the entities that could increase the rollback probability. The dense partition is the one that suffers most from rollbacks because they have a denser concentration of events that a single straggler can undo.

Regarding the epidemiologic agent-based model, the SIRS variant is considered, where recovered agents eventually revert to the vulnerable state. Each agent in our model is located in one of an adjustable number of fully connected domains, each of which has the same initial number of agents. Each agent has eight randomly selected neighbours in the same area, so the number of areas affects how localised agent interactions are.

Transition delays are drawn from exponential distributions with fixed or dynamic rates. The infection rate for susceptible agents is proportional to the number of infected neighbours. As a result, agents entering or leaving the infected state must notify their neighbours to reschedule their transition to the infected state based on the new rate. Transitions from the recovered state to the susceptible state occur at a constant rate of 1. Two other transitions introduce dynamic changes to the topology defined by the neighbourhood relationships of the agents. The first type of transition randomly changes an agent's neighbours within its current region, potentially changing its infection rate or the infection rates of its neighbours. The second type of transition involves the movement of an agent.

The second type of transition randomly moves an agent to another region and connects the agent to new neighbours in the new region. The rates at which these two types of transition occur allow us to control the degree of computational load and agent interaction within each region, as well as the interdependence of transitions between regions. Overall, this system is similar to epidemic models used in real-world epidemic studies [49], which attempt to capture the effects of daily and long-distance mobility of populations.

5.3.2 Experimental Results

In Figure 5.4, we show the speedup over the corresponding sequential simulation for the PHold benchmark, using a total of 16, 24 and 32 threads empowering the two integrated algorithms. The plots depict the performance variation as the number of threads responsible for simulating the high-activity partition changes. The two curves correspond to the high-activity partition simulated by the TW or WR threads. In some configurations, simulation runs are more than 10 times slower than the corresponding serial execution. The dashed line represents the configuration where the high-activity LPs are uniformly distributed among processing threads to balance the computational load. With our knowledge of the chosen models, this can be accomplished statically and accurately.

In the imbalanced model, the total execution time changes drastically depending on the workload partitioning to the threads. Noteworthy, when using TW and WR for the high-activity partition, a single configuration exhibits the best performance, independently of the total thread count. A less apparent observation is that the optimally partitioned configurations surpass the load-balanced configuration in performance. This is because, despite achieving better load balance, the communication of the high-activity partition becomes dispersed into more threads, resulting in less-efficient event management operations.

The important result is that, with PHold, the best static configuration is always found using only TW threads. Nevertheless, thrashing phenomena are observed for the TW algorithm if the thread count is too high. This is because when fewer LPs are bound to a TW thread, the system becomes over-optimistic, and the high-activity partition is subject to more rollbacks. The rollback cost is much higher for the high-activity partition, as more work is wasted. In contrast, the WR algorithm is less sensitive to increased concurrency in the high-activity partition because WR threads can effectively leverage the reduced aggressiveness to decrease the rollback occurrence in the simulation. Apparently, WR threads are more resilient to tighter simulation interactions, but their average throughput is lower. This is evidenced by configurations equipped with WR achieving peak performance with more threads.

Figure 5.5 reports the SIR-model speedup for the same thread configurations. As can be seen, the scenario is significantly different: we consistently achieve improved performance over the sequential simulation when WR threads manage the high-activity partition. This is because the cost of rollbacks is higher, and the LP interactions in the high-activity partition are highly dynamic and tightly coupled. In this scenario, TW becomes excessively optimistic, causing significant clock skewing and leading to worse performance. The more cautious approach of WR can achieve better returns from optimism, reducing the overall number of rollbacks.

In summary, Figure 5.3 demonstrates that for the PHOLD model, the partitioned TW configurations yielded the best performance, whereas, in the SIR model, the hybrid configurations performed the best. However, with 32 threads, all configurations were more or less on par. The SIR model has a high degree of coupling, so even with 24 threads, we achieved a speedup comparable to what was attained with 16 threads. This indicates that we were nearing the parallelisation limit for the model, at least using our current techniques. Interestingly, the load-balanced TW compensated for its inefficiencies through sheer processing power. Nevertheless, we can deduce that the hybrid approach is considerably more efficient for the SIR model, especially when using fewer threads.

To provide further insight into these findings, we have illustrated the total number of rollbacks experienced by the TW and WR thread pools throughout the simulation in Figure 5.6. The results show that the WR configuration experiences

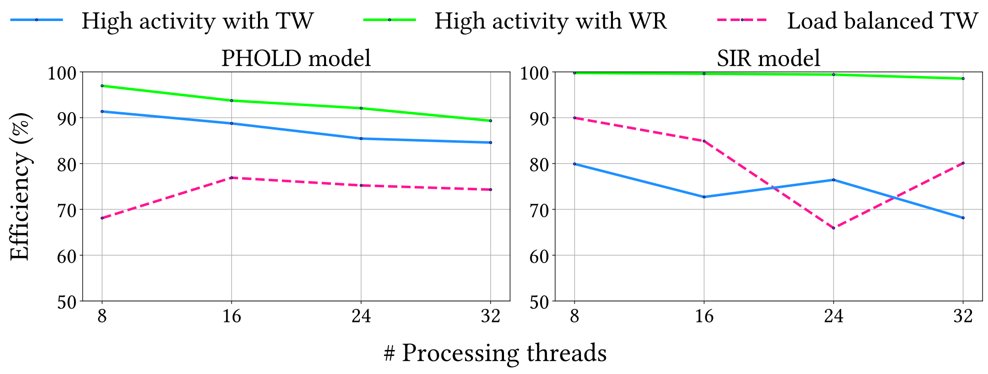


Figure 5.2. Efficiency, picking the best partitioning

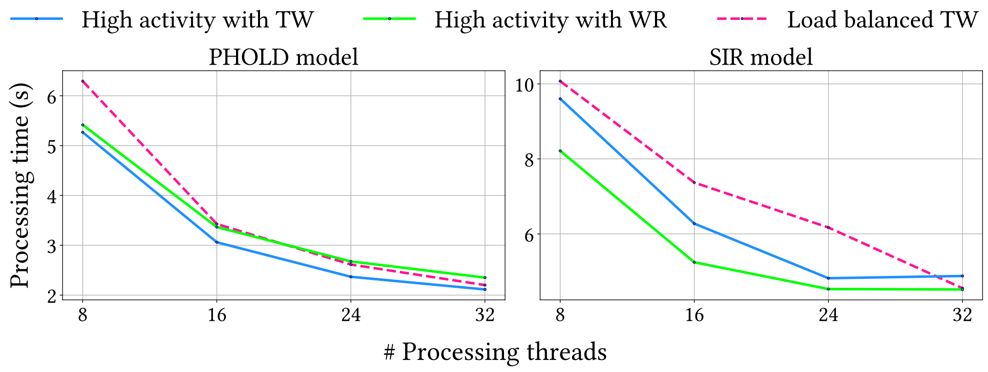


Figure 5.3. Scaling, picking the best partitioning

significantly fewer rollbacks overall. This is due to the algorithm's avoidance of over-optimism, resulting in a lower probability of a WR thread engaging in an incorrect speculative trajectory. In contrast, the configuration using only TW threads experiences a much higher rate of rollbacks.

Figure 5.7 demonstrates that the high-activity partition managed by TW is responsible for the increased number of rollbacks in TW-only configurations. In contrast, the very same partition managed by WR generates a minimal amount of antimessages. However, the WR threads must wait at the end of windows, leading to blocking synchronisation when committing new windows. Therefore, considering the better performance provided by the hybrid configuration, it can be inferred that the time spent by TW threads in incorrect trajectories and sending antimessages is superior but roughly comparable to the time spent by WR in blocking wait.

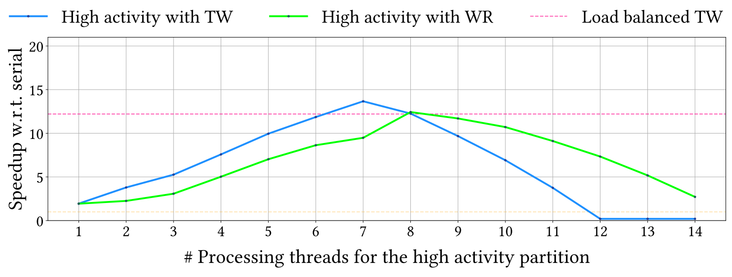
The last finding is validated by the simulation efficiency shown in Figure 5.2, demonstrating that the hybrid approach outperforms the other strategies for both the evaluated models. Efficiency is determined by computing the proportion of committed events in relation to the overall number of executed events, expressed as a percentage. It is once again evident that the WR's less aggressive characteristics significantly decrease rollback occurrences, albeit at the expense of idle periods on processing threads. In particular, in a model such as SIR, with a limited degree of parallelism that can be captured in the high-activity portion of the model, the hybrid synchronisation method can achieve remarkably high levels of efficiency.

These plots over time show that, even for a simplified epidemic model, we observe a dynamicity in behaviour that an autonomic policy can potentially exploit. From this overall experimentation, we can draw the following observations. Evenly distributing the computational workload among threads in a uniformly synchronised simulation may not always be optimal, as communication costs, even within the same machine, can influence its effectiveness. Partitioning the simulation in uneven ways can bring measurable performance benefits and allows using different synchronisation algorithms that better adapt to the model's dynamics.

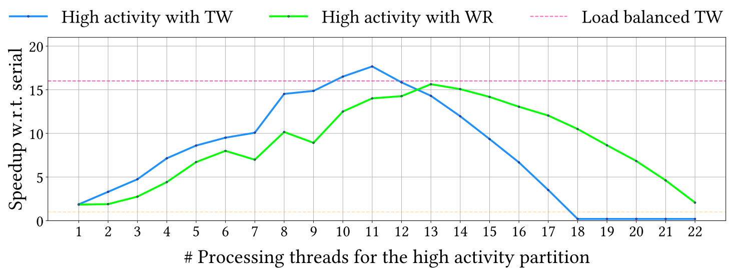
From this overall experimentation, we can draw the following observations. Depending on the model dynamics, Window Racer and Time Warp can adequately capture the model's parallelism under different conditions. In all cases, there is an optimal static configuration that depends on the model characteristics. Since the workload dynamics can change over time, we emphasise that this optimal static configuration may also vary. Therefore, the autonomic policy we envisaged in this Chapter is fundamental because it can capture the best-suited parallelism level in certain simulation phases, and tune the configuration to deliver better performance.

5.4 Discussion

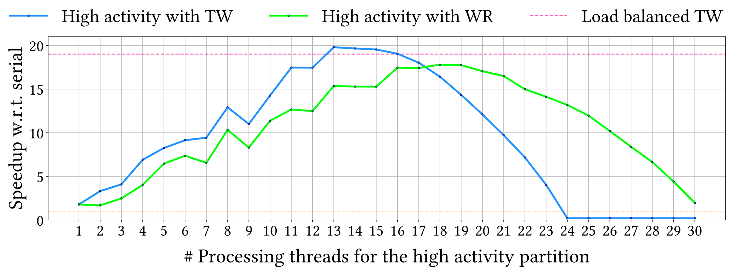
The experimental results that we have shown clearly demand the introduction of an autonomic policy that can allow for the proper selection of the number of threads running the Window Racer and the Time Warp algorithms, depending on the workload of the model. Furthermore, an additional dimension of optimisation could entail dealing with multiple pools of Window Racer threads, thus exploring how



(a) 16 threads



(b) 24 threads



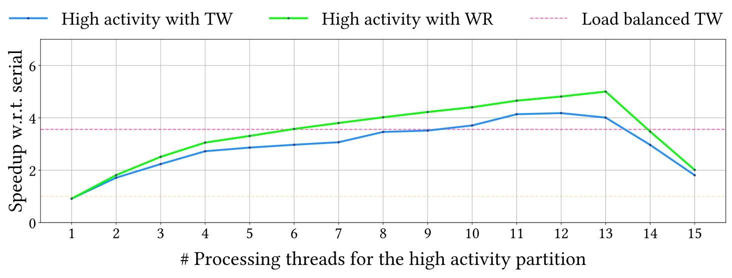
(c) 32 threads

Figure 5.4. PHold Performance

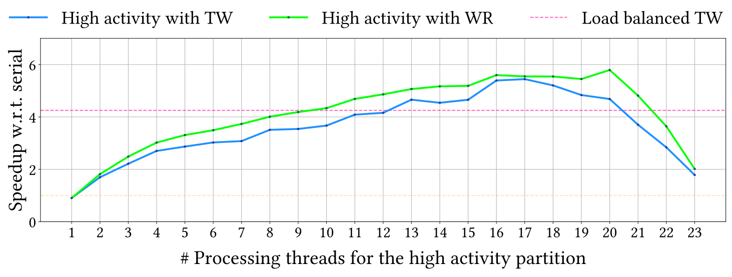
multiple FESs for the Window Racer part may affect the simulation performance.

Although we delegate the study of an autonomic policy for runtime selection of the best configuration in relation to model dynamics to a future line of research, the work we have presented in this Chapter emphasises once again that a general-purpose runtime environment cannot be realised without self-optimisation techniques. The effects of autonomic risk management policies will be shown in action in Chapter 6, albeit in a different context.

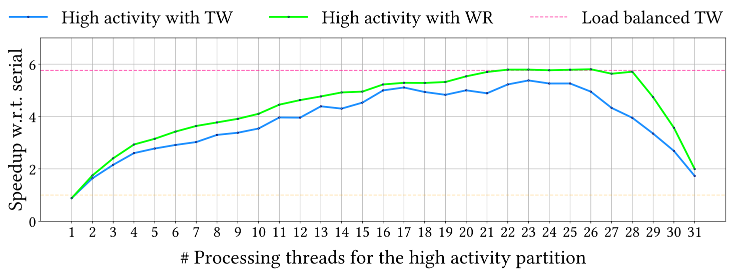
Finally, as mentioned earlier, the integration methodology presented in this chapter can easily be extended to other synchronisation algorithms. The introduction of new optimistic algorithms appears immediate: the presence of Time Warp makes this operation painless. Indeed, since it is the algorithm that runs the most



(a) 16 threads



(b) 24 threads

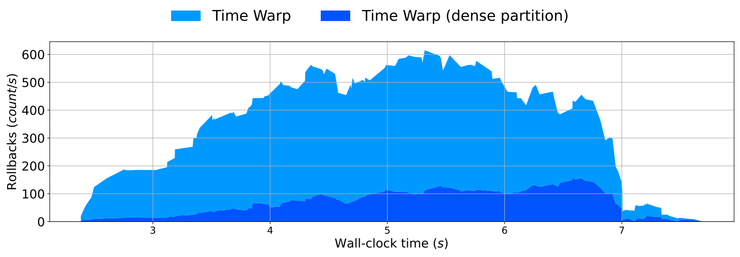


(c) 32 threads

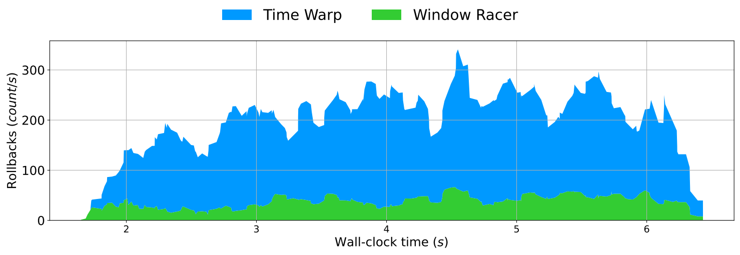
Figure 5.5. SIR Performance

significant risk, it is also the one with the vastest requirements to recover from incorrect trajectory portions. Therefore, as we have done for Window Racer, it is always possible to correct the problems of other less aggressive algorithms by exploiting Time Warp's capabilities.

On the other hand, additional pieces of information must be included should we want to introduce conservative algorithms, as theorised in [61]. The fundamental one is the concept of lookahead. Indeed, by introducing a lookahead value (either provided by the modeller or estimated by the execution environment), it will be possible to expand the autonomic decision-making model by also considering the possibility of entirely avoiding the cost of supporting the corrective actions of speculative algorithms.

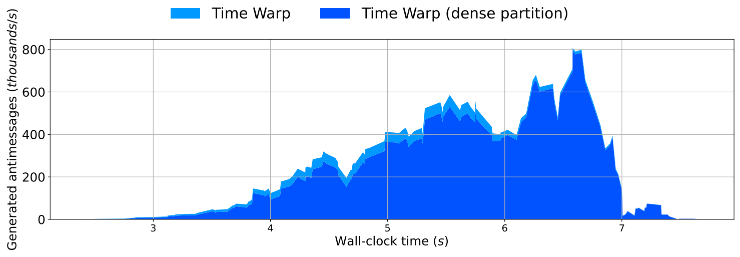


(a) Best Time Warp-only configuration

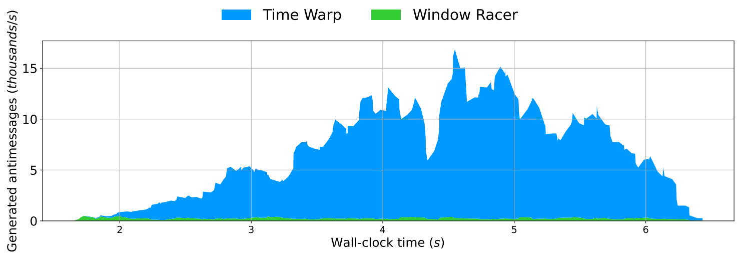


(b) Best hybrid configuration

Figure 5.6. SIR Rollbacks over time: 24 threads



(a) Best Time Warp-only configuration



(b) Best hybrid configuration

Figure 5.7. SIR Antimessages sent over time: 24 threads

Chapter 6

Risk Management in Stochastic Simulations

One of the facets of risk is to allow diverging from the correct simulation trajectory. As we have discussed extensively, in the case of Time Warp, the aim is to contain or correct the effects of risk to avoid incorrect simulation results.

However, in the case of stochastic simulations, this risk may be acceptable. Indeed, stochastic simulations provide results that approximate what happens in reality—the use of probability distributions already introduces a certain degree of approximation. Therefore, the question arises as to whether it is possible to accept a certain level of risk, thus deviating from the result of a more precise simulation and providing higher performance.

In this direction, the recent work in [120] has shown how it is possible to obtain simply from the model developer information related to how much risk it is willing to tolerate, i.e. how much a stochastic model can bear divergences in the simulation, while still providing acceptable results. It is clear that the modeller's intervention is essential in this context: different simulation models, or even different studies based on the same model, may have different accuracy requirements. Therefore, it is necessary to allow the execution environment to obtain information on the degree of accuracy of interest to manipulate the risk inherent in the executions correctly.

However, the modeller is mainly interested in the model aspects. Therefore, he can only provide information on the degree of accuracy he expects from the simulation results. Basing execution environment choices solely on this information could be counterproductive.

Indeed, the modeller may select a degree of detail (or approximation) such that the level of risk is too high. As we showed in Chapter 5, depending on the dynamics of the model, the degree of acceptable risk from the point of view of performance may vary significantly. Therefore, as theorised in Chapter 5, it is crucial to develop autonomic self-optimising policies that allow the execution environment to correct execution dynamics to manage risk appropriately.

In this chapter, we show how it is possible to extend the work in [120] to make the choices made by the runtime environment aware of the degree of risk being observed in the execution of the model. Referring to more classical concepts in the literature, the risk manipulation we implement in this chapter concerns both the accumulated error in the simulation results and the aggressiveness of the simulation itself. In our new definition of risk, these two concepts are subsumed into one.

6.1 Approximate Rollbacks

The technique proposed in [120] is named *approximate rollbacks*. It stems from the observation that the typical way to support the rollback operation of an LP is to *precisely* reconstruct its last correct state. Optimisations to this approach (e.g., [119, 15, 18]) were mainly involved in reducing its cost.

Conversely, the work in [120] took an orthogonal approach to the cost reduction for rolling back the LP to a past state based on risk tolerance in simulation. The fundamental idea of this work is that it is not necessary to precisely reconstruct a previous simulation state of a stochastic simulation, provided that the accuracy is compatible with the goals of the simulation study. Therefore, an approximate rollback is based on the idea that we may obtain sufficiently-accurate simulation results even if the LP affected by a causality error resumes its execution from a state that is a reasonable approximation of the one that should have been restored.

The central concept behind approximate rollbacks is that the state of an LP can be partitioned into *core* and *non-core* portions. The core portion allows reconstructing the whole state in an approximated way, so only the core portion can be dealt with in state saving and restore operations. Of course, the possibility of devising a part of the LP state as the core is application-dependent; hence the approximate rollback technique requires a bit more intervention from the application programmer in terms of interaction with the state management logic offered by the optimistic runtime environment. To reduce the need for intervention by the programmer, approximate rollbacks provide a state-management architecture that allows saving the core-state portion transparently and supports an application-level callback which, starting from a saved core-state portion, will *approximately* reconstruct the whole state when a rollback occurs. Our architecture also offers the possibility to dynamically change the identification of the core-state portion, further optimising the execution of the approximate rollback technique when the LP state dynamically changes in shape and semantics.

Our architectural proposal still enables the coexistence of approximate rollback phases and traditional non-approximate ones based on the precise reconstruction of past states. Also, approximate rollbacks leave the simulation model with the possibility to switch between the two at runtime. The final effect is to enable the simulation model to choose what phases of the simulation run can tolerate approximations and what cannot. This choice can depend on the runtime observation of the actual state of an LP and its evolution trajectory so that the approximate mode can be enabled or disabled depending on whether specific predicates hold or not. Ultimately, activating or deactivating this support can be based on the passage of logical time and the desire not to allow more than a certain percentage of the elapsed logical time to be handled with approximate rollbacks.

The benefit of this approach is mainly related to the performance improvement that can be observed thanks to the reduction in the overhead related to state saving/ restore proper of optimistic simulations. Of course, the drawback is a (possible) loss in simulation results' accuracy. Nevertheless, this loss of accuracy can be tolerated in many scenarios specifically related to stochastic simulations, where we typically estimate the properties of the simulated system through statistical techniques.

One of these scenarios is model calibration [30], during which selected parameters are varied within reasonable bounds until a sufficient correspondence between one or several model output variables and respective measurements is obtained. This phase could require running many simulations so that a performance improvement in a single simulation run can result in a non-negligible time reduction of the overall calibration process. Similarly, and for the same reasons, also sensitivity analysis [140] could benefit from approximate rollbacks.

Of course, the benefits of approximate rollbacks depend on the actual simulation dynamics, so some models (or model's execution phases) might provide better performance when run in precise mode. To account for this specific aspect, we introduce an autonomic policy that determines at runtime, based on the current model dynamics, what is the best-suited execution mode. In this way, the model developer is only required to mark the core portions of the simulation state. The runtime environment will then self-tune its internal operations to improve the simulation performance.

The approximate rollback technique deals with the capability of a speculative PDES application to resume its execution after a rollback bringing an LP back to logical time T, from a state $\tilde{s}$ not fully matching the original state s observed in forward execution at the same time T.

Let us denote with $VT(x)$ the virtual time of a state x and $I(x)$ the correct information stored in the state variables forming the state x. Let us also call $\tilde{s} \leftarrow s$ the relation between two states $\tilde{s}$ and s such that $\tilde{s}$ represents the restoration of s after a rollback. Then, the approximate rollback technique can be described through the following expression:

$$\tilde{s} \leftarrow s \Rightarrow (VT(\tilde{s}) = VT(s)) \wedge (I(\tilde{s}) \subseteq I(s))\,. \tag{6.1}$$

By Equation 6.1, the logical time of the two states $\tilde{s}$ and s involved in the $\tilde{s} \leftarrow s$ relation is identical. At the same time, s is a superset of the correct information maintained in $\tilde{s}$. In other words, the approximate rollback technique restores a state with some missing correct information. Still, no added information (i.e., no added state variable) in $\tilde{s}$ was not initially present in s. This aspect is crucial from the point of view of model implementation since the application programmer knows that after an approximate rollback, they will never need to deal with a state variable value not already observed in forward execution.

Concerning model execution, the relation $I(\tilde{s}) \subseteq I(s)$ implies that to create a snapshot that can be used to restore $\tilde{s}$, we can log fewer data compared to s when observing s in forward execution. While this is typical of incremental state-saving techniques, approximate rollbacks do not relate data saved in the log to write operations occurring in forward execution: we only need to skip parts of s to create the log used to restore $\tilde{s}$. This also means that we do not need any tracing mechanism (based on, e.g., software instrumentation) to track state updates to reduce the log size. Also, we do not need to re-traverse the incremental log chain backwards to rebuild s. We simply restore the saved *core information* of s, namely $I(\tilde{s})$ rather than $I(s)$, which enables restoring $\tilde{s}$ in place of s.

Another implication of the relation $I(\tilde{s}) \subseteq I(s)$ is that the approximate state $\tilde{s}$ does not guarantee to precisely record the *history* of the events processed at the LP up to the restoration time T. Indeed, parts of the information stored by the events in the initially-observed state s are no longer available after rolling back to $\tilde{s}$. This behaviour contrasts with the sparse state-saving technique, which reduces the total amount of logged state information but allows reconstructing a precise state via the coasting forward operation. Indeed, the coasting forward phase can regenerate a simulation state precisely recording the whole history of events that have been processed without a causality violation, including all intermediate events between the last correct checkpoint and the timestamp T of the last-correct event. Avoiding this type of precise state reconstruction makes the state restoration time upon a rollback independent of the granularity of the events.

Overall, large states and coarse-grain events (possibly updating large portions of the state) are no longer a concern when employing approximate rollbacks, especially when $I(\tilde{s})$ is a significantly 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 computation. Indeed, coarse-grain events with many updates on the state will likely lead to coarse-grain backward computing steps, independently of whether the backward computing phase is based on reverse event handlers [15] or reverse reconstruction of memory-location values [18].

A final point must be discussed, related to the following question: «*once we restore $\tilde{s}$ (instead of s), are we able to guess the missing piece of information between $I(s)$ and $I(\tilde{s})$?*». Considering stochastic simulation as the target, the answer is yes. In particular, we can complement the restoration of $\tilde{s}$ with the invocation of an application-level callback that takes $\tilde{s}$ and changes it to (possibly) reduce the distance between $I(s)$ and $I(\tilde{s})$. Overall, denoting with CF this callback function, we have that the final state restored at time T in an approximate rollback is $CF(\tilde{s})$. Of course, introducing CF to support the approximate reconstruction of the state from which to resume the simulation after a rollback adds some runtime overhead to the state reconstruction process. However, we still avoid the dependence between the cost of the approximate state reconstruction and the amount of rolled-back events.

At this point, we can provide another indication of how to build $\tilde{s}$, which is the core of s, to make the callback function CF effective: $\tilde{s}$ should include information on what parts of the state are missing between s and $\tilde{s}$. However, the missing pieces are a "do not care" and will be regenerated (approximately) by the callback function CF. It is left in the hand of the model developer to determine what information is suited for the approximate reconstruction via CF and what is more relevant to the simulated stochastic process—this part should be maintained in the core of s.

6.2 The Autonomic Policy

As already mentioned, approximate rollbacks aim to reduce the burden on the simulation runtime environment for supporting/performing a rollback operation. When the simulation state s is much larger than the approximate state $\tilde{s}$, i.e. when core memory is a tiny fraction of the overall simulation state, the cost to save/restore a previous simulation state can be expected to be small. At the same time, to correctly resume the simulation, the runtime must invoke CF, whose execution time might be non-minimal, especially if the simulation model state has arbitrarily complex invariants to be respected.

Therefore, we propose an *autonomic policy* that allows switching between the precise and approximate modes at runtime, based on model-related dynamics. The goal of this policy is to support effective, transparent decision-making at the level of the simulation runtime environment. In particular, the simulation model developer is still required to enable the approximate mode and to explicitly mark the core portions of the simulation state, but then the runtime environment can quickly switch between the two execution modes to reduce the overall performance penalty. Of course, if required, the modeller can still explicitly switch to precise execution to ensure the correctness of some phases of the simulation run, thus bypassing any autonomic decision by the runtime environment. Hence, our autonomic policy can only operate along any phase where the approximate mode is permitted.

We base our autonomic policy on two analytical models that describe the *per-event overhead*, i.e. the per-event time spent in activities required to support the restoration of a previous consistent state. Our models capture that if no rollback is ever executed, the state-saving overhead is wasted time, subtracted from fruitful forward event processing. First, we define the per-event overhead when executing in precise mode as follows:

$$OH^P = \frac{s\delta_s}{\chi} + P_r\left(s\delta_r + \delta_c\frac{\chi-1}{2}\right) \tag{6.2}$$

where s is the average size of the simulation state, δ_s is the average cost to log a *single byte* of the simulation state, δ_r is the average cost to restore it, χ is the checkpointing interval, P_r is the rollback probability, and δ_c is the average time to process an event during the coasting forward phase (i.e., the *silent execution*). This model complies with what is already proposed by the literature in the context of precise rollbacks supported via state saving [135].

Equation (6.2) captures that, for each event processed in forward execution, we pay the cost to take a periodic checkpoint $s\delta_s$ distributed over the events that fall in the checkpointing interval χ. With a certain probability P_r, we also pay the cost to restore a previous simulation state. This cost accounts for restoring the previous consistent state ($s\delta_r$) and the coasting forward phase, which on average requires replaying half of the events in a checkpointing interval [33]. We explicitly consider the cost δ_c for the re-execution of events because, due to their silent execution nature, the granularity of replayed events can be smaller than when in forward execution because event messages are not actually delivered.

Similarly, we define the per-event overhead when executing in approximate mode as follows:

$$OH^A = \frac{\alpha s \delta_s}{\chi} + P_r \left(\alpha s \delta_r + R + \delta_c \frac{\chi - 1}{2} \right) \tag{6.3}$$

where, beyond the previously defined variables, R is the average cost to execute the restore function CF and α is the fraction of state marked as core by the model developer, i.e. such that $\tilde{s} = \alpha s$. Equation (6.3) captures that, when running in approximate mode, we pay a reduced state saving cost because we do not log the whole simulation state ($\alpha s \delta_s$), but in case of a rollback, the state-restore cost increases proportionally to the cost R for executing the CF function.

The checkpointing interval χ must be properly set for the two overheads to be comparable. In particular, independently of the execution mode, an *optimal* checkpointing interval χ_{opt} must be selected. This ensures that the computed overheads do not depend, even indirectly, on a suboptimal execution strategy which, for example, requires replaying more events than needed when running the coasting forward phase. In this way, OH^P and OH^A can be evaluated using χ^P_{opt} and χ^A_{opt}, respectively. To select the proper value of χ_{opt}, we borrow from classical optimal checkpointing interval selection strategies, such as the one presented in [74]. In particular, χ_{opt} can be computed as:

$$\chi_{opt} = \left\lceil \sqrt{\frac{\omega}{P_r \delta_c}} \right\rceil \tag{6.4}$$

where ω can be set to $2s\delta_s$ for the precise scenario, and to $2\alpha s \delta_s$ for the approximate scenario. Equation (6.4) captures that the checkpointing interval should be increased if the state-saving cost is high, but not too much to pay a higher coasting-forward cost in case of a rollback[1]. Moreover, it accounts for the fact that the rollback behaviour itself can be affected by variations in the checkpoint interval. Of course, any other optimal checkpointing interval selection algorithm (see, e.g., [33]) can be suitably employed.

Once the proper values χ^P_{opt} and χ^A_{opt} are determined, we can compare OH^P and

[1]In any case, a classical upper bound on the checkpoint interval needs to be used—like the value 40 selected in various works [135, 5]—in order to avoid negative interference with fossil collection.

OH^A. Our autonomic policy makes a decision on the best-suited execution mode by computing:

$$\beta = \frac{OH^P}{OH^A} \tag{6.5}$$

so that the approximate execution mode is selected if $\beta > 1$, or more conservatively if $\beta > 1.2$ or a higher value that expresses a given minimal percentage of execution speed improvement by the approximate rollback technique.

We note that our autonomic approach has the objective of determining if the approximate strategy that the application layer would like to use can result effective for what concerns performance. Hence, we exploit equations (6.2) and (6.3) according to the following sequence of steps:

1) we enable an execution phase with approximate rollback when the application requests it—this phase allows us to estimate at runtime all the values of the parameters required to compare approximate and precise rollbacks according to the above equations;

2) we select the best performing of the two solutions—according to the ratio in equation (6.5).

However, if the precise mode is selected as the best-performing one, but the runtime dynamics of the LP (e.g. the rollback frequency) change along the simulation lifetime, we might no longer have all the necessary information to recompute the cost model for the approximate execution mode—in fact, the parameter R cannot be estimated when running in precise mode. To overcome this problem, we periodically re-execute the sequence of the above two steps along the simulation lifetime—hence re-using again the approximate technique for a while. We are therefore able to select again the best rollback mode after all the parameters required for the model-based decision are available.

To also account for changes in the simulation dynamics at runtime, we employ an exponential moving average to update the estimations of all the model variables. In our experiments, we have set the constant smoothing factor to 0.125, but this value should be fine-tuned depending on the velocity at which the model outcome changes.

To reduce the burden of obtaining granularity measures (δ_s, δ_r, δ_c) while the simulation execution proceeds, we suggest relying on clock cycle counts rather than wall-clock time seconds. The availability of such counts is nowadays quite widespread on off-the-shelf computer architectures, e.g. relying on the Time Stamp Counter accessed via the `rdtscp` instruction on x86 architectures, or the processor Cycle Counter accessed via the `PMCCNTR` register available since ARMv6 on ARM architectures.

We finally note that the immediate consequence of the way we instantiate the overhead models is that our autonomic policy is local to every LP. In particular, at any given wall-clock time instant, a subset of the LPs can run in precise mode while others are in approximate mode. This is a desirable property of our autonomic policy because it allows accounting for imbalanced models, in which state reconstruction approximation can differ depending on the dynamics of each single simulation model. This approach indirectly allows for different execution phases in the simulation run, where a single LP can initially benefit, e.g., from an approximate execution, while later it is better supported by the precise mode.

6.3 Experimental Assessment

This section presents an experimental study to evaluate our approximate rollback technique. We first describe the computer platforms used for the experiments. Then we provide details of the simulation applications used as a test bed and their results.

6.3.1 Computing Platforms

We performed the simulations related to our experimental analysis using three different bare-metal architectures. These three architectures were chosen as representatives of different types of computing nodes that can support high-performance simulations. The first is a Dell Precision 5820 Tower *workstation*, equipped with an Intel® Core™ i9-10900X CPU @ 3.70GHz, consisting of 10 physical cores and 20 hyperthreads, with 16 GB of RAM. The second is a *server* machine equipped with two Intel® Xeon™ e5-2699v4 processors @ 2 .0 GHz, each consisting of 22 physical cores and 44 hyperthreads, for a total of 44 physical cores and 88 hyperthreads. It has 256 GB of RAM. The third architecture is a computing node of a supercomputer (belonging to Cineca's Galileo 100 cluster), equipped with two Intel® CascadeLake™ 8260 processors @ 2.4 GHz, each equipped with 24 physical cores, for a total of 48 physical cores (hyperthreads are disabled). The *galileo* machine is equipped with 384 GB of RAM.

6.3.2 Test-bed Applications

We have modified the TBC model described in Section 1.2 to explicitly take advantage of approximate rollbacks during speculative execution. In our reworking of this model, individuals are implemented according to the agent-based modelling specification described in [112]. This way, agents belonging to different states are mapped to different hash tables, and only a subset of the hash tables belongs to the core memory. In particular, we have identified healthy and recovered agents as core-memory agents. These agents are always accurately restored in the approximate rollback operation. Less-relevant agents, i.e. those belonging to the infected, under treatment and sick states, are approximate. The core retains only the corresponding counter for them, but no individual-specific information (e.g., age, sex) is checkpointed and restored in the approximate rollback. The loss of this information on rollback and its stochastic reconstruction via CF leads to a deviation of population characteristics (such as the percentage of individuals within specific age ranges, the percentage of outsiders and the percentage of individuals with other diseases) from the values set at the start of the simulation. In other words, for a given simulation run, an approximate rollback leads to a slight variation in the sample of individuals moving through the area, which can lead to changes in the disease statistics.

We configured the TBC epidemiological model to conduct a micro-simulation of a medium-sized metropolitan city. Specifically, we used 16,384 LPs, each covering a square region of $5.8m^2$, for a total simulated area of $96km^2$. The total number of agents managed by the model is 1.6 million. The agents move randomly around the city area so that each LP handles, on average, 100 agents—a population density of

16,000 people per square kilometre. This configuration resembles the data for the city of Barcelona in 2019. At the start of the simulation, 95.59% of the population is healthy, 4.28% is infected, 0.12% is cured, and the remaining 0.01% are sick or untreated people. Despite the significantly small number of infected persons, the agents are externally dynamic. Their frequent movement leads the simulated environment to a pandemic, with clear recurring phases of peak infection and low disease spread. We simulated a total of 10,000 days of evolution in the SIR model.

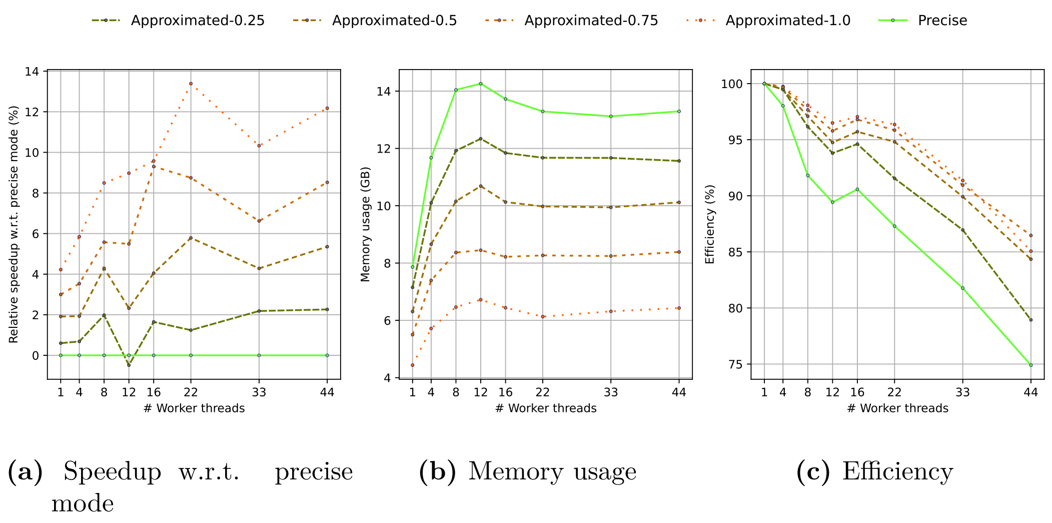
In our experiments with the TBC model, we also consider manual switches among the execution modes. This means we have coded manual policies to determine when to rely on approximate or precise execution. This allows us to compare the results obtained using the autonomic policy discussed in Section 6.2 against the results that could be obtained by exploiting domain knowledge from the modeller. With the manual policy, the switches from the approximate to the precise phase (and vice versa) take place according to a threshold related to the number of agents in the sick state. This choice is because if the number of sick agents increases, their impact is more easily captured by a stochastic configuration. Conversely, in the case of a small number of agents in this state, their precise characterisation is more relevant for studying the evolution of contagion.

To perform the manual switch we used two different configurations. In the first one (referred to as *manual A* in the plots), transitions from the precise to the approximate execution mode take place if more than half of the residing agents in an LP belong to the core memory. Conversely, in the second (referred to as *manual B* in the plots), transitions from the precise to the approximate execution mode take place if less than half of the residing agents in an LP belong to the core memory.

6.3.3 Results

The results we present in this section are averaged, for each point, over 30 different runs using different seeds. The checkpoint interval χ is set to the optimal value in all the configurations according to the performance models we discussed in Section 6.2.

As it will be shown, the results curves have a dynamic and non-linear trend.



(a) Speedup w.r.t. precise mode **(b)** Memory usage **(c)** Efficiency

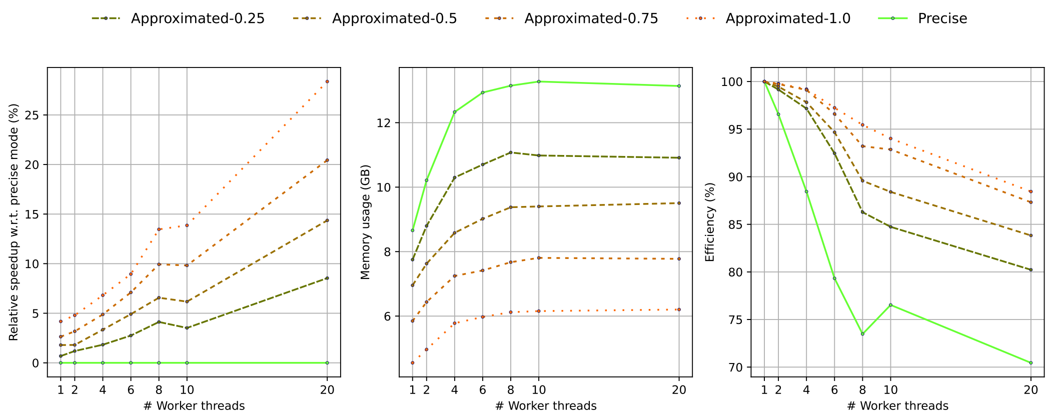
Figure 6.1. PHOLD results on *server*

This phenomenon is related to the hardware configuration of the used machines. In particular, in the case of the *server* and *galileo* architectures, the processors use an interconnect between cores based on two loops. As soon as the execution begins to use the second ring, a fluctuation in results is observed, which is slowly amortised. This phenomenon is related to the increase in the hardware synchronisation cost for enabling the execution of cache coherency protocols between cores. The high memory pressure of optimistic simulations exacerbates this effect. For all the runs that have been carried out on *server* and *galileo* hyperthreading was turned off. Hence no phenomena related to hardware contention on individual cores are observed.

Conversely, in the case of the *workstation* platform, the results appear less noisy. This is because the single processor has a mesh interconnect [21], designed for more general workloads, which amortises the cost of cache coherency protocols. In these settings, results were also collected with hyperthreading active.

Figures 6.1, 6.2 and 6.3 show the results for the different configurations of the PHold Memory model obtained on the three different computing infrastructures used for the experimentation. The shown configurations report the fraction of the LP state which is considered non-core. In the rollback phase, this fraction is subject to reconstruction via CF.

As it can be seen from the results, the approximate rollback technique is able to provide performance improvements over the precise rollback mode (see Fig-

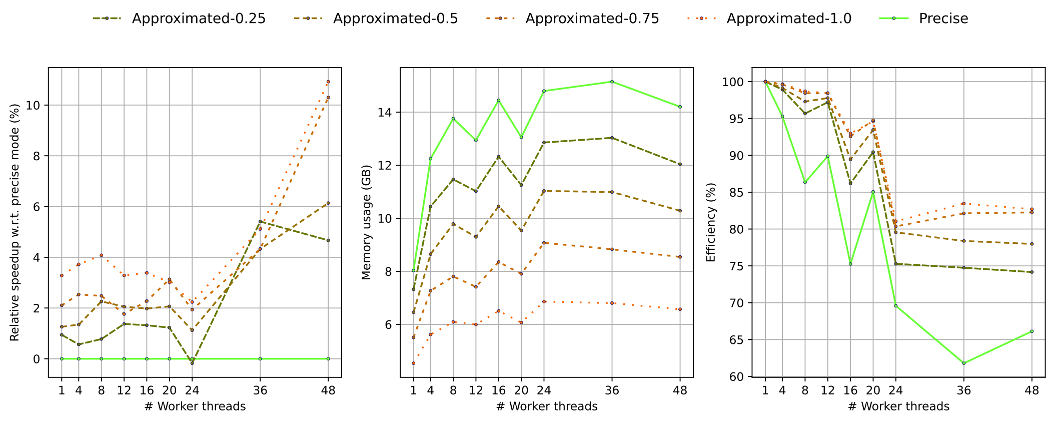


(a) Speedup w.r.t. precise mode **(b)** Memory usage **(c)** Efficiency

Figure 6.2. PHOLD results on *workstation*

ures 6.1a, 6.2a, 6.3a). Also, the performance gain provided by the approximate rollback technique tends to increase along two directions: 1) when the fraction of the LP state that can be rolled back through approximation increases and 2) when the number of threads in the underlying PDES engine increases.

These observations outline how the approximate rollback technique introduces a new dimension—based on CF combined with partial checkpointing including the core state portion—which can (a) permit the exploitation of performance tradeoffs not explorable via other literature techniques, and (b) provide a reduced impact of state saving and/or rollback operations on the critical path of the forward execution, which can generate the favourable throttling effects identified by the seminal work in [119]. As for the latter assertion, a faster state saving and/or rollback technique provides more effective support for a better mutual alignment of the LPs along virtual time (in fact, none of them delays too much their evolution along virtual time when either taking a checkpoint or rolling back), disfavouring the increase of the incidence of rollback. With PHold Memory, this is actually offered by the approximate technique. In fact, the experimental data (see Figures 6.1c, 6.2c and 6.3c) show how increasing the number of threads running the application—hence increasing the degree of actual parallelism—yields to a more limited decrease of the efficiency of the simulation run when the approximate rollback technique is



(a) Speedup w.r.t. precise mode **(b)** Memory usage **(c)** Efficiency

Figure 6.3. PHOLD results on *galileo*

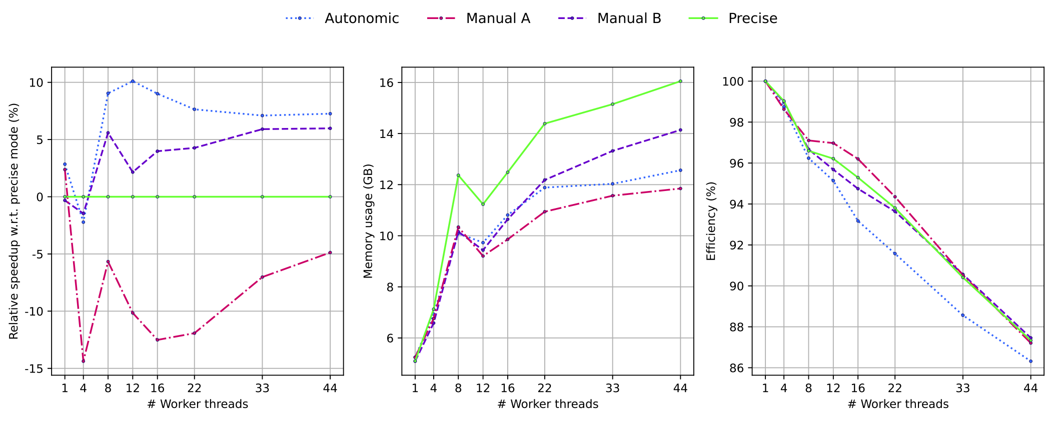
used[2].

It is also evident (see Figures 6.1b, 6.2b, 6.3b) that approximate rollbacks allow significant memory savings. This can additionally favour the speed of the execution because of the effects related to better locality and cache exploitation. In the literature, these have been shown to be a first-level aspect for performance improvement in speculative PDES [40, 14, 160, 89, 101].

As for the TBC model, its results outline additional interesting aspects. In particular, they allow us to compare the autonomic selection of the best-suited rollback mode (approximate vs precise) to both (i) the traditional precise mode and (ii) the “manual” mode, where the programmer forces in what parts of the simulation execution approximate rollbacks should be activated.

The performance data with TBC are shown in Figures 6.4, 6.5, and 6.6, where we see how our autonomic proposal can provide improvements in the execution speed (up to 10%) even under the scenario of non-relevant rollback incidence. In fact, the efficiency of TBC runs mostly keeps above 90%, and is anyhow higher than 85% even for larger thread counts—in general, this tends to reduce the possibilities of optimisation at the level of the checkpointing and state restoration layer. At the same time, the advantages of the autonomic policy also appear when comparing it

[2]We recall that the efficiency is the ratio between the number of committed events and the total number of processed events, including the ones that are eventually rolled back.



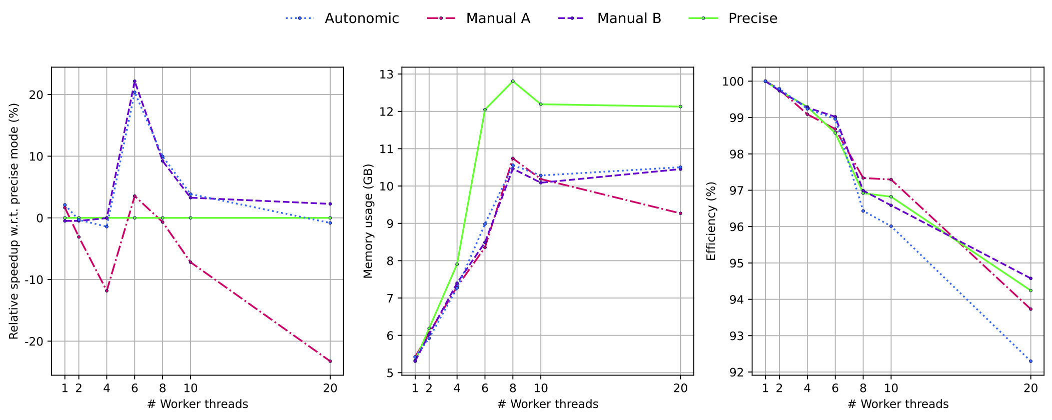
(a) Speedup w.r.t. precise mode **(b)** Memory footprint **(c)** Efficiency

Figure 6.4. TBC results for the server

with the manual strategies. One of these strategies shows reasonable performance improvement over the precise rollback mode, while the other shows definitely worse performance. The autonomic policy prevents this performance degradation at all, avoiding that specific decisions on when (and for what LP) to keep the approximate mode active made by the programmer can actually create problems in performance.

The need for mixing approximate and precise rollbacks (along time and across different objects) according to the autonomic choice (hence avoiding the exclusive usage of approximate rollbacks) is also motivated by the results in Figure 6.7. They show the distribution of simulated agents with respect to LPs. As can be observed, this distribution is almost uniform, except for the objects at the edge. This phenomenon is related to the mobility of the agents, who choose their target region among the adjacent objects. As shown in Figure 6.8, this behaviour causes some workload imbalance (between 1% and 4%). This figure highlights the stripes of the simulated region assigned to each worker thread of the simulation platform.

This imbalance, however, is also observable in some central stripes of the simulated region, as a side effect of the imbalance at the edge and as a result of the movement of the agents along the simulation. Given that in TBC the efficiency is higher compared to PHold Memory, and the cost for the execution of CF is greater—since more complex tasks need to be carried out in the TBC model—for LPs with a bit higher frequency of rollback, it can result in more effective to setup



(a) Speedup w.r.t. precise mode **(b)** Memory footprint **(c)** Efficiency

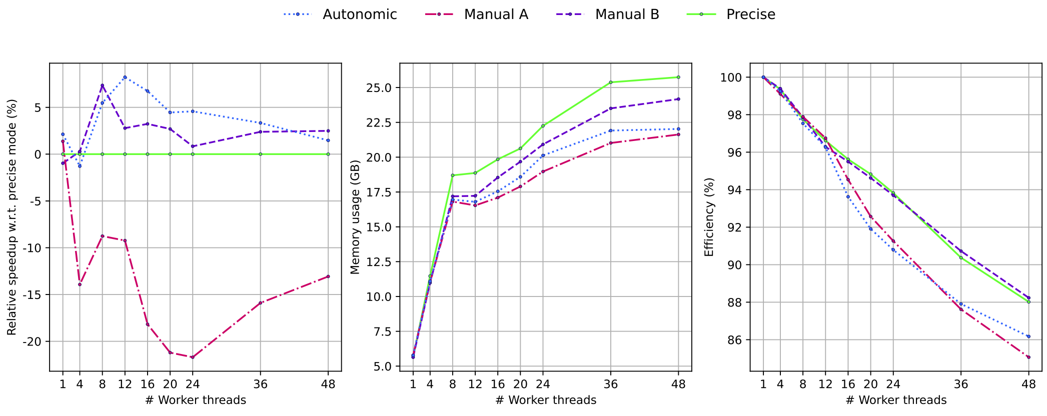
Figure 6.5. TBC results for the workstation

the precise rollback mode. This choice is actually made by the autonomic policy. In fact, as shown in Figure 6.9, LPs that are at the edge of a region of space—which are the ones more prone to receiving straggler event messages—are those that spend the most time in precise mode when using the autonomic policy.

As for memory usage, the mixed exploitation of approximate and precise rollbacks occurring with the autonomic policy makes the amount of used memory a bit closer to the precise mode with respect to what we noted for the PHold Memory model. However, there is no evident negative impact on the final performance.

Concerning the accuracy of the results, in Figures 6.10 and 6.11, we show the trend of the global simulation state for the 10,000-day scenario that was simulated. As it can be seen, the model run in precise mode shows the clear pandemic trend of the infection, characterised by sequences of infection waves.

If the window of interest is short (Figure 6.10), we observe that all configurations can provide a comparable trend in which the succession of two waves of contagion is evident. The contagion bell-distributions have different shapes and durations, which is understandable, given the approximation. Interestingly, the manual approach, in which the model's programmer determines when to switch from approximate to precise execution, can provide the closest results to the precise model. This is understandable since the model developer is the one who has the most knowledge of the application domain and can exploit the model's best dynamics. However, according



(a) Speedup w.r.t. precise mode **(b)** Memory footprint **(c)** Efficiency

Figure 6.6. TBC results for the supercomputer node

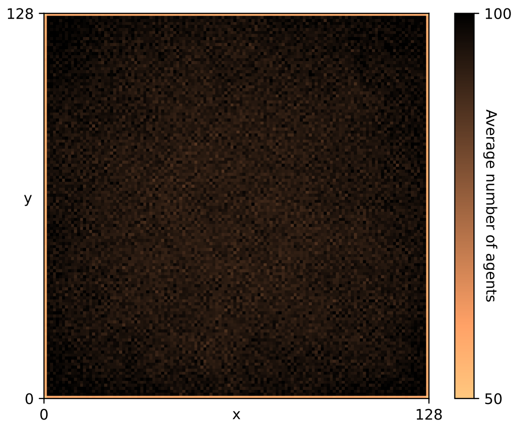
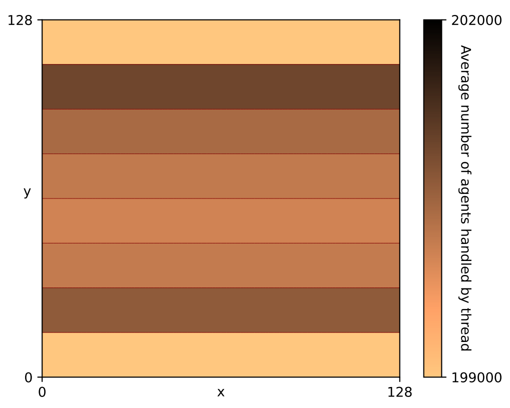


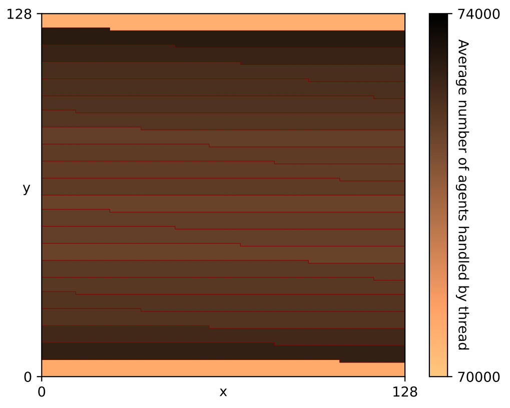
Figure 6.7. The number of agents present in average in the TBC model cells

to Figure 6.4a, such knowledge not necessarily enables a significant improvement in simulation performance, as the execution dynamics may be adverse.

When the window of interest for the simulation is larger (Figure 6.11), the execution mode based on the autonomic policy is still able to capture the simulation dynamics of the precise mode execution. This is because (also according to the above discussion) it excludes excessive negative effects on the simulation precision caused by LPs which would tend to rely on approximate state reconstruction too frequently. At the same time, the autonomic policy avoids excessive inconsistencies that would be caused by less effective manual settings imposed by the programmer for the usage of the approximate rollback technique. This is because the autonomic

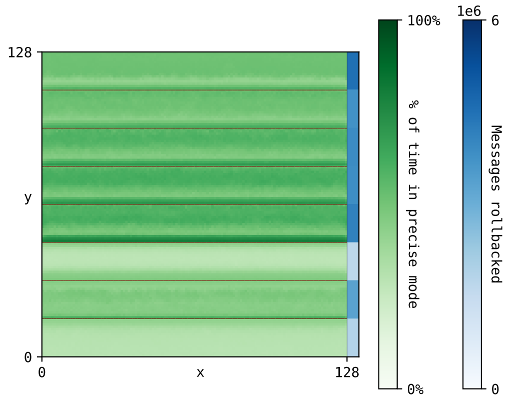


(a) 8 threads: ∼ 1% unbalance

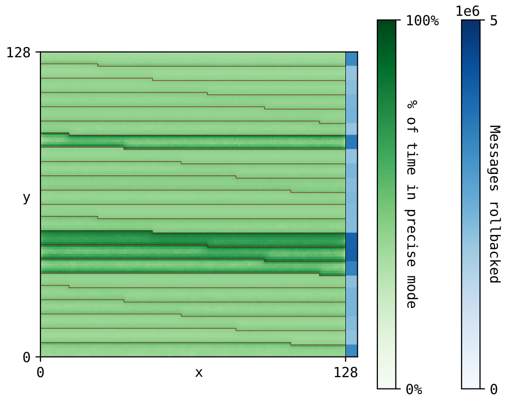


(b) 22 threads: ∼ 4% unbalance

Figure 6.8. Average agents load partitioned on processing threads



(a) 8 worker threads

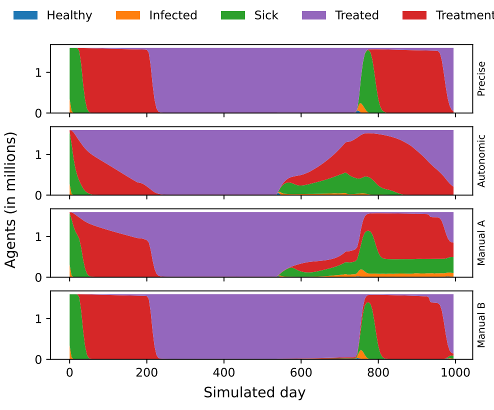


(b) 22 worker threads

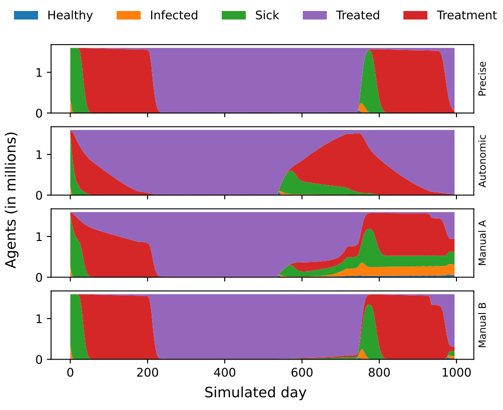
Figure 6.9. Time spent per LP in approximated/precise mode with the autonomic policy

policy captures when the impact of the usage of CF tends to become excessive.

Overall, approximate rollbacks—and their employment in combination with the model-based autonomic policy for the selection and configuration of the best-suited rollback mode—appear to be an innovative effective solution.

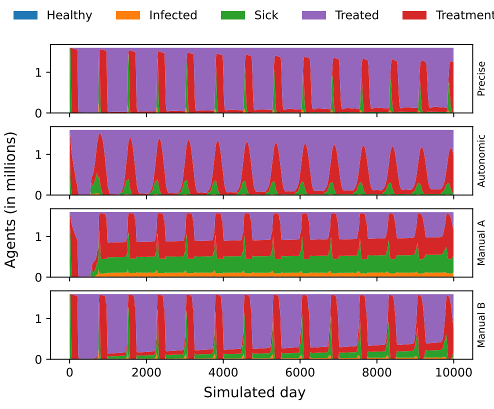


(a) 22 cores

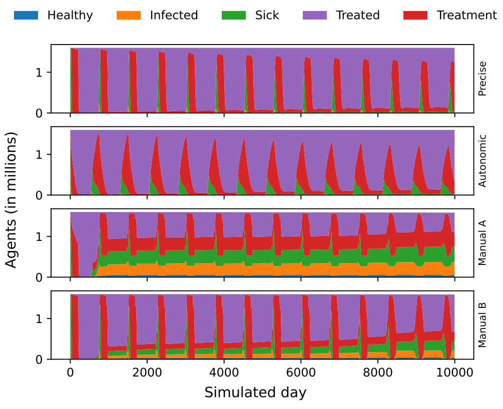


(b) 44 cores

Figure 6.10. TBC precision: 1000 simulated days.



(a) 22 cores



(b) 44 cores

Figure 6.11. TBC precision: 10000 simulated days.

Chapter 7

Practical Tie-Breaking

Simultaneous events [58] are a significant problem for discrete-event simulation (DES). In the physical world, the occurrence of simultaneous events is something that the physical system can handle autonomously by its very nature. When attempting to translate the workings of the physical world into a simulation model, the modeler finds itself having to handle simultaneous events explicitly.

If simultaneous events are not additive and commutative, processing them in a different order may lead to different simulation results. In some cases, the choice of an incorrect order may be a source of risk because the model may incur error conditions that cause it to fail. This concept is well understood in the parallel/distributed simulation community [93], but in the case of simultaneous events, such error phenomena can also occur in purely sequential simulations.

The only way to ensure that a simulation is correct in the case of simultaneous events is to entrust the modeler, through the use of a *tie-breaking* function, with the task of finding a correct ordering for the invariants of the portion of the world one is trying to simulate. For example, in [61], the authors theorize the need to handle *sets of events* at the model level. By providing the event dispatcher with a set, it is possible to ask the model to deal with them in a manner consistent with its characteristics.

In the case of optimistic parallel/distributed simulations [59], this approach may be hard to implement. The speculative nature of forward execution may still lead to handling sets of events when they are not fully formed because an antimessage could be received before the corresponding positive one. However, in the case of sequential simulation, this method may be sufficient to handle correctness, except for the case when some events in a set with the same timestamp generate a new simultaneous event. From a practical point of view, this interaction pattern is important, as it allows implementing *sensing capabilities* in the models, which are relevant, e.g., for agent-based simulation [1]. This is so relevant that, a seminal Time-Warp work introduced the concept of *query messages* to solve the dichotomy between event sets and the need to access portions of the simulation state in read-only [62].

The scenario in which the modeler has provided a tie-breaking function capable of correctly handling simultaneous events may be ideal. The assumption that such a function is available can only materialize when the simulation model is stable, complete, and correct. Indeed, in the life cycle of a simulation model [107, 138], the modeler may need to delay the implementation of such a function as they are focused on developing the core dynamics of the model. Similarly, if the model is subject to evolutionary maintenance, or if the model is incorporated into a simulation of simulations in an attempt to reuse existing models in larger models, it is possible that the coding of the tie-breaking function is no longer correct and needs to be modified, which may be time-consuming. In this case, the modeler may decide to suspend the use of the current tie-breaking function in order to redesign it.

In this dynamic view of models, a substantial difference materializes between parallel and sequential execution of the same model. Indeed, sequential models, even in the presence of simultaneous events, may have the characteristic of being able to exhibit *deterministic* executions. Conversely, when the model is executed on parallel/distributed architectures, two different executions may lead to different results if simultaneous events are not handled properly. This is also true when the same seed is used to configure pseudo-random number generators for stochastic simulations.

This situation is clearly problematic. The modeler may need to execute a non-minimal number of identically-configured runs to compare simulation results for debugging or performance evaluation purposes. For parallel/distributed simulations, developers of runtime environments are therefore obliged to explicitly handle simul-

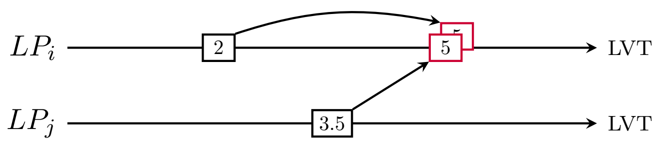


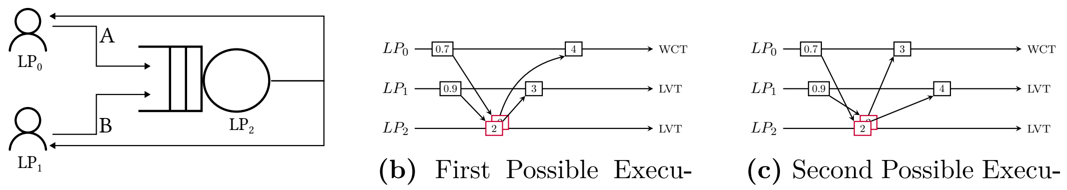
Figure 7.1. Lack of Total Ordering: the two events at logical time 5 should "happen" at the same time, therefore it is unspecified which of the two should be dequeued first from the FES.

taneous events even if the modeler has not provided a model-oriented tie-breaking function.

In this Chapter, we present a practical tie-breaking function that guarantees the reproducibility of executions. The major contribution of our proposal lies in the fact that, unlike the solutions presented in the literature, our approach also allows for domain information to be considered transparently in the simulation model. Thus, the ordering provided by our approach is such that indecision between two simultaneous events only occurs when, even at the application level, the model cannot distinguish between two different events.

In particular, as we shall show, when compared to sequential execution, our proposal is such that any indecision on the ordering between two events in a parallel/distributed execution only occurs when the sequential model would also have to decide on ordering two perfectly-identical events. Moreover, our proposal provides the same ordering of events if the model runs sequentially or in parallel, even if optimistic approaches are adopted.

Therefore, using our approach, deterministic executions can be achieved even in the case of parallel/distributed executions in the absence of tie-breaking functions at the model level. Our approach considers the complex instance of concurrent simulation based on optimistic synchronisation. In this context, our proposal may suffer from incomplete information related to the delay in receiving certain events. However, this phenomenon is inherent in the concept of speculative simulation *aggressiveness* and can therefore be solved by techniques already present in the literature [62].



(a) Simulated Scenario. **(b)** First Possible Execution. **(c)** Second Possible Execution.

Figure 7.2. Inconsistent Runs in a Parallel/Distributed Simulation.

7.1 Problem Statement

In a DES, whether sequential or parallel, the simulation state is updated by the execution of events that are the *atomic unit of processing*. An event has an all-or-nothing nature: it must be executed entirely or not at all.

Therefore, the simulation state undergoes a set of updates each time an event is processed. As already formalized in Chapter 2, for a simulation in a certain state s_k, the execution of an event e_k at simulation time t_k can be seen as the application of a transition function δ that produces a state update, i.e.:

$$\delta(s_k, e_k) = (s_{k+1}, (e_a, e_b, e_c, \dots)) \tag{7.1}$$

In this reformulation we will omit the events $(e_a, e_b, e_c, \dots)$ generated by the transition function since they aren't needed. We will also assume that s_0 is the state induced by the initialisation transition – the initial simulation state. Then, we can iteratively apply the function δ to obtain a sequence of processed events that will eventually lead to a final state:

$$s^* = \delta(e_n, \dots, \delta(e_2, \delta(e_1, \delta(e_0, s_0)))) \tag{7.2}$$

The order in which events are executed is decisive for attaining a terminal state of the simulation of interest. Given a sequence of events $e_1, e_2, \dots, e_n$, reversing the execution of two events e_i, e_j with $i \neq j$ can cause an alteration in the trajectory of the simulation. Indeed, if the events e_i and e_j are not *commutative with respect to the function* f, i.e. if $\delta(e_j, \delta(e_i, s_h)) \neq \delta(e_i, \delta(e_j, s_h))$, it is easy to observe that after a sequence of m events we have that:

$$\begin{aligned} s_m &= \delta(e_m, \ldots, \delta(e_j, \delta(e_i, \ldots, \delta(e_1, \delta(e_0, s_0))))) \neq \\ s'_m &= \delta(e_m, \ldots, \delta(e_i, \delta(e_j, \ldots, \delta(e_1, \delta(e_0, s_0))))) \end{aligned} \tag{7.3}$$

The classical approach to DES requires the presence of a data structure called the *future event set* (FES), which is typically implemented as a priority queue. Therefore, a query can be performed on the FES to extract the highest-priority event. Due to the temporal nature of simulations, the priority can be immediately associated with the minimum time in the FES.

Simultaneity of events refers to the context in which, at a certain timestamp $\bar{t}$, there are at least two events to be processed. Therefore, the presence of e_1 and e_2 associated respectively with timestamps $t_1 = t_2 = \bar{t}$ poses a problem concerning the extraction operation from the FES, since e_1 and e_2 do not have a *strict* total ordering with respect to logical time. This scenario is shown in Figure 7.1. We consider the problem of *tie-breaking* as the need to reconstruct a deterministic order over all events, even when two or more are associated with the same timestamp. Formally, this problem can be defined as follows:

Definition 7.1.1 (Tie-Breaking Oracle)**.** The operation of extracting the next event from the Future Event Set is entrusted to an oracle O which determines the next event e to be executed to obtain a final simulation state s^* consistent with the behaviour of the real-world system being simulated.

It is evident that it is impossible to disregard the model's dynamics to define the oracle O. This is the reason why it has been recognized in the literature [61] that the modeler must provide a *tie-breaking* function allowing to correctly implement O even when the events are not commutative. However, this oracle O may not be available for the above reasons related to the simulation development cycle or model evolution/composition. Nevertheless, from the point of view of model execution support, it is necessary to define *some* deterministic total ordering on the events in the FES.

We note that, as a necessary condition, the output of a correct simulation, in which tie-breaks are handled, must be deterministic. In this scenario, sequential simulations suffer less than parallel/distributed simulations. Indeed, determinism is a common property of FES implementations; however, if the data structure employed for the FES is randomized, then even a serial simulation may typically be non-deterministic. Nevertheless, the deterministic ordering occurring during a sequential simulation might not necessarily represent the characteristics of the physical system being investigated.

Therefore, in the case of sequential simulation, Definition 7.1.1 can be relaxed by introducing the concept of *partial oracle P* according to the following

Definition 7.1.2 (Partial Tie-Breaking Oracle)**.** The operation of extracting the next event from the Future Event Set is entrusted to a partial oracle P, which determines the next event e to be executed so as to obtain a sequence of events $\{e_0, e_1, \dots, e_n\}$ to reach a final state of simulation $\widetilde{s}^*$ such that $e_0 \prec e_2 \prec \dots \prec e_n$, where the $\prec$ operator behaves according to the total order defined by the event timestamps plus some FES implementation dependant criteria to break ties deterministically.

Clearly, it is likely that $\widetilde{s}^* \neq s^*$—hence the partiality of the oracle P. Anyhow, Definition 7.1.2 has an important property related to the reproducibility and repeatability of executions. Since the oracle P chooses a sequence of events in a deterministic manner, two different executions configured in exactly the same way will lead to the same final state of simulation $\widetilde{s}^*$. Any pseudo-random number generator seeds in the case of stochastic models must also be included in the model configuration.

We argue that, in the case of parallel/distributed simulations, the realisation of the partial oracle P, according to Definition 7.1.2, is complicated and also not useful. In fact, the behaviour of P depends on the characteristics of the sequential simulation *implementation*; in other words, it is biased in a way not representative of the model dynamics. Still, we are left with the problem of parallel non-deterministic simulations. Indeed, two different parallel executions may lead to the scenario shown in Figure 7.2. Here, we are depicting an extremely simple scenario where two users (modeled as two distinct LPs) contact the same service over a network. Both requests arrive at the same timestamp ($t = 2$). Assuming that the server has a delay of 1 virtual second to process a request, the order according to which they are received affects the response time. Imagine that user A can tolerate a delay of up to 2 seconds, while user B can tolerate a delay of up to 3 seconds. In the case of the scenario in Figure 7.2b, the outcome of the simulation would be that the configuration does not respect the model's invariants, while in the execution in Figure 7.2c, the solution would be acceptable. To exemplify the partial oracle strategy's bias, we note that *one* serial execution engine might consistently and deterministically generate the output shown in Figure 7.2b, while *another* serial engine could consistently and deterministically replicate the situation depicted in Figure 7.2c.

The processing and commit order of events depends on many factors *external to* the simulation, such as scheduling dynamics at the operating system level or network latencies. Therefore, in order to be able to guarantee the reproducibility of executions even in parallel/distributed simulations, it is necessary to find a strategy that does not rely on local implementation properties such as the partial oracle P of Definition 7.1.2.

7.2 Practical Tie-Breaking Technique

Our proposal is born from the intuition that, if two events are *indistinguishable* by the simulation model, they can be executed in any order without affecting the final state of the simulation. The same concept also holds for sequences of indistinguishable events.

To better formulate our intuition, we provide a definition of ties that is compatible with the logical framework discussed in Section 7.1. Let e be a generic event injected in the system, associated with the timestamp t_e at which the event must be executed. A tie follows the definition below.

Definition 7.2.1 (Scheduling Equivalence)**.** Two events e_1 and e_2 are *scheduling equivalent*, namely $e_1 \sim e_2$, if $t_{e_1} = t_{e_2}$.

The goal of the Partial Oracle P discussed in Definition 7.1.2 is therefore to *enhance* the scheduling equivalence from Definition 7.2.1 by enforcing its own ordering $\prec$ that comes from the rules enforced by P. The body of work discussed in Section 4.7 has effectively tried to extend this notion of scheduling equivalence by defining an ordering $\prec_P$ that can be applied to a set of events e_i such that $t_{e_i} = t_{e_j} \forall i \neq j$.

The simplest definition of $\prec_P$ that has been considered in the early literature and in various early implementations of PDES runtime environments extends the previous definition of events, in various forms. In particular, we can consider an event e as a tuple $e = \langle t_e, c_e, s_e, d_e \rangle$, where c is the *event class* (also referred to as event type), s is the sender of the event and d is its destination. We can practically assume that we can build some (lexicographic) ordering on c_e, s_e, and d_e. Indeed, in many implementations, these elements of the tuple are already numbers, but it is straightforward to define some mapping to $\mathbb{N}$ or $\mathbb{R}$ that can define an ordering over the values. Then, we can enhance Definition 7.2.1 as follows:

Definition 7.2.2 (Enhanced Scheduling Equivalence)**.** Two events e_1 and e_2 are *scheduling equivalent*, namely $e_1 \sim e_2$, if $t_{e_1} = t_{e_2}$, $c_{e_1} = c_{e_2}$, $s_{e_1} = s_{e_2}$, $d_{e_1} = d_{e_2}$.

There are two important implications of Definition 7.2.2. First, such a definition of equivalence may create a bias with respect to the choices that the tie-breaking Oracle O of Definition 7.1.1 might make. Indeed, regardless of the ordering of the elements, P could choose differently from O since, by definition, P is unaware of the dynamics of the model. However, for our practical purposes, this bias might be tolerable: P is not intended to be a *correct* replacement of O, but an *acceptable approximation* if O is not available.

The strongest implication, however, is that $\prec_P$ defined in accordance with Definition 7.2.2 is a weak total order. Therefore, using $\prec_P$ defined in this way does not solve the problem at all since it is still possible to find two events $e_1 \sim e_2$ such that the scheduler of the runtime environment is unable to make a deterministic choice, even if their processing order could impact the simulation trajectory. Since the goal of our approach is to support reproducibility and replicability, Definition 7.2.2 is not sufficient.

There are two important implications of Definition 7.2.2. First, it is evident that such a definition of equivalence may create a bias with respect to the choices that the tie-breaking Oracle O of Definition 7.2.2 might make. Indeed, regardless of the fact that there is an ordering on the elements, this could be different from that of O since, by definition, the enhanced scheduling equivalence is unaware of the dynamics of the model. However, for our practical purposes, this bias might be tolerable: the provided order is not intended to be a *correct* replacement of O, but an *acceptable approximation* if O is not available.

As mentioned, several of the works discussed in Section 4.7 have addressed this problem by implicitly providing an extension of Definition 7.2.2 that allows this problem to be addressed. In particular, it is possible to construct an ordering $\prec_P$ that imposes a deterministic ordering by extending the definition of the event e. Indeed, we can consider an event e as a tuple $e = \langle t_e, c_e, s_e, d_e, b_e \rangle$, where b_e are *arbitrary bits* provided by the model developer that implicitly describes the priority of e over other tying events. Abiding by this definition, we can construct an improved ordering $\prec_P$ leveraging the following definition.

Definition 7.2.3 (User-Defined Enhanced Scheduling Equivalence)**.** Two events e_1 and e_2 are *scheduling equivalent*, namely $e_1 \sim e_2$, if $t_{e_1} = t_{e_2}$, $c_{e_1} = c_{e_2}$, $s_{e_1} = s_{e_2}$, $d_{e_1} = d_{e_2}$, $b_{e_1} = b_{e_2}$.

At first sight, this definition should solve the problem of ties. Indeed, the modeller can set the value of b_e arbitrarily, thus deciding what is the precedence between events. However, in our reference scenario, this strategy is highly objectionable for two reasons. The first concerns transparency towards the modeller: if they are asked to provide additional information to define an Oracle P that succeeds in resolving any remaining ties, the authors of this work then wonder why this strategy is better than explicitly requesting that the Oracle O be realised directly. Indeed, as discussed in [61], the qualities of O are clearly superior to those of P, from a modelling perspective. Reasoning on this aspect, a solution defining an event e as the simpler tuple $e = \langle t_e, b_e \rangle$ may also be a better solution, as it would eliminate the artificial bias introduced by Definition 7.2.2. Moreover, in our reference scenario, we consider the model as an evolving object, so the properties according to which b_e should be valued could easily change. This is therefore not a practical,

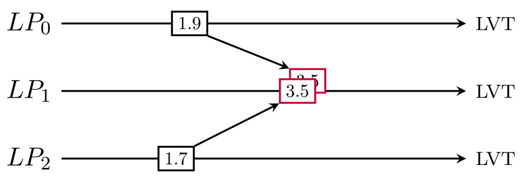


Figure 7.3. Simultaneous events generated by different LPs.

life-cycle-oriented approach to models.

As mentioned, this problem related to transparency towards the modeller was addressed in [85] by deciding to entrust the values of b_e to a pseudo-random number generator with special properties. In this way, it is both possible to explore alternative simulation trajectories and obtain reproducible executions, without bothering the modeler.

We follow an alternative path in our proposal as we reason about indistinguishability between events. Two events are actually indistinguishable if they expose the exact same information to the simulation model. To better formulate this concept, we redefine an event e as the tuple $e = \langle t_e, p_e \rangle$, where p_e is the event's payload, intended as the collection of all the event properties observable by the model. We can then define indistinguishable events as follows.

Definition 7.2.4 (Indistinguishable Events)**.** Two events e_1 and e_2 are indistinguishable, namely $e_1 \overset{?}{\doteq} e_2$, if $t_{e_1} = t_{e_2}$, and $p_{e_1} = p_{e_2}$.

According to this definition, two events are indistinguishable when their timestamps and payloads are bit-wise identical. We do not consider the pair of LPs involved and the event's class as a necessary part of the event payload, as, in the general case, they may not be needed by the model logic. Moreover, if we take into account indistinguishable events, it is not important to make a deterministic choice as to which events execute first on different LPs: indeed, given the concurrent nature of Time Warp, it is sufficient to provide local guarantees—we discuss aspects of cross-LP causality in Section 7.2.1.

By relying on Definition 7.2.4, we can build an ordering $\prec_P$ with interesting properties. First and foremost, we emphasise that this ordering is not a strict total order. At first sight, this may appear to be a limitation, e.g. when compared with the proposal in [85]. However, as discussed above, our ordering is not able to distinguish between two events if they are, indeed, indistinguishable. From a practical point of view, therefore, it is interesting to ask what anomalies at the model level such indistinguishability may entail. Our thesis is that the following property holds.

Property 7.2.1. *Regardless of the order in which a sequence of indistinguishable events is processed, the result of the simulation is unchanged.*

If, for a sequence of events, changing their processing order changes the result of the simulation, that implies that at least a pair of events $e1, e2$ in the sequence are distinguishable by the model. In other words, a model could use the difference in the observed information in $e1, e2$ to order them deterministically. This choice would be part of the perfect oracle O, but in its absence, we'll show that a simulation engine can always take a deterministic, although biased, choice.

Finally, by relying on Definition 7.2.4, we can build a total ordering $\prec_L$ in the following way:

Definition 7.2.5 (Lexicographic Tie-breaking)**.** Given two events e_1 and e_2, $e_1 \prec_L e_2 \Leftrightarrow t_{e_1} \leq t_{e_2} \vee (t_{e_1} = t_{e_2} \wedge p_{e_1} \leq p_{e_2}$

From the point of view of the simulation engine, the payload comparison does not need to understand the semantics of its content, i.e.: a simple bit-wise comparison is sufficient to always break ties. We observe that, if $e_1 \prec_L e_2 \wedge e_2 \prec_L e_1$ if and only if $e_1 \overset{?}{\doteq} e_2$; in other words, our total order is unable to order two events only if they are indistinguishable. This means that $\prec_L$ defines an order of events that deterministically induces the same simulation trajectory.

Our approach also has the valuable property of being independent of the initialisation order of the LPs. On the contrary, employing an ordering based on any scheduling equivalence defined above can result in a scenario where the initialisation order determines the order of tied events scheduled by LPs during initialisation. This makes the outcome of the whole simulation dependent on the order of evaluation of LPs.

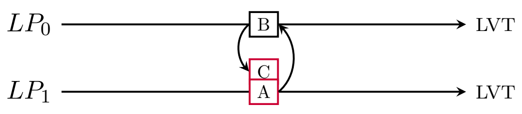


Figure 7.4. Zero-Lookahead Cycle.

Model developers sometimes unknowingly rely on the ordering implicitly guaranteed by some scheduling equivalence property, which can lead to models that subtly depend on this behaviour to function correctly. For instance, in an agent-based model, an agent departure and return may be scheduled at two randomly-sampled times t and $t + \delta t$, respectively. If $\delta t = 0$, this would mean that the logical dependence of the two events is encoded only in their scheduling order. In more complex interactions, this can lead to difficult-to-debug issues where the model works correctly in a sequential simulation but crashes in parallel execution.

By using our tie-breaking strategy, we can ensure that a model functions correctly in both sequential and parallel execution modes. If the sequential simulation that uses our tie-breaking strategy runs correctly, then—assuming that speculative simulation trajectories do not cause irreparable side effects [93]—the parallel execution will also be correct.

7.2.1 Handling Cross-LP Causality

To understand the implications of our tie-breaking strategy, we have to discuss the implications of our approach when cross-LP interactions are observed. There are two cases of interest here. The first is depicted in Figure 7.3, where one LP is the target of two simultaneous events generated by two different LPs.

The tie-breaking methodology we introduced leaves no doubt in this case. If differences between events can be identified, the ordering will be well-defined and reproducible. If the events are indistinguishable, then we can apply Property 7.2.1; therefore the order in which these events are executed is irrelevant: the simulation results will be unchanged.

A more interesting implication of our practical tie-breaking methodology relates to *zero-lookahead cycles*. Let us consider the scenario depicted in Figure 7.4. Here there is a circular causal dependence $A \to B \to C$, which involves two different LPs.

Let us consider the most difficult problem, i.e. the one in which events A and C have empty payloads. We observe that for a correct simulation, we must have that $A \prec_L B$ and $B \prec_L C$. Also, our ordering $\prec_L$ defined according to Definition 7.2.5 is not able to make a choice, i.e.: $A \doteqdot C$, so we must also have $B \doteqdot C$ and $B \doteqdot A$.

According to our defined total order, C would not be a straggler for A. Therefore, C's execution will still be subsequent to A. This property can be generalised, thus stating that, if an LP has executed an event e_1 and subsequently receives an event $e_2 \doteqdot e_1$, the reception of e_2 must not cancel the execution of e_1. Apparently, this can be regarded as contrary to the concept of reproducible and repeatable execution. Once again, thanks to Property 7.2.1, we consider the two executions (e_1 before e_2 and e_2 before e_1) to be perfectly equivalent with respect to the final result of the simulation.

If, on the other hand, the two events are distinguishable, then a deterministic order can be imposed to decide if the execution is consistent or not. The reasoning we have just set out applies to zero-lookahead cycles, and by extension, it can be applied to any form of zero-lookahead events. Therefore, we can exploit the definition of indistinguishability between events to solve the management of zero-lookahead events locally as well.

Interestingly, Property 7.2.1 can also be applied to relax the implementations of the data structures used for the FES. Indeed, should A and C be reprocessed silently due to the receipt of a straggler, it is irrelevant whether they are reprocessed in the order A, C or in the order C, A. This is a strong implication of Property 7.2.1: indeed, should the events be reprocessed in the order C, A we would be faced with a *causality violation that can be ignored.*

7.2.2 Implementation Details

One advantage of our proposed tie-breaking strategy is that it can be evaluated completely locally to the LP, therefore making it suitable for parallel and distributed simulations. Implementing this approach in an existing simulation engine only requires a few changes. In conservative simulations, the scheduling policy needs to be expanded so as to consider the bit-wise comparison of the payloads in case of event ties. In optimistic simulations, it is also necessary to include the extended tie-breaking logic in the straggler detection mechanism. No other significant changes are needed.

In addition, when developing and debugging a model, having the $\prec_L$ ordering of events defined according to Property 7.2.1 can be useful in identifying causality violations and ensuring that the model schedules events only in the future. Although arbitrary, the repeatable and reproducible ordering imposed by our tie-breaking strategy can help the model developer to detect tricky same-timestamp dependency cycles and other causality issues. Moreover, as we highlight in Section 7.2.3, the modeller has a simple way to alter the arbitrariness of the ordering.

One important caveat to note with this tie-breaking method is that in practical implementations, especially in low-level languages like C, the determinism of bit-wise comparisons can be skewed by uninitialised data contained in the event payload. This issue can arise due to several reasons such as the model failing to initialise some members of the event payload, the presence of padding bytes in the event payload structure, or the model writing uninitialised bytes in the event payload.

Although addressing these issues may demand effort from the model developer, there are several straightforward strategies that can be useful for their mitigation. For example, padded data structures can be detected at compile time using GCC's `-Wpadded` flag. During debugging, tools such as Valgrind [90] can be employed to verify whether a tie has been broken due to uninitialised bytes. Programming languages may offer additional alternative solutions to some of the issues mentioned earlier. For example, in C++, a padding-free comparison can be implemented using template meta-programming techniques such as those provided by the Boost.PFR library.

7.2.3 Relations with Other Tie-Breaking Schemes

Breaking ties by comparing event payloads bit-wise is effective but may result in an event order that is not meaningful model-wise. Indeed, as discussed above, modellers are accustomed to the use of event classes to handle potential ties.

However, this problem can be solved easily. Indeed, it is sufficient for the modeller to insert into the model payload the additional information (such as the event class) that it intends to be used to break the ties. By doing so, we can elegantly solve the problem outlined earlier with the agent-based model by assigning two different classes to *depart* and *return* events. Our experience with model development suggests that prioritising specific event classes over others can already solve most of the issues related to tied events.

From this discussion, it is therefore clear that our practical tie-breaking approach is not at odds with other system-level solutions to solve this problem. In fact, it is a generalisation of them. Indeed, any of the strategies based on Properties 7.2.1–7.2.3 (and any proposal discussed in Section 4.7) can be re-implemented within an ordering defined according to our strategy.

However, there are two ways to materialise these relations. The simplest involves realising the change at the runtime environment level: the environment developer can transparently insert within events any information they wish to be considered by the tie-breaking approach we have presented. With this strategy, different properties can be guaranteed transparently to the modeller.

However, we consider this possibility problematic, as it inserts a bias of which the modeller may be unaware, as many of the methods in Chapter 4.7 do. Our strategy is free from bias originated by simulation engine details. If the model ignores the characteristics of our tie-breaking policy, it does not impose a meaningful order on the members of its events payloads. In this case, we experience an unchecked order bias, but at least it is characterized and encoded in how the model generates events. Therefore, the second way is to make the use of specific strategies non-transparent to the modeller. In this way, it is the modeller who will be able to include in the event payload as much information as they deem useful for ordering the events. In this way, it is possible to create simulation environments that are anyhow correct and support repeatable and replicable executions. The model developer still has control over the choices made by the simulator. In this sense, the practicality of our approach is greater, as it allows transparent and non-transparent approaches to be combined in a single implementation.

Table 7.1. Model configurations.

Model	#LPs	Remote events	Committed events
PHOLD	2^{21}	10%	$\sim 527M$
ETIES-easy	65536	50%	$\sim 393M$
ETIES-hard	131072	10%	$\sim 1100M$
TBC	16384	$\sim$ 80%	$\sim 739M$

7.3 Experimental Assessment

Our evaluation focuses on two aspects: the performance impact and the effect on model accuracy when compared to a version of the simulator that treats tied events as part of the same equivalence class, resulting in their execution in any order they are delivered.

The experimental evaluation was run on a machine equipped with two Intel® Xeon™ e5-2699v4 processors @2.0 GHz, each consisting of 22 physical cores and 44 hyperthreads, for a total of 44 physical cores and 88 hyperthreads. It has 256 GB of RAM, even though the maximum size of the resident set utilised by the runs is approximately 42GB.

7.3.1 Testbed Applications

Beyond the benchmarks introduced in Section 1.2, we have also adopted additional models previously presented in [85]. We have evaluated the models using configurations similar to those in [85]. Table 7.1 provides an overview of the four model configurations. Each point in each configuration is the average of 20 runs, with the highest observed coefficient of variation being approximately 0.05, indicating reasonably reliable results.

The *event-ties* (E-TIES) model is a synthetic benchmark that aims to investigate how simulation engines handle large volumes of tied events. It operates in rounds that are triggered every unit of virtual time. In each round, each logical process starts one or more chains of tied events. If an LP receives an event as part of a chain, it can extend it by sending another event or terminate it if a pre-configured length is reached. Additionally, if an LP A terminates a chain and happens to be the first chain initiated in that round by another logical process B, then A

sends an event to B to schedule the next round, effectively closing the chain. This guarantees progress in the model and creates long logical dependence chains that should adequately stress a tie-breaking implementation. In this model, we use the current chain length as the event class, ensuring that tied events are ordered by their position in their respective chains.

We examined two E-TIES configurations with distinct features, as presented in Table 7.1. The *easy* configuration was designed such that in each round, each LP only initiates a single chain of events with a maximum length of two. On the other hand, the *hard* configuration involves each LP producing three chains of length five, which results in far more demanding tie-breaking activities.

For each configuration, we show four different plots. In order from left to right, we have:

1. the event processing time in seconds, which does not include initialisation and finalisation costs;
2. the speedup computed over the corresponding serial configuration;
3. the efficiency computed as the percentage of committed events over the total number of executed events;
4. the cost per event, divided between the actual event processing and event extraction costs. These measures are expressed in nanoseconds, computed using the `rdtsc` instruction of the *x86* instruction set.

We point out that, in ROOT-Sim, both the serial and parallel engines eagerly insert scheduled events in the Future Event Set; therefore, event insertion operations are already included in the event processing costs.

7.3.2 Experimental Results

The results for the PHold model are shown in Figure 7.5. It is clear that there is no significant difference in behaviour between the two configurations. As expected, the tie-breaking logic is not stressed to a measurable point, even though the number of LPs is non-minimal. This is because timestamp deltas are drawn from an exponential distribution, and the randomisation of timestamps across 64-bit floating point

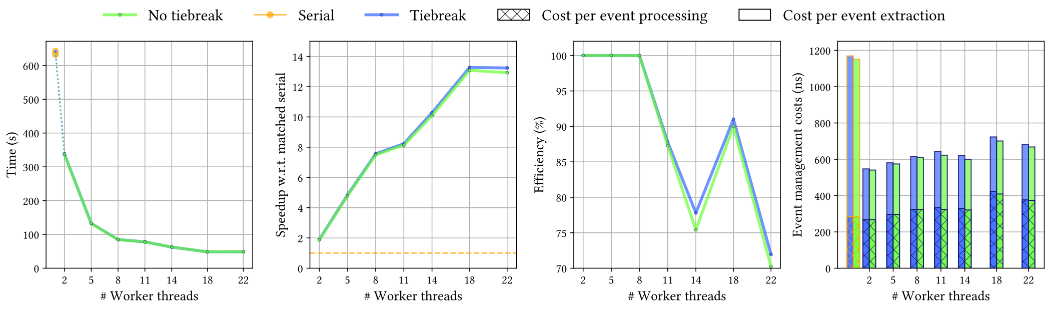


Figure 7.5. Results for *PHold* model

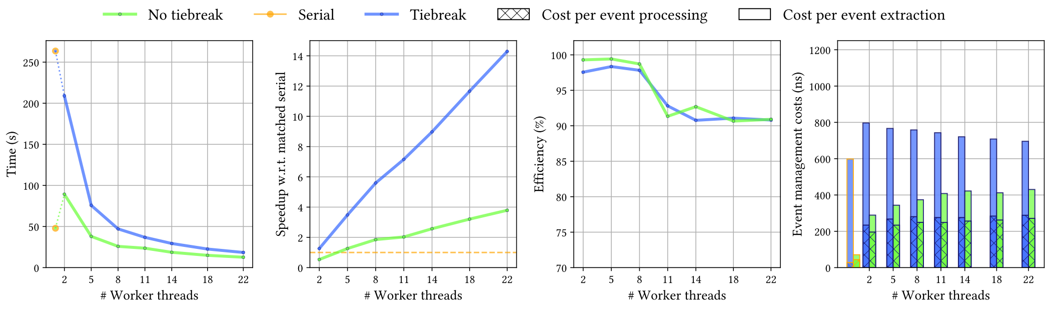


Figure 7.6. Results for *event-ties* model, easy configuration

samples is sufficient to avoid event ties. In addition, we should note that even with our tie-breaking policy, given the nature of PHold, tied events would be considered logically concurrent. Therefore, this model may not be the best choice to evaluate the characteristic of our tie-breaking policy. We want to note that the serial runtime shows unusually high event extraction costs, which is unsurprising given that the ROOT-Sim serial engine is optimised for smaller-scale models.

In contrast, the E-TIES model presents a more diverse trend, even in its *easy* configuration (Figure 7.6), highlighting the cost of using a tie-breaking strategy. Ignoring ties provides a significant advantage, especially at low core counts, yet both configurations exhibit good performance. Nevertheless, the overall speedup indicates a favourable trend for the tie-breaking configuration. Possibly, the configuration without tie-breaks cannot scale further because the parallelism of the model is already being vastly exploited.

By analysing the costs, we can draw two noteworthy observations. First, the serial execution of E-TIES *easy* exhibits lower event processing costs than PHold, despite its more complex logic. This could be attributed to the fact that, in execu-

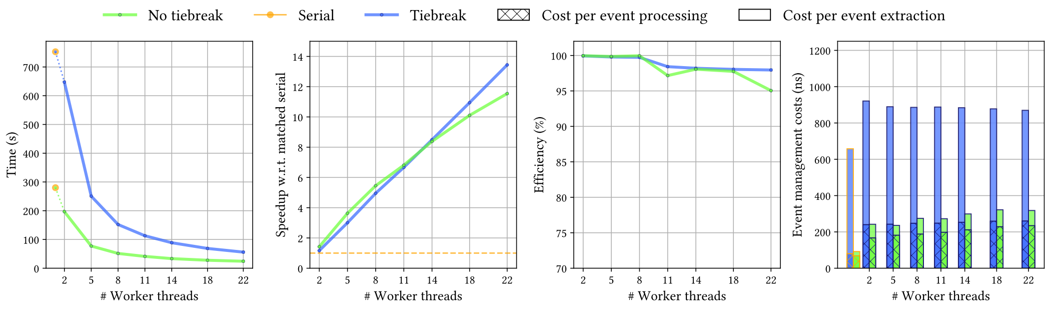


Figure 7.7. Results for *event-ties* model, hard configuration

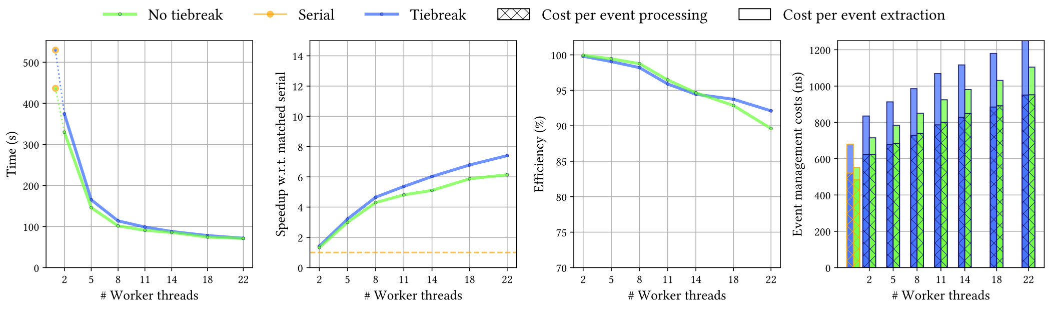


Figure 7.8. Results for *TBC* model

tions of E-TIES without the tie-breaking logic, event insertion in the heap usually requires few operations as the events end up placed at the end of the heap. Likewise, in tie-breaking executions, the chain position identifier is used as the event type. Most newly-scheduled chain events do not happen before those in the queue, resulting in event insertions that require only a limited number of operations. Conversely, in PHold, timestamps are sampled from an exponential distribution; therefore, they are better distributed across an interval of logical time. As a result, logarithmic heap costs are observed since heap bubbling operations are necessary during insertions.

The other interesting fact is the vast difference in event extraction costs, which could explain the observed performance gap between the two parallel configurations. To further prove this, efficiency is mostly conserved between the two configurations, with only a slight increase in the number of rollbacks, possibly due to the stricter causality requirement imposed by our event ordering.

The results from the more demanding version of E-TIES, shown in Figure 7.7 reinforce these findings, showing that ignoring ties results in even better performance. As more events are tied, extraction costs increase significantly, strengthening the

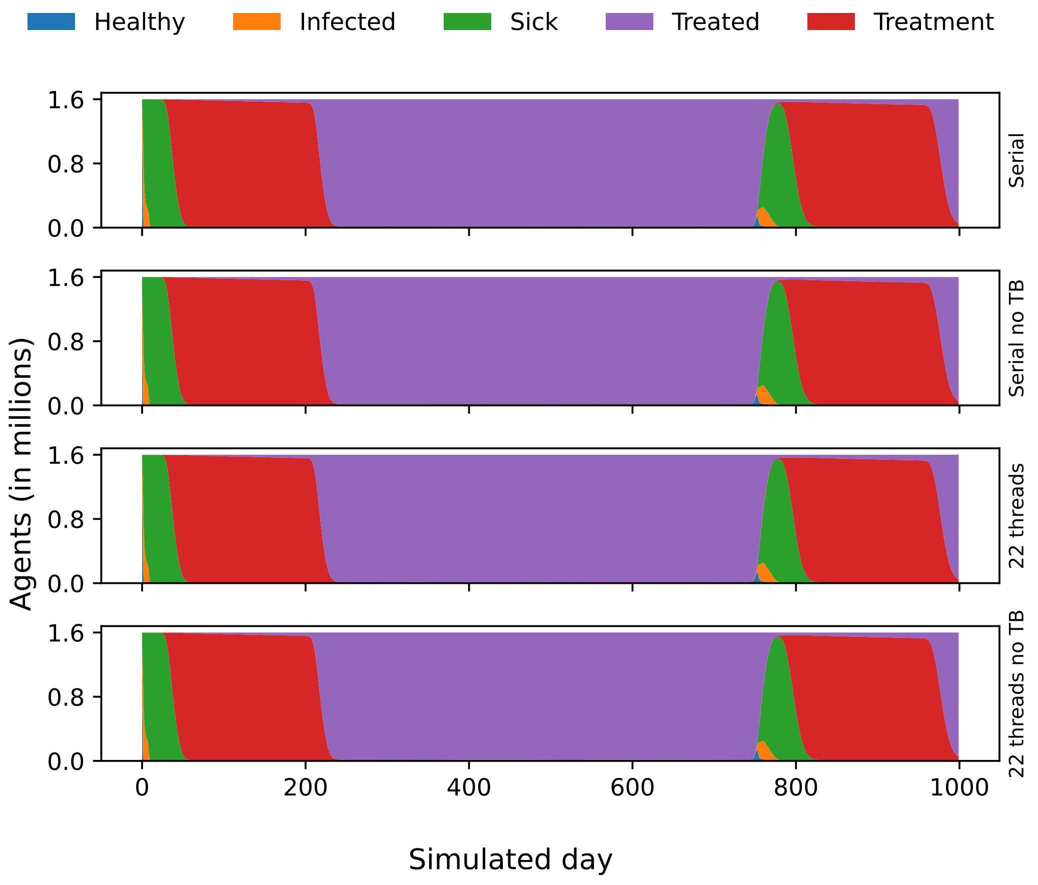


Figure 7.9. Evolution of the TBC model

argument that queue management is the main factor affecting performance. Nevertheless, the speedup shows that both configurations scale well with a larger model. At the same time, the efficiency confirms that the lower performance of tie-breaking runs is not due to an increase in rollbacks.

To conclude the first part of our evaluation, we observe that the experimental results from the TBC model, reported in Figure 7.8, suggest that tie-breaking may have a limited impact on performance for real-world models. While a noticeable difference can be observed at lower thread counts, the added cost of tie-breaking is effectively parallelised away with higher core count configurations.

The evaluation of PHold and TBC shows that our proposal has a negligible cost when there are few or no event ties but a high cost when ties are prevalent. We have shown that this is not due to parallelisation inefficiencies but rather because the FES of the simulation engine is effectively doing more work in the attempt to provide the correct event extraction ordering.

This increased cost is inevitable when using a priority queue that relies on element comparisons. This is because it requires at least one operation involving a logarithmic number of comparisons with respect to the events in the queue. Even if the model developer writes an efficient event comparison function, the cost of event inspection would still be much higher than timestamp comparison, which often requires only a few machine instructions. Furthermore, existing priority-queue data structures that do not involve direct timestamp comparisons are unsuitable for this purpose. For example, in a calendar queue, all tied events would end up in the same bucket, resulting in disastrous linear extraction times.

In other words, it seems that if the aim is running a simulation full of tied events, trying to obtain parallel deterministic executions may result in far more stress on queue management operations. As a solution, it may be possible to alleviate this burden by assigning the responsibility of reordering messages with the same timestamp to the straggler detection system. Although this approach is interesting, our initial investigations suggested that the associated cost of more frequent rollbacks trumps any other performance gain.

Moving onto the second part of our evaluation, we consider how the lack of tie-breaking affects a model's dynamics. For this analysis, we utilise the output produced by the TBC model. To extract the relevant data from the simulation, each LP in TBC schedules an event at regular intervals, which saves the count of the five classes of agents to a buffer that belongs to the LP's state. Upon simulation completion, the agent counts from each LP are merged by timestamp to produce a comprehensive evolution timeline of agent states.

The outcomes depicted in Figure 7.9 demonstrate no notable variation in the results among the different types of executions. This may lead one to believe that tie-breaking is insignificant in real-world scenarios. However, two factors should be considered. First, this model was chosen for its resistance to ties, enabling non-tie-broken executions to run without errors. Secondly, each model run produced slightly different trajectories, even with an identical pseudo-random generator seed.

In contrast, runs with tie-breaks are entirely deterministic for the same pseudo-random seed. As mentioned, deterministic simulation runs are valuable, particularly during model debugging.

As previously noted, models that do not heavily utilise tie-breaking logic do not experience a significant decrease in performance. Thus, it may be beneficial to maintain the enhanced tie-breaking feature enabled anyway.

7.4 Performance of Event Set Management Strategies

As we have mentioned earlier, a considerable amount of time is spent managing the event set when tie-breaking rules are enforced, and there is a non-negligible number of ties. Therefore, it is interesting to briefly study the effect of using different strategies for handling sets of events. As we will show, the effects of different strategies open up to a new dimension of risk management. Therefore, in this section, we also emphasise that a single component of a simulation runtime environment (the event set) can directly amalgamate multiple nuances of risk.

The concept of the event set is such a key concept for discrete-event simulation that it had appeared in the scientific literature since the early days of discrete-event simulation [114], when this technique was still the prerogative of operations research. Without an event set, it is impossible to realise any execution environment for discrete event simulation, regardless of the mechanisms used to support its execution.

The risk implications related to the event set arise from the strategy used to implement it, which can directly affect the LP clock skew, as it is essential to perform the operations to handle this data structure on the critical path of the simulation. This aspect of risk is additional to that already discussed in connection with the solution of ties. Furthermore, depending on the organisation of the execution environment and the paradigm supporting the simulation, the event set may become a shared structure, with the consequent need to guarantee correctness in accesses.

When using optimistic synchronisation, the ES has to maintain also already-processed events. Indeed, a rollback operation may restore a previous simulation state, thus requiring the reprocessing of past events. Therefore, when relying on speculative synchronisation, it is possible to divide future and past events into two (logical) different data structures, namely the FES and the PES, which we already introduced in Chapter 5.

Previous literature results have shown that if the execution grain of events is relatively fine, the ES can become the bottleneck of a PDES system [36]. To compensate for these issues (and also deal explicitly with concurrency), multiple data structures have been proposed in the literature—we have already surveyed them in Section 4.3. As it has been shown experimentally [27, 34, 128, 136], different data structures offer different performance guarantees depending on the model dynamics or the event distribution. While some data structures are typically better suited for larger models, there is no clear winner.

Nevertheless, here we advocate that there is an additional axis of intervention in managing the event set that has not been sufficiently explored in the literature. Indeed, typical PDES runtime environment implementations (see, e.g., [12, 81]) rely on a per-LP ES or FES/PES. The consensus on this organisation probably stems from a direct correspondence between the theoretical model of discrete-event simulation and the resulting implementation. However, we believe it is useful to shed light on the implications of risk, considering this time in the nuance of poor simulation performance due to inefficient CPU utilisation and excessive wait times for shared resources (see Section 2.3).

Moreover, since there is a clear division between the concept of FES and PES in Time Warp, it is also possible to assume the use of different data structures for these subsets of the ES. This section aims not to provide an exhaustive scrutiny of all possible combinations. Instead, we want to show how even simple choices that differ from those identified in the literature can significantly impact the simulation performance.

Given the high core count provided by modern multi/many-core systems, it could be beneficial to rethink the classical use of data structures used to implement ES, regardless of which data structure is employed. For example, a good (to the best of our knowledge, unexplored) strategy could be maintaining a single per-thread ES.

Here, we consider two main ES management strategies and data structures that combine the abovementioned ones. The first relies on two separate per-LP data structures that keep the FES and the PES. The FES is implemented as a min-heap, while the PES is a singly-linked list. Newly processed events are extracted from the FES data structure and pushed back into the PES. Upon a rollback operation, invalidated events from the PES are moved back to the FES.

To ensure a locally-causal execution, i.e. to implement the Lowest Timestamp First (LTF) scheduling strategy [35], a worker thread has to observe all the FES queues of the LPs it is currently managing. In this scenario, the expected scheduling time is $O(n \log(k))$, where n is the number of LPs managed by the worker thread and k is the number of elements of the targeted FES min-heap. Instead, the expected runtime cost for inserting an event is $O(\log(k))$. The same cost is incurred for invalidating an event from the PES of an LP.

The second approach maintains the per-LP PES, but each worker thread now keeps all the future events of its managed LPs in a single FES using a min-heap queue. Each worker thread can implement an LTF scheduling policy by simply observing its own FES. This operation costs $O(\log(nk))$ since only a single (although larger) queue has to be managed. This improvement comes at the expense of insertions and invalidations of events, which cost $O(\log(nk))$ as well.

The experiments we report here have been carried out on a `c5a.16xlarge` AWS instance, equipped with an unspecified AMD EPYC™ processor exposing 64 vCPUs. We have used the classic PHold model set up as a benchmark so that all configurations commit approximately 2^{30} events per run. We have used two configurations of the model. The first is a small configuration involving only 128 LPs. The second is a larger one, encompassing 16,384 LPs. The goal of this study is therefore to see how different strategies to implement the ES affect the simulation throughput, when the model's scale varies.

The results for the small configuration can be observed in Figure 7.10. When simulating 128 LPs, the per-LP solution can marginally outperform the per-thread solution. As illustrated in Figure 7.12, the benefit is minimal – a maximum of 4% – but consistently observable. It's reasonable to disregard the data with thread counts exceeding 32 since both configurations underperform after that threshold.

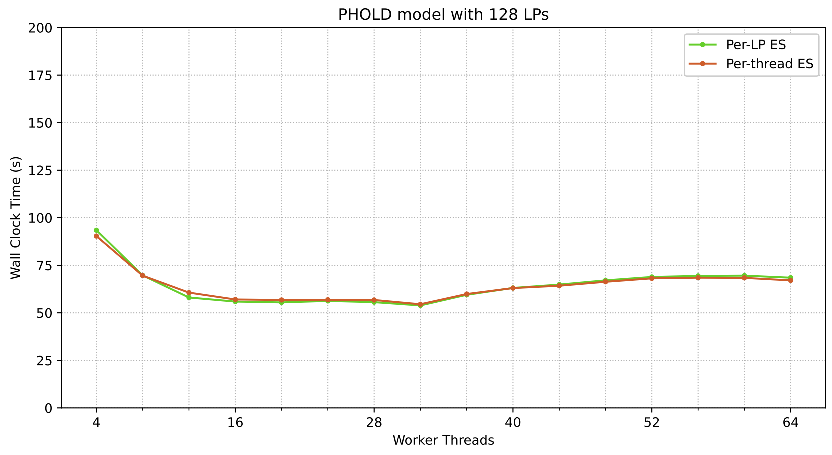


Figure 7.10. Results with PHold Model—128 LPs.

This outcome aligns with our expectations since the Amazon instance used in the experiment offers 64 vCPUs backed by 32 physical CPU cores. By binding worker threads to CPUs, configurations beyond the 32nd thread would execute on both hyperthreads of a few physical cores, leading to an uneven distribution of processing power among worker threads.

With this limited number of LPs, PHold follows a nearly serial simulation path: rollbacks frequently occur, favouring the first strategy's lower event invalidation cost. The advantage is minor since rollback costs are dominated by state restoration and anti-events operations. However, certain use cases may involve simulating a number of LPs similar to the number of threads; in such situations, having a single FES per LP might be worth considering.

Figure 7.11 presents the findings for the larger configuration. Both solutions scale with 16,384 LPs in this case, but the per-thread ES strategy outperforms the per-LP strategy. The first strategy exhibits super-linear scaling, mainly because the scheduling costs increase linearly with the number of assigned LPs per worker thread. For instance, with four threads, event scheduling operations must check 4,192 queues, while with 32 threads, that number becomes 512.

As indicated in Figure 7.12, the per-thread ES strategy achieves a speedup as high as 21 when dealing with a lower number of worker threads, and still an impressive 44% at the other end of the scale.

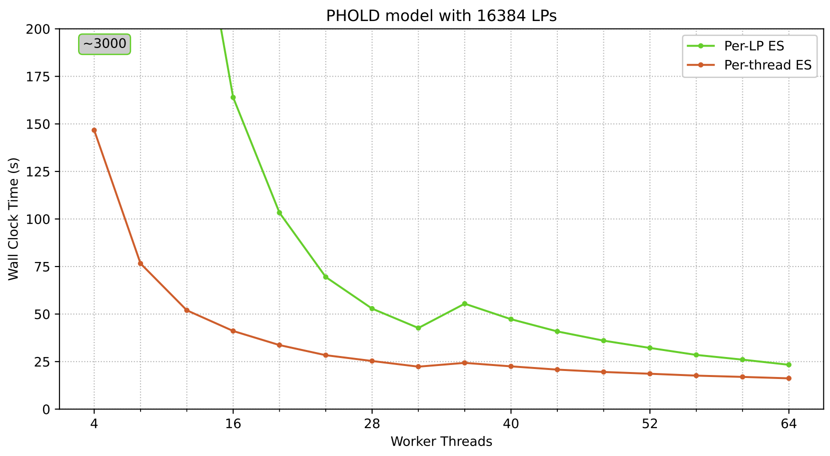


Figure 7.11. Results with PHold Model—16,384 LPs.

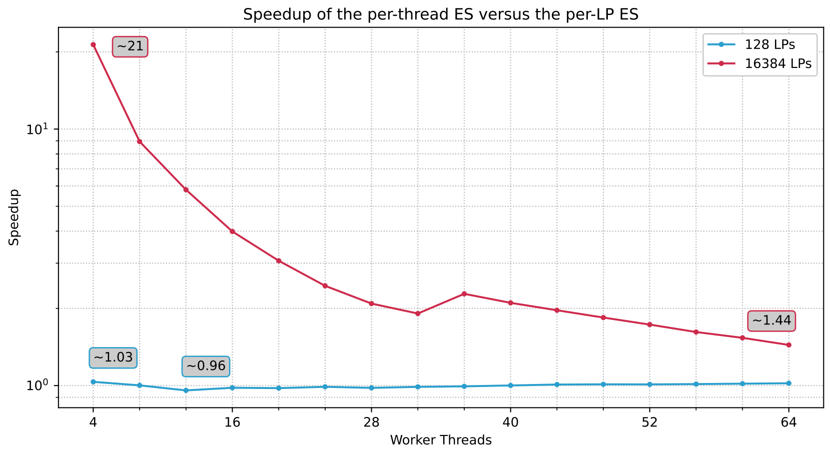


Figure 7.12. Speedup.

Once again, the effect of hyperthreading on the simulation performance is noticeable. It is more evident for the per-LP ES strategy, which is outperformed by the 28-thread configuration when run with 36 threads.

7.5 Discussion

The simple yet effective tie-breaking technique we have presented in this Chapter enables replicable and repeatable executions of concurrent simulations in a way similar to what would be observed in a sequential execution using the same technique. Our approach is easy to implement for both model developers and simulation engine designers.

Although our strategy may not be the optimal solution from a modelling point of view, it still serves as a fundamental tool to implement a suitable model-specific tie-breaking strategy. Our experimental results showed that the cost of using our technique is negligible for models with infrequent event ties. Most importantly, we demonstrated that, in any case, our technique does not hinder parallel scaling properties.

Additionally, we have shown that there may be cases where the entire tie-breaking logic is unnecessary, even for real-world models. However, we argue that for such models, the cost of tie-breaking is negligible, and therefore, we recommend enabling it.

We also provided insights on the performance implications for models with many ties. A relevant aspect is related to the implementation of the event set. In particular, the number of operations executed by the selected data structure can severely affect the performance. This result suggests that the event set can be related to the notion of risk. We have therefore investigated possible strategies to manage event sets.

The results of this investigation have highlighted that the body of literature on data structure for the event set possibly lacks a dimension for exploration. While much emphasis has been placed on the specific data structure used, less emphasis has been placed on how these structures can be used together to support a single simulation.

Our exploratory results confirm that a good ES data structure is essential for performance, as also indicated by the results related to the tie-breaking strategy, but they highlight that their usage pattern also plays a crucial role.

We believe that this is a research line that deserves much attention. Indeed, there could be other sources of risk to take into account that may have a direct impact on the overall simulation performance. As highlighted in this chapter, we envisage the possibility of relying on non-FIFO data structures to implement the ES. The implications on simulation dynamics may reveal interesting results.

As a side note, it is worth mentioning that even small details like thread pinning, in conjunction with a suboptimal utilisation of hyperthreading, can result in varying levels of risk exposure. In this case, configurations with 36 threads, for the most part, performed worse, possibly due to an imbalance in processing power available to the worker threads, leading to a slightly increased frequency of rollbacks.

Additionally, as highlighted in Chapter 6, employing autonomic self-optimisation is paramount to delivering competitive simulations in the face of highly-dynamic workloads. The same strategy could be applied to the management of the ES, mainly if the number of ties dynamically changes over different phases of the simulation. Indeed, we may design simulation environments that can switch the set's organisation and the actual data structure at runtime. Clearly, the involved costs might be significantly high: making the wrong decision could hamper the overall performance. According to our revisited notion of risk, this could be an additional source.

Chapter 8

Conclusions

In this thesis, we addressed the problem of risk management in PDES. In doing so, we have attempted to reinterpret various concepts from the classical literature within a single conceptual framework named, precisely, risk.

Clearly, we do not expect the work we have done over the past three years to be complete: to gather under one umbrella the vast amount of simulation techniques that have been proposed is a grand challenge. Nevertheless, we believe that we have contributed in our own small way to repositioning, consolidating and bridging in new ways several concepts now established by the scientific community.

First and foremost, however, we are designers, engineers and developers of high-performance execution environments whose goal is to provide modellers with the highest possible performance from a given hardware architecture. The conceptual framework we have provided is therefore steeped in technical aspects and strongly geared towards the modeller.

This aspect is close to our heart since, in our little experience, we have observed an internal dichotomy within the simulation community between those who deal directly with models and those who, in the end, are concerned with supporting their execution. For example, in Chapter 3, we observed how the most popular methodologies for the simulation of Spiking Neural Networks lead to potentially inaccurate results and possibly higher computational costs. However, at the same time, there is a proliferation of high quality models used in multiple real-world studies that have been adapted to simulation paradigms that maybe weren't the best fit for the application. We have observed the same phenomenon in other fields, such as demography or traffic simulation.

During this thesis work, we identified two critical aspects that make high-performance simulation still niche compared to real-world applications. The first is related to the lack of uniformity in approaches: the landscape is exceptionally vast, and often very specific problems are attempted to be solved with elegant and ingenious solutions. However, as we have shown in this thesis, an optimal solution for a particular problem can be highly suboptimal for other application contexts.

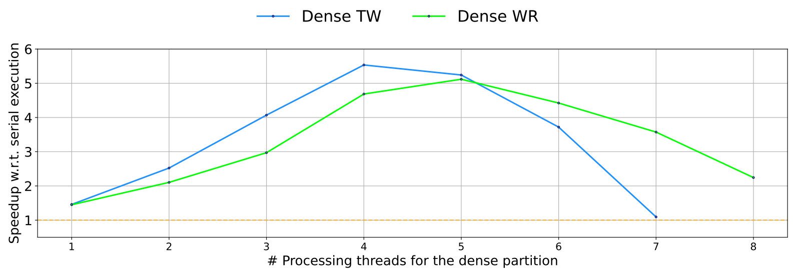
Therefore, we believe that a critical step in enabling solutions already in the literature to become appealing to model developers requires an effort to standardise approaches. This is the driving force behind the proposal in Chapter 5. We believe that, with a technological and methodological effort, it is possible to arrive in the years to come at a unified simulation methodology that allows, for any type of model, never to obtain speedups of less than 1.

The other obstacle to the widespread adoption of novel simulation techniques proposed by the research community is related to the nature of the experiments typically conducted. To better convey the idea, we report in Figure 8.1 some results, still awaiting publication, collected on a workstation-class machine. The setup is similar to that discussed in Chapter 5. With 10 total cores to play with, even in an unbalanced PHOLD configuration, a decent speedup of 5 is achieved by both our proposed hybrid algorithm and pure Time Warp implementation.

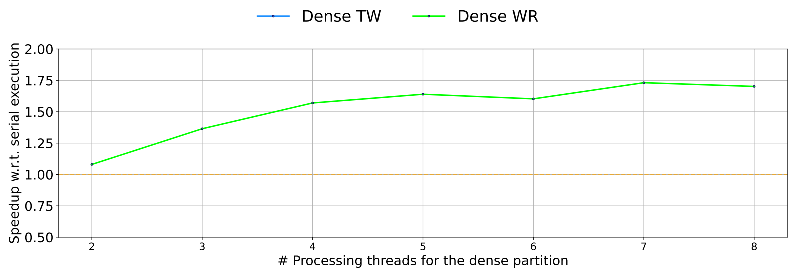
If we make the same comparison using the SIR model, we do not see any points plotted for the Time Warp curve: no configuration can achieve a speedup higher than one. The extreme adoption of synthetic models may lead to conclusions that are not entirely correct on the goodness of some proposals since the results may be strongly divergent when exercised with real models.

In general, we consider the scarcity of real-world benchmark model suites and the excessive adoption of synthetic models in configurations that are too simple to simulate to be two competing factors to the low adoption of more advanced simulation techniques, as these are not really exercised in disparate real-world scenarios.

In our own small way, we have always tried to compare our solutions with real



(a) PHOLD



(b) SIR

Figure 8.1. Synthetic benchmarks: running on a Intel® Core™ i9-10900X

models, however often related to the same domain. Regardless of the good results presented in the chapters of this thesis, it was the experimental comparison with real models that allowed us to achieve that level of optimisation, whether static or autonomic, that made all the difference.

Although our risk concept is probably not immediately actionable, the technical innovations we have presented certainly are. We have presented several methods for manipulating the level of risk in parallel simulations, which enable us to achieve very good tradeoffs between accuracy and performance by reducing or exploiting risk.

Our experiments, which encompassed both synthetic and real-world applications, have illustrated the significant impact of simulation engine design choices on the level of risk and demonstrated the efficacy of our technical contributions. We believe these contributions will be helpful in the simulation community and provide new avenues for future research.

Overall, the journey we took to get to the writing of this thesis was tiring but enjoyable. Some of the topics we have tried to address innovatively may suggest new lines of research that, we believe, will lead to good results if they are guided by extensive experimental comparison with real models. We hope that the reader of these contributions will share our vision and embark on new research. The risk is to discover hundreds of new facets of PDES.

Bibliography

[1] Sameera Abar, Georgios K Theodoropoulos, Pierre Lemarinier, and Gregory M P O'Hare. 2017. Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Computer Science Review* 24 (2017), 13–33. `https://doi.org/10.1016/j.cosrev.2017.03.001`

[2] Philipp Andelfinger, Andrea Piccione, Alessandro Pellegrini, and Adelinde Uhrmacher. 2022. Comparing Speculative Synchronization Algorithms for Continuous-Time Agent-Based Simulations. In *Proceedings of the 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT '22)*. IEEE, Piscataway, NJ, USA, 57–66. `https://doi.org/10.1109/DS-RT55542.2022.9932067`

[3] Philipp Andelfinger and Adelinde Uhrmacher. 2021. Optimistic Parallel Simulation of Tightly Coupled Agents in Continuous Time. In *Proceedings of the 2021 IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, Piscataway, NJ, USA, 1–9. `https://doi.org/10.1109/DS-RT52167.2021.9576156`

[4] Philipp Andelfinger and Adelinde M Uhrmacher. 2023. Synchronous Speculative Simulation of Tightly Coupled Agents in Continuous Time on CPUs and GPUs. *Simulation: Transactions of the Society for Modeling and Simulation International* (2023).

[5] Laurent R G Auriche, Francesco Quaglia, and Bruno Ciciani. 1998. Run-time selection of the checkpoint interval in time warp based simulations. *Simulation Practice and Theory* 6 (1998), 461–478.

[6] Duane Ball and Susan Hoyt. 1990. The Adaptive Time-Warp Concurrency Control Algorithm. In *Distributed Simulation (PADS '90)*, David Nicol (Ed.). Society for Computer Simulation, San Diego, CA, USA, 174–177.

[7] Blind Authors. 2023. Zero Lookahead? Zero Problem. The Window Racer Algorithm. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '23)*. ACM, New York, NY, USA. Under Review.

[8] Azzedine Boukerche and Sajal K Das. 1997. Dynamic load balancing strategies for conservative parallel simulations. In *Proceedings of the eleventh workshop on Parallel and distributed simulation* (Lockenhaus, Austria) *(PADS '97)*. IEEE Computer Society, Piscataway, NJ, USA, 20–28. `https://doi.org/10.1145/268826.268897`

[9] Romain Brette, Michelle Rudolph, Ted Carnevale, Michael Hines, David Beeman, James M Bower, Markus Diesmann, Abigail Morrison, Philip H Goodman, Frederick C Harris, Jr, Milind Zirpe, Thomas Natschläger, Dejan Pecevski, Bard Ermentrout, Mikael Djurfeldt, Anders Lansner, Olivier Rochel, Thierry Vieville, Eilif Muller, Andrew P Davison, Sami El Boustani, and Alain Destexhe. 2007. Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience* 23, 3 (Dec. 2007), 349–398. `https://doi.org/10.1007/s10827-007-0038-6`

[10] Randy Brown. 1988. Calendar Queues: a Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Commun. ACM* 31 (1988), 1220–1227.

[11] N Brunel. 2000. Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons. *Journal of computational neuroscience* 8, 3 (May 2000), 183–208. `https://doi.org/10.1023/a:1008925309027`

[12] Christopher D Carothers, David Bauer, and Shawn Pearce. 2002. ROSS: A high-performance, low-memory, modular Time Warp system. *Journal of parallel*

and distributed computing 62, 11 (Nov. 2002), 1648–1669. https://doi.org/10.1016/S0743-7315(02)00004-7

[13] Christopher D Carothers and Kalyan S Perumalla. 2010. On Deciding Between Conservative and Optimistic Approaches on Massively Parallel Platforms. In *Proceedings of the 2010 Winter Simulation Conference*, Björn Johansson, Sanjay Jain, and Jairo Montoya-Torres (Eds.). IEEE, Piscataway, NJ, USA, 678–687. https://doi.org/10.1109/WSC.2010.5679119

[14] Christopher D Carothers, Kalyan S Perumalla, and Richard M Fujimoto. 1999. The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture. In *Proceedings of the 1999 Winter Simulation Conference*, P A Farrington, H B Nembhard, D T Sturrock, and G W Evans (Eds.). WSC'99, Vol. 2. IEEE, Piscataway, NJ, USA, 1624–1633. https://doi.org/10.1109/WSC.1999.816902

[15] Christopher D Carothers, Kalyan S Perumalla, and Richard M Fujimoto. 1999. Efficient Optimistic Parallel Simulations Using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation* 9, 3 (July 1999), 224–253. https://doi.org/10.1145/347823.347828

[16] Kanianthra Mani Chandy and Jaydev Misra. 1979. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering* SE-5, 5 (Sept. 1979), 440–452. https://doi.org/10.1109/tse.1979.230182

[17] Chun-Hung Chen and Yu-Chi Ho. 1995. An approximation approach of the standard clock method for general discrete-event simulation. *IEEE Transactions on Control Systems Technology* 3, 3 (Sept. 1995), 309–317. https://doi.org/10.1109/87.406978

[18] Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2017. Transparently Mixing Undo Logs and Software Reversibility for State Recovery in Optimistic PDES. *ACM Transactions on Modeling and Computer Simulation* 27, 2 (May 2017), 1–26. https://doi.org/10.1145/3077583

[19] Gennaro Cordasco, Rosario De Chiara, Ada Mancuso, Dario Mazzeo, Vittorio Scarano, and Carmine Spagnuolo. 2013. Bringing together efficiency and effectiveness in distributed simulations: The experience with D-Mason. *Simulation* 89, 10 (Oct. 2013), 1236–1253. `https://doi.org/10.1177/0037549713489594`

[20] Vittorio Cortellessa and Francesco Quaglia. 2000. Aggressiveness/Risk Effects Based Scheduling in Time Warp. In *Proceedings of the 2000 Winter Simulation Conference*, Jeffrey A Joines, Russel R Barton, Keebom Kang, and Paul A Fishwick (Eds.). IEEE, Piascatawy, NJ, USA, 409–417. `https://doi.org/10.1109/WSC.2000.899746`

[21] Miles Dai. 2021. *Reverse Engineering the Intel Cascade Lake Mesh Interconnect*. Master's thesis. Massachusetts Institute of Technology.

[22] Faryad Darabi Sahneh, Caterina Scoglio, and Piet Van Mieghem. 2013. Generalized Epidemic Mean-Field Model for Spreading Processes Over Multilayer Complex Networks. *IEEE/ACM Transactions on Networking* 21, 5 (Oct. 2013), 1609–1620. `https://doi.org/10.1109/TNET.2013.2239658`

[23] Samir Das, Richard M Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. 1994. GTW: A Time Warp System for Shared Memory Multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, Jeffrey D Tew, Mani S Manivannan, Deborah A Sadowski, and Andrew F Seila (Eds.). Society for Computer Simulation International, San Diego, CA, USA, 1332–1339. `https://doi.org/10.1109/WSC.1994.717527`

[24] Samir R Das. 1996. Adaptive Protocols for Parallel Discrete Event Simulation. In *Proceedings of the 28th conference on Winter simulation* (Coronado, California, USA) *(WSC '96)*. IEEE Computer Society, USA, 186–193. `https://doi.org/10.1145/256562.256602`

[25] Lorenzo Dematté and Tommaso Mazza. 2008. On Parallel Stochastic Simulation of Diffusive Systems. In *Computational Methods in Systems Biology*, Monika Heiner and Adelinde M Uhrmacher (Eds.). LNCS, Vol. 5307.

Springer, Berlin, Heidelberg, Germany, 191–210. https://doi.org/10.1007/978-3-540-88562-7_16

[26] Phillip M Dickens and Paul F Reynolds, Jr. 1990. SRADS with Local Rollback. In *Distributed Simulation (PADS '90)*, David Nicol (Ed.). Society for Computer Simulation, San Diego, CA, USA, 161–164.

[27] Tom Dickman, Sounak Gupta, and Philip A Wilsey. 2013. Event pool structures for PDES on many-core Beowulf clusters. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Montr©al, Québec, Canada) *(SIGSIM PADS '13)*. Association for Computing Machinery, New York, NY, USA, 103–114. https://doi.org/10.1145/2486092.2486106

[28] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean Claude Andre, David Barkai, Jean Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney MacCabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S Mueller, Wolfgang E Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. 2011. The international exascale software project roadmap. *The international journal of high performance computing applications* 25, 1 (2011), 3–60. https://doi.org/10.1177/1094342010391989

[29] Jack Dongarra, Steven Gottlieb, and William T C Kramer. 2019. Race to Exascale. *Computing in science & engineering* 21, 1 (Jan. 2019), 4–5. https://doi.org/10.1109/MCSE.2018.2882574

[30] Klaus Eckhardt, Nicola Fohrer, and Hans-Georg Frede. 2005. Automatic model calibration. *Hydrological processes* 19, 3 (Feb. 2005), 651–658. `https://doi.org/10.1002/hyp.5613`

[31] Peter G Fennell, Sergey Melnik, and James P Gleeson. 2016. Limitations of discrete-time approaches to continuous-time contagion dynamics. *Physical Review E. Covering statistical, nonlinear, biological, and soft matter physics* 94, 5-1 (Nov. 2016), 052125. `https://doi.org/10.1103/PhysRevE.94.052125`

[32] Alois Ferscha and Satish K Tripathi. 1994. *Parallel and distributed simulation of discrete event systems.* Technical Report. University of Maryland at College Park. `https://doi.org/10.5555/193923`

[33] Josef Fleischmann and Philip A Wilsey. 1995. Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS '95).* IEEE Computer Society, Piscataway, NJ, USA, 50–58. `https://doi.org/10.1145/214282.214298`

[34] Romain Franceschini, Paul-Antoine Bisgambiglia, and Paul Bisgambiglia. 2015. A comparative study of pending event set implementations for PDEVS simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium* (Alexandria, Virginia) *(DEVS '15).* Society for Computer Simulation International, San Diego, CA, USA, 77–84. `https://doi.org/10.5555/2872965.2872976`

[35] Richard M Fujimoto. 1990. Parallel Discrete Event Simulation. *Commun. ACM* 33, 10 (Oct. 1990), 30–53. `https://doi.org/10.1145/84537.84545`

[36] Richard M Fujimoto. 1990. Performance of Time Warp Under Synthetic Workloads. In *Distributed Simulation (PADS '90)*, David Nicol (Ed.). Society for Computer Simulation International, San Diego, CA, USA, 23–28.

[37] Richard M Fujimoto. 1999. Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. In *Proceedings of the 13th Workshop on Parallel and*

Distributed Simulation (PADS '99). IEEE Computer Society, Washington, DC, USA, 46–53. `https://doi.org/10.1109/PADS.1999.766160`

[38] Richard M Fujimoto. 2000. *Parallel and Distributed Simulation Systems*. Wiley, Hoboken, NJ, USA.

[39] Richard M Fujimoto and Maria Hybinette. 1997. Computing Global Virtual Time in Shared-Memory Multiprocessors. *ACM Transactions on Modeling and Computer Simulation* 7 (1997), 425–446. `https://doi.org/10.1145/268403.268404`

[40] Richard M Fujimoto and Kiran S Panesar. 1995. Buffer management in shared-memory time warp systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation* (Lake Placid, NY, USA) *(PADS'95, Vol. 25)*. IEEE Comput. Soc. Press, Piscataway, NJ, USA, 149–156. `https://doi.org/10.1109/pads.1995.404306`

[41] Fabrizio Gagliardi, Miquel Moreto, Mauro Olivieri, and Mateo Valero. 2019. The international race towards Exascale in Europe. *CCF Transactions on High Performance Computing* 1, 1 (May 2019), 3–13. `https://doi.org/10.1007/s42514-019-00002-y`

[42] Marc-Oliver Gewaltig and Markus Diesmann. 2007. *NEST (NEural Simulation Tool)*. Vol. 2. Scholarpedia, Chapter 4. `https://doi.org/10.4249/scholarpedia.1430`

[43] Samanwoy Ghosh-Dastidar and Hojjat Adeli. 2009. Spiking neural networks. *International journal of neural systems* 19 (2009), 295–308. `https://doi.org/10.1142/S0129065709002002`

[44] Michael A Gibson and Jehoshua Bruck. 2000. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *The journal of physical chemistry. A* 104, 9 (March 2000), 1876–1889. `https://doi.org/10.1021/jp993732q`

[45] Daniel T Gillespie. 1976. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of computational physics* 22, 4 (Dec. 1976), 403–434. `https://doi.org/10.1016/0021-9991(76)90041-3`

[46] David W Glazer and Carl Tropper. 1993. On Process Migration and Load Balancing in Time Warp. *IEEE Transactions on Parallel and Distributed Systems* 4, 3 (March 1993), 318–327.

[47] David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* 23, 1 (March 1991), 5–48. `https://doi.org/10.1145/103162.103163`

[48] Samuel Greengard. 2020. Neuromorphic chips take shape. *Commun. ACM* 63, 8 (July 2020), 9–11. `https://doi.org/10.1145/3403960`

[49] Gerrit Großmann, Michael Backenköhler, and Verena Wolf. 2020. Importance of Interaction Structure and Stochasticity for Epidemic Spreading: A COVID-19 Case Study. In *Quantitative Evaluation of Systems*, Marco Gribaudo, David N Jansen, and Anne Remke (Eds.). Lecture Notes in Computer Science, Vol. 12289. Springer International Publishing, Cham, Switzerland, 211–229. `https://doi.org/10.1007/978-3-030-59854-9\_16`

[50] Sounak Gupta and Philip A Wilsey. 2014. Lock-free Pending Event Set Management in Time Warp. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Denver, Colorado, USA) *(SIGSIM PADS)*. ACM, New York, NY, USA, 15–26. `https://doi.org/10.1145/2601381.2601393`

[51] Alexander Hanuschkin, Susanne Kunkel, Moritz Helias, Abigail Morrison, and Markus Diesmann. 2010. A general and efficient method for incorporating precise spike times in globally time-driven simulations. *Frontiers in neuroinformatics* 4 (Oct. 2010), 1–19. `https://doi.org/10.3389/fninf.2010.00113`

[52] Joshua Hay and Philip A Wilsey. 2015. Experiments with Hardware-based Transactional Memory in Parallel Simulation. In *Proceedings of the 3rd ACM*

SIGSIM Conference on Principles of Advanced Discrete Simulation (London, United Kingdom) *(SIGSIM PADS '15)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/2769458.2769462

[53] Dirk Helbing and Péter Molnár. 1995. Social force model for pedestrian dynamics. *Physical Review E, covering statistical, nonlinear, biological, and soft matter physics* 51, 5 (May 1995), 4282–4286. https://doi.org/10.1103/physreve.51.4282

[54] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12 (1990), 463–492. https://doi.org/10.1145/78969.78972

[55] Xavier R Hoffmann and Marián Boguñá. 2019. Memory-induced complex contagion in epidemic spreading. *New journal of physics* 21, 3 (March 2019), 033034. https://doi.org/10.1088/1367-2630/ab0aa6

[56] Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2018. The Ultimate Share-Everything PDES System. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '18)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/3200921.3200931

[57] Alonso Inostrosa-Psijas, Veronica Gil-Costa, Mauricio Marin, and Gabriel Wainer. 2018. Semi-asynchronous approximate parallel DEVS simulation of web search engines. *Concurrency and computation: practice & experience* 30, 7 (April 2018), e4149. https://doi.org/10.1002/cpe.4149

[58] David Jefferson and Henry Sowizral. 1982. *Fast concurrent simulation using the Time Warp mechanism. Part I. local control.* Technical Report N-1906-AF. The Rand Corporation, Santa Monica, CA, USA.

[59] David R Jefferson. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 404–425. https://doi.org/10.1145/3916.3988

[60] David R Jefferson. 1990. Virtual time II: Storage Management in Conservative and Optimistic Systems. In *Proceedings of the 9th Symposium on Principles of Distributed Computing (PODC '90)*. ACM, New York, NY, USA, 75–89. https://doi.org/10.1145/93385.93403

[61] David R Jefferson and Peter D Barnes. 2022. Virtual Time III, Part 1: Unified Virtual Time Synchronization for Parallel Discrete Event Simulation. *ACM Transactions on Modeling and Computer Simulation* 32, 4 (Sept. 2022), 1–29. https://doi.org/10.1145/3505248

[62] David R Jefferson, B Beckman, F Wieland, L Blume, and Michele Diloreto. 1987. Time Warp Operating System. In *Proceedings of the eleventh ACM Symposium on Operating systems principles* (Austin, Texas, USA) *(SOSP '87)*. Association for Computing Machinery, New York, NY, USA, 77–93. https://doi.org/10.1145/41457.37508

[63] Matthias Jeschke, Alfred Park, Roland Ewald, Richard Fujimoto, and Adelinde M Uhrmacher. 2008. Parallel and Distributed Spatial Simulation of Chemical Reactions. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS '08)*. IEEE, Piscataway, NJ, USA, 51–59. https://doi.org/10.1109/PADS.2008.20

[64] Kevin Jones and Samir R Das. 1998. Combining optimism limiting schemes in time warp based parallel simulations. In *1998 Winter Simulation Conference. Proceedings*, Medeiros, D.J. and Watson, Edward F. and Carson, John S. and Manivannan, Mani S. (Ed.). WSC '98, Vol. 1. IEEE, Piscataway, NJ, USA, 499–505 vol.1. https://doi.org/10.1109/WSC.1998.745027

[65] J Kent Peacock, J W Wong, and Eric G Manning. 1979. Distributed simulation using a network of processors. *Computer Networks (1976)* 3, 1 (Feb. 1979), 44–56. https://doi.org/10.1016/0376-5075(79)90053-9

[66] Jeffrey O Kephart and David M Chess. 2003. The Vision of Autonomic Computing. *Computer* 36, 1 (Jan. 2003), 41–50. https://doi.org/10.1109/MC.2003.1160055

[67] William Ogilvy Kermak and Anderson Gray McKendrick. 1927. A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character* 115, 772 (1927), 700–721. `https://doi.org/10.1098/rspa.1927.0118`

[68] Arne Kesting, Martin Treiber, and Dirk Helbing. 2010. Enhanced intelligent driver model to access the impact of driving strategies on traffic capacity. *Philosophical Transactions of the Royal Society A. Mathematical, physical, and engineering sciences* 368, 1928 (Oct. 2010), 4585–4605. `https://doi.org/10.1098/rsta.2010.0084`

[69] Ki Hyung Kim, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park. 1997. Ordering of simultaneous events in distributed DEVS simulation. *Simulation practice and theory* 5, 3 (March 1997), 253–268. `https://doi.org/10.1016/s0928-4869(96)00009-2`

[70] Mariam Kiran, Paul Richmond, Mike Holcombe, Lee Shawn Chin, David Worth, and Chris Greenough. 2010. FLAME: simulating large populations of agents on parallel hardware architectures. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. IFAAMAS, Richland, SC, USA, 1633–1636. `https://doi.org/10.5555/1838206.1838517`

[71] Douglas Kothe, Stephen Lee, and Irene Qualters. 2019. Exascale Computing in the United States. *Computing in science & engineering* 21, 1 (Jan. 2019), 17–29. `https://doi.org/10.1109/MCSE.2018.2875366`

[72] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21 (1978), 558–565. `https://doi.org/10.1145/359545.359563`

[73] Yi-Bing Lin and Edward D Lazowska. 1991. Processor Scheduling for Time Warp Parallel Simulation. In *Advances in Parallel and Distributed Simulation (PADS '91)*, David Nicol, Richard M Fujimoto, and Vijay Madisetti (Eds.). Society for Computer Simulation, San Diego, CA, USA, 11–14.

[74] Yi-Bing Lin, Bruno R Preiss, Wayne M Loucks, and Edward D Lazowska. 1993. Selecting the checkpoint interval in time warp simulation. *ACM SIGSIM Simulation Digest* 23, 1 (July 1993), 3–10. `https://doi.org/10.1145/174134.158460`

[75] Jason Liu. 2009. *Parallel Discrete-Event Simulation.* Vol. 35. Wiley, Hoboken, NJ, USA, 12. `https://doi.org/10.1002/9780470400531.eorms0639`

[76] Boris D Lubachevsky. 1989. Efficient Distributed Event-Driven Simulations of Multiple-loop Networks. *Commun. ACM* 32, 1 (Jan. 1989), 111–123. `https://doi.org/10.1145/63238.63247`

[77] Charles M Macal. 2010. To Agent-based Simulation from System Dynamics. In *Proceedings of the 2010 Winter Simulation Conference*, Björn Johansson, Sanjay Jain, and Jairo Montoya-Torres (Eds.). IEEE, Piscataway, NJ, USA, 371–382. `https://doi.org/10.1109/WSC.2010.5679148`

[78] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2016. A Lock-Free O(1) Event Pool and Its Application to Share-Everything PDES Platforms. In *Proceedings of the 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT '16).* IEEE, Piscataway, NJ, USA, 53–60. `https://doi.org/10.1109/DS-RT.2016.33`

[79] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. 2016. A Non-Blocking Priority Queue for the Pending Event Set. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques* (Prague, Czech Republic) *(SIMUTOOLS).* Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST), Brussels, Belgium, 46–55.

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

Singapore) *(SIGSIM-PADS '17)*. Association for Computing Machinery, New York, NY, USA, 15–26. `https://doi.org/10.1145/3064911.3064926`

[81] D E Martin, P A Wilsey, R J Hoekstra, E R Keiter, S A Hutchinson, T V Russo, and L J Waters. 2003. Redesigning the WARPED simulation kernel for analysis and application development. In *Proceedings of the 36th Annual Simulation Symposium (SIMSYM '03)*. IEEE, Piscataway, NJ, USA, 216–223. `https://doi.org/10.1109/SIMSYM.2003.1192816`

[82] Yukinori Matsumoto and Kazuo Taki. 1992. *Adaptive Time-Ceiling for Efficient Parallel Discrete Event Simulation.* Technical Report TR-0798. Institute for New Generation Computer Technology.

[83] Satoshi Matsuoka. 2021. Fugaku and A64FX: the First Exascale Supercomputer and its Innovative Arm CPU. In *Proceedings of the 2021 Symposium on VLSI Circuits*. IEEE, Piscataway, NJ, USA, 1–3. `https://doi.org/10.23919/VLSICircuits52068.2021.9492415`

[84] Friedemann Mattern. 1993. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *J. Parallel and Distrib. Comput.* 18 (1993), 423–434. `https://doi.org/10.1006/jpdc.1993.1075`

[85] Neil McGlohon and Christopher D Carothers. 2021. Toward Unbiased Deterministic Total Orderings Of Parallel Simulations With Simultaneous Events. In *Proceedings of the 2021 Winter Simulation Conference*, Sojung Kim, Ben Feng, Katy Smith, Sara Masoud, Zeyu Zheng, Claudia Szabo, and Margaret Loper (Eds.). IEEE, Piscataway, NJ, USA, 1–15. `https://doi.org/10.1109/WSC52266.2021.9715459`

[86] Horst Mehl. 1991. Speed-up of Conservative Distributed Discrete Event Simulation Methods by Speculative Computing. In *Proceedings of the Multiconference on Advances in Paralleland Distributed Simulation (PADS '91)*, Vijay Krishna Madisetti, David Nicol, and Richard M Fujimoto (Eds.). Society for Computer Simulation, San Diego, CA, USA, 163–166.

[87] Cristina Montañola-Sales, Joan Francesc Gilabert-Navarro, Josep Casanovas-Garcia, Clara Prats, Daniel López, Joaquim Valls, Pere Joan Cardona, and Cristina Vilaplana. 2015. Modeling tuberculosis in Barcelona. A solution to speed-up agent-based simulations. In *Proceedings of the 2015 Winter Simulation Conference*, Levent Yilmaz, W K V Chan, I Moon, T M K Roeder, C Macal, and M D Rossetti (Eds.). IEEE, Piscataway, NJ, USA, 1295–1306. `https://doi.org/10.1109/WSC.2015.7408254`

[88] Cristina Montañola-Sales, Bhakti S S Onggo, Josep Casanovas-Garcia, Jose María Cela-Espín, and Adriana Kaplan-Marcusán. 2016. Approaching parallel computing to simulating population dynamics in demography. *Parallel computing* 59 (Nov. 2016), 151–170. `https://doi.org/10.1016/j.parco.2016.07.001`

[89] Federica Montesano, Romolo Marotta, and Francesco Quaglia. 2022. Spatial/Temporal Locality-based Load-sharing in Speculative Discrete Event Simulation on Multi-core Machines. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Atlanta, GA, USA) *(SIGSIM-PADS '22)*. Association for Computing Machinery, New York, NY, USA, 81–92. `https://doi.org/10.1145/3518997.3531026`

[90] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electronic notes in theoretical computer science* 89 (2003), 44–66. `https://doi.org/10.1016/S1571-0661(04)81042-9`

[91] David M Nicol. 1993. The Cost of Conservative Synchronization in Parallel Discrete Event Simulations. *J. ACM* 40, 2 (April 1993), 304–333. `https://doi.org/10.1145/151261.151266`

[92] David M Nicol. 1996. Principles of conservative parallel simulation. In *Proceedings of the 28th conference on Winter simulation*, John M Charnes, Douglas J Morrice, Daniel T Brunner, and James J Swain (Eds.). IEEE Computer Society, Piscataway, NJ, USA, 128–135. `https://doi.org/10.1145/256562.256591`

[93] David M Nicol and Xiaowen Liu. 1997. The Dark Side of Risk (What your

Mother Never Told you About Time Warp). In *Proceedings of the 11th Workshop on Parallel and distributed simulation (PADS '97)*. IEEE Computer Society, Washington, DC, USA, 188–195. https://doi.org/10.1145/268826.268920

[94] David M Nicol, Chris C Micheal, and Patrick Inouye. 1989. Efficient Aggregation Of Multiple LPs In Distributed Memory Parallel Simulations. In *Proceedings of the 1988 Winter Simulation Conference*, Edward A MacNair, Kenneth J Musselman, and Philip Heidelberg (Eds.). IEEE, Piscataway, NJ, USA. https://doi.org/10.1109/wsc.1989.718742

[95] Avinash C Palaniswamy and Philip A Wilsey. 1993. Adaptive Bounded Time Windows in an Optimistically Synchronized Simulator. In *Proceedings of the Third Great Lakes Symposium on VLSI (VLSI '93)*. IEEE Computer Society, Washington, DC, USA, 114–118. https://doi.org/10.1109/GLSV.1993.224467

[96] Avinash C Palaniswamy and Philip A Wilsey. 1993. An Analytical Comparison of Periodic Checkpointing and Incremental State Saving. In *Proceedings of the 7th workshop on Parallel and Distributed Simulation (PADS '93)*. ACM Press, New York, New York, USA, 127–134. https://doi.org/10.1145/158459.158475

[97] Avinash C Palaniswamy and Philip A Wilsey. 1994. Scheduling Time Warp Processes Using Adaptive Control Techniques. In *Proceedings of the 2004 Winter Simulation Conference*, Jeffrey D Tew, Mani S Manivannan, Deborah A Sadowski, and Andrew F Seila (Eds.). IEEE, Piscataway, NJ, USA, 731–738. https://doi.org/10.1109/WSC.1994.717422

[98] Alessandro Pellegrini, Sebastiano Peluso, Francesco Quaglia, and Roberto Vitali. 2016. Transparent Speculative Parallelization of Discrete Event Simulation Applications Using Global Variables. *International journal of parallel programming* 44, 6 (Dec. 2016), 1200–1247. https://doi.org/10.1007/s10766-016-0429-2

[99] A Pellegrini and F Quaglia. 2013. A study on the parallelization of terrain-

covering ant robots simulations. In *Euro-Par 2013: Parallel Processing Workshops*, Dieter Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L Scott, and Josef Weidendorfer (Eds.). LNCS, Vol. 8374. Springer, Heidelberg, Germany, 585–594. `https://doi.org/10.1007/978-3-642-54420-0\_57`

[100] Alessandro Pellegrini and Francesco Quaglia. 2014. Wait-free Global Virtual Time Computation in Shared Memory Time Warp Systems. In *Proceedings of the 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '14)*. IEEE, Piscataway, NJ, USA, 9–16. `https://doi.org/10.1109/SBAC-PAD.2014.38`

[101] Alessandro Pellegrini and Francesco Quaglia. 2015. NUMA Time Warp. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (London, United Kingdom) *(SIGSIM PADS' 15)*. ACM, New York, NY, USA, 59–70. `https://doi.org/10.1145/2769458.2769479`

[102] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2009. DiDyMeLoR: Logging only Dirty Chunks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects. In *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS '09)*. IEEE, Piscataway, NJ, USA, 45–53. `https://doi.org/10.1109/PADS.2009.24`

[103] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2012. The ROme OpTimistic Simulator: Core Internals and Programming Model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS)*. ICST, Brussels, Belgium, 96–98. `https://doi.org/10.4108/icst.simutools.2011.245551`

[104] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2015. Autonomic State Management for Optimistic Simulation Platforms. *IEEE Transactions on Parallel and Distributed Systems* 26 (2015), 1560–1569. `https://doi.org/10.1109/TPDS.2014.2323967`

[105] S Peluso, D Didona, and F Quaglia. 2012. Supports for transparent object-migration in PDES systems. *Journal of Simulation* 6, 4 (Nov. 2012), 279–293. `https://doi.org/10.1057/jos.2012.13`

[106] Giuseppe Perrone, Aurelio Zinno, and Noto La Diega. 2001. Fuzzy discrete event simulation: A new tool for rapid analysis of production systems under vague information. *Journal of intelligent manufacturing* 12, 3 (June 2001), 309–326. `https://doi.org/10.1023/A:1011213412547`

[107] Luiz Felipe Perrone. 2014. On the evolution toward computer-aided simulation. In *Modeling and Simulation-Based Systems Engineering Handbook*, Daniele Gianni, Andrea D'Ambrogio, and Andreas Tolk (Eds.). CRC Press, Boca Raton, FL, USA, 95–118. `https://doi.org/10.1201/b17902-6`

[108] P Peschlow and P Martini. 2006. Towards an efficient branching mechanism for simultaneous events in distributed simulation. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation* (Singapore) *(PADS'06)*. IEEE, Piscataway, NJ, USA, 133–133. `https://doi.org/10.1109/pads.2006.36`

[109] Patrick Peschlow and Peter Martini. 2007. A discrete-event simulation tool for the analysis of simultaneous events. In *Proceedings of the 2nd International ICST Conference on Performance Evaluation Methodologies and Tools* (Nantes, France). ICST, Bruxelles, Belgium, 1–10. `https://doi.org/10.4108/nstools.2007.2019`

[110] Patrick Peschlow and Peter Martini. 2007. Efficient analysis of simultaneous events in distributed simulation. In *Proceedings of the 11th International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'07)*. IEEE, Piscataway, NJ, USA, 244–251. `https://doi.org/10.1109/ds-rt.2007.21`

[111] Andrea Piccione. 2022. Comparing Different Event Set Management Strategies in Speculative PDES. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '22)*. ACM, New York, NY, USA, 55–56. `https://doi.org/10.1145/3518997.3534993`

[112] Andrea Piccione, Matteo Principe, Alessandro Pellegrini, and Francesco Quaglia. 2019. An Agent-Based Simulation API for Speculative PDES Runtime Environments. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Chicago, IL, USA) *(SIGSIM-PADS)*. ACM, New York, NY, USA, 83–94. `https://doi.org/10.1145/3316480.3322890`

[113] Matthew D Pickett, Gilberto Medeiros-Ribeiro, and R Stanley Williams. 2013. A scalable neuristor built with Mott memristors. *Nature materials* 12 (2013), 114–117. `https://doi.org/10.1038/nmat3510`

[114] M Pidd. 1984. Computer Simulation for Operational Research in 1984. In *Developments in Operational Research*, R W Eglese and G K Rand (Eds.). Pergamon Press, Oxford, UK, 19–30. `https://doi.org/10.1016/B978-0-08-031829-5.50007-5`

[115] Adriano Pimpini, Andrea Piccione, Bruno Ciciani, and Alessandro Pellegrini. 2022. Speculative distributed simulation of very large Spiking Neural Networks. In *Proceedings of the 2022 SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS)*. ACM, New York, NY, USA, 93–104. `https://doi.org/10.1145/3518997.3531027`

[116] Adriano Pimpini, Andrea Piccione, and Alessandro Pellegrini. 2022. On the Accuracy and Performance of Spiking Neural Network Simulations. In *2022 IEEE/ACM 26th International Symposium on Distributed Simulation and Real Time Applications (DS-RT) (DS-RT '22)*. IEEE, Piscataway, NJ, USA, 96–103. `https://doi.org/10.1109/DS-RT55542.2022.9932062`

[117] Mark Plagge, Christopher D Carothers, Elsa Gonsiorowski, and Neil Mcglohon. 2018. NeMo: A Massively Parallel Discrete-Event Simulation Model for Neuromorphic Architectures. *ACM Transactions on Modeling and Computer Simulation* 28 (2018), 1–25. `https://doi.org/10.1145/3186317`

[118] Chi-Sang Poon and Kuan Zhou. 2011. Neuromorphic Silicon Neurons and

Large-Scale Neural Networks: Challenges and Opportunities. *Frontiers in neuroscience* 5 (2011), 108. `https://doi.org/10.3389/fnins.2011.00108`

[119] Bruno R Preiss, Wayne M Loucks, and Ian D Macintyre. 1994. Effects of the checkpoint interval on time and space in time warp. *ACM Transactions on Modeling and Computer Simulation* 4, 3 (July 1994), 223–253. `https://doi.org/10.1145/189443.189444`

[120] Matteo Principe, Andrea Piccione, Alessandro Pellegrini, and Francesco Quaglia. 2020. Approximated Rollbacks. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Miami, FL, Spain) *(SIGSIM-PADS '20)*. ACM, New York, NY, USA, 23–33. `https://doi.org/10.1145/3384441.3395984`

[121] Francesco Quaglia. 1998. Event History Based Sparse State Saving in Time Warp. In *Proceedings of the twelfth workshop on Parallel and distributed simulation (PADS '98)*. IEEE Computer Society, Piscataway, NJ, USA, 72–79. `https://doi.org/10.1145/278008.278018`

[122] Francesco Quaglia. 2001. A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation. *IEEE Transactions on Parallel and Distributed Systems* 12 (2001), 346–362. `https://doi.org/10.1109/71.920586`

[123] Francesco Quaglia. 2015. A Low-overhead Constant-time Lowest-timestamp-first CPU Scheduler for High-performance Optimistic Simulation Platforms. *Simulation Modelling Practice and Theory* 53 (April 2015), 103–122. `https://doi.org/10.1016/J.SIMPAT.2015.01.009`

[124] Francesco Quaglia and Roberto Beraldi. 2004. Space uncertain simulation events: some concepts and an application to optimistic synchronization. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS)*. IEEE, Piscataway, NJ, USA, 181–188. `https://doi.org/10.1109/PADS.2004.1301299`

[125] F Quaglia and V Cortellessa. 2000. Grain sensitive event scheduling in time warp parallel discrete event simulation. In *Proceedings Fourteenth Workshop on*

Parallel and Distributed Simulation (PADS '00). IEEE, Piscataway, NJ, USA, 173–180. `https://doi.org/10.1109/PADS.2000.847163`

[126] Francesco Quaglia and Vittorio Cortellessa. 2002. On the processor scheduling problem in time warp synchronization. *ACM Transactions on Modeling and Computer Simulation* 12 (2002), 143–175. `https://doi.org/10.1145/643114.643115`

[127] Steven F Railsback and Volker Grimm. 2019. *Agent-Based and Individual-Based Modeling: A Practical Introduction, Second Edition.* Princeton University Press, Princeton, NJ, USA.

[128] Dhananjai M Rao and Julius D Higiro. 2019. Managing Pending Events in Sequential and Parallel Simulations Using Three-tier Heap and Two-tier Ladder Queue. *ACM Transactions on Modeling and Computer Simulation* 29, 2 (March 2019), 1–28. `https://doi.org/10.1145/3265750`

[129] Dhananjai M Rao, Narayanan V Thondugulam, Radharamanan Radhakrishnan, and Philip A Wilsey. 1998. Unsynchronized Parallel Discrete Event Simulation. In *Proceedings of the 1998 Winter Simulation Conference*, Deborah J Medeiros, Edward F Watson, John S Carson, and Mani S Manivannan (Eds.). WSC '98, Vol. 2. IEEE, Piscataway, NJ, USA, 1563–1570. `https://doi.org/10.1109/WSC.1998.746030`

[130] Mike Reape. 1989. A logical treatment of semi-free word order and bounded discontinuous constituency. In *Fourth Conference of the European Chapter of the Association for Computational Linguistics (EACL '89)*. Association for Computational Linguistics, Manchester, England, 103–110. `https://doi.org/10.3115/976815.976829`

[131] Peter L Reiher, Frederick Wieland, and David Jefferson. 1989. Limitation of Optimism in the Time Warp Operating System. In *Proceedings of the 21st Winter Simulation Conference*, Edward A MacNair, Kenneth J Musselman, and Philip Heidelberger (Eds.). ACM, New York, NY, USA, 765–770. `https://doi.org/10.1145/76738.76834`

[132] Craig W Reynolds. 1987. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics* 21 (1987), 25–34. `https://doi.org/10.1145/37402.37406`

[133] Paul F Reynolds. 1988. A Spectrum of Options for Parallel Simulation. In *Proceedings of the 20th Winter Simulation Conference*, Michael A Abrams, Peter L Haigh, and John C Comfort (Eds.). ACM, New York, NY, USA, 325–332. `https://doi.org/10.1109/WSC.1988.716181`

[134] Paul F Reynolds, Chrisopher F Weight, and J Robert Fidler, II. 1989. Comparative Analyses Of Parallel Simulation Protocols. In *1989 Winter Simulation Conference Proceedings*, Edward A MacNair, Kenneth J Musselman, and Philip Heidelberg (Eds.). IEEE, Piscataway, NJ, USA, 671–679. `https://doi.org/10.1109/WSC.1989.718741`

[135] Robert Rönngren and Rassul Ayani. 1994. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*. ACM, New York, NY, USA, 110–117. `https://doi.org/10.1145/182478.182577`

[136] Robert Rönngren and Rassul Ayani. 1997. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation* 7 (1997), 157–209.

[137] Rassul Rönngren and Michael Liljenstam. 2003. On event ordering in parallel discrete event simulation. In *Proceedings 13th Workshop on Parallel and Distributed Simulation (PADS '99)*. IEEE Comput. Soc, Piscataway, NJ, USA, 1–8. `https://doi.org/10.1109/pads.1999.766159`

[138] Andreas Ruscheinski and Adelinde Uhrmacher. 2017. Provenance in modeling and simulation studies — Bridging gaps. In *Proceedings of the 2017 Winter Simulation Conference*, Wai Kin (Victor), Andrea D'Ambrogio, Gregory Zacharewicz, Navonil Mustafee, Gabriel Wainer, and Ernest Page (Eds.). IEEE, Piscataway, NJ, USA, 872–883. `https://doi.org/10.1109/WSC.2017.8247839`

[139] Naimeh Sadeghi, A Robinson Fayek, and S P Mosayebi. 2013. Developing a fuzzy discrete event simulation framework within a traditional simulation engine. In *Proceedings of the 2013 Joint IFSA World Congress and NAFIPS Annual Meeting (NAFIPS '13)*. IEEE, Piscataway, NJ, USA, 1102–1106. `https://doi.org/10.1109/IFSA-NAFIPS.2013.6608554`

[140] Andrea A Saltelli, Marco Ratto, Terry Andres, Francesca Campolongo, Jessica Cariboni, Debora Gatelli, Michaela Saisana, and Stefano Tarantola. 2008. *Global sensitivity analysis: The primer*. Wiley-Blackwell, Hoboken, NJ. `https://doi.org/10.1002/9780470725184`

[141] David Schneider. 2022. The Exascale Era is Upon Us: The Frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. *IEEE Spectrum* 59, 1 (Jan. 2022), 34–35. `https://doi.org/10.1109/MSPEC.2022.9676353`

[142] Markus Schordan, Tomas Oppelstrup, Michael Kirkedal Thomsen, and Robert Glück. 2020. Reversible languages and incremental state saving in optimistic parallel discrete event simulation. In *Reversible Computation: Extending Horizons of Computing*. Springer International Publishing, Cham, 187–207. `https://doi.org/10.1007/978-3-030-47361-7\_9`

[143] Sven Sköld and Robert Rönngren. 1996. Event sensitive state saving in time warp parallel discrete event simulations. In *Proceedings of the 28th conference on Winter simulation* (Coronado, California, United States) *(WSC '96)*. ACM Press, New York, New York, USA, 653–660. `https://doi.org/10.1145/256562.256779`

[144] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-adjusting Binary Search Trees. *J. ACM* 32, 3 (July 1985), 652–686. `https://doi.org/10.1145/3828.3835`

[145] Tapas K Som and Robert G Sargent. 2000. Model Structure and Load Balancing in Optimistic Parallel Discrete Event Simulation. In *Proceedings of the*

14th Workshop on Parallel and Distributed Simulation (PADS '00). IEEE, Piscataway, NJ, USA, 147–154. `https://doi.org/10.1109/PADS.2000.847158`

[146] Sudhir Srinivasan and Paul F Reynolds. 1998. Elastic Time. *ACM Transactions on Modeling and Computer Simulation* 8, 2 (April 1998), 103–139. `https://doi.org/10.1145/280265.280267`

[147] Jeffrey S Steinman. 1991. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. In *Advances in Parallel and Distributed Simulation (PADS '91)*, Vijay K Madisetti, David Nicol, and Richard M Fujimoto (Eds.). Society for Computer Simulation, San Diego, CA, USA, 1111–1115.

[148] Jeffrey S Steinman. 1992. SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation. *International Journal in Computer Simulation* 2 (1992), 251-286.

[149] Jeffrey S Steinman. 1993. Breathing Time Warp. *Simuletter* 23, 1 (July 1993), 109–118. `https://doi.org/10.1145/174134.158473`

[150] Marcel Stimberg, Romain Brette, and Dan F M Goodman. 2019. Brian 2, an intuitive and efficient neural simulator. *eLife* 8, e47314 (Aug. 2019), e47314. `https://doi.org/10.7554/eLife.47314`

[151] Wen Jun Tan, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2021. Causality and Consistency of State Update Schemes in Synchronous Agent-based Simulations. In *Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Virtual Event, USA) *(SIGSIM-PADS '21)*. ACM, New York, NY, USA, 57–68. `https://doi.org/10.1145/3437959.3459262`

[152] Wai Teng Tang, Rick Siow Mong Goh, and Ian Li-Jin Thng. 2005. Ladder Queue: An O(1) Priority Queue Structure for Large-scale Discrete Event Simulation. *ACM Transactions on Modeling and Computer Simulation* 15, 3 (July 2005), 175–204. `https://doi.org/10.1145/1103323.1103324`

[153] Seng Chuan Tay, Gary S H Tan, and Karthik Shenoy. 2003. Algorithms and analyses: piggy-backed time-stepped simulation with 'super-stepping'. In *Proceedings of the 2003 Winter Simulation Conference*, Stephen E Chick, Paul J Sánchez, David Ferrin, and Douglas J Morrice (Eds.). Informs, Catonsville, MD, USA, 1077–1085. `https://doi.org/10.5555/1030818.1030961`

[154] Seng Chuan Tay, Yong Meng Teo, and Siew Theng Kong. 1997. Speculative parallel simulation with an adaptive throttle scheme. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation* (Lockenhaus, Austria) *(PADS '97)*. IEEE Computer Society, Piscataway, NJ, USA, 116–123. `https://doi.org/10.1145/268826.268909`

[155] Tommaso Tocci, Alessandro Pellegrini, Francesco Quaglia, Josep Casanovas-Garcia, and Toyotaro Suzumura. 2017. ORCHESTRA: An Asynchronous Wait-free Distributed GVT Algorithm. In *Proceedings of the 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT '17)*. IEEE, Piscataway, NJ, USA, 1–8. `https://doi.org/10.1109/DISTRA.2017.8167666`

[156] Liem Tran and Lucien Duckstein. 2002. Comparison of fuzzy numbers using a fuzzy distance measure. *Fuzzy Sets and Systems* 130, 3 (Sept. 2002), 331–341. `https://doi.org/10.1016/S0165-0114(01)00195-6`

[157] Martin Treiber and Venkatesan Kanagaraj. 2015. Comparing numerical integration schemes for time-continuous car-following models. *Physica A: Statistical Mechanics and its Applications* 419 (Feb. 2015), 183–195. `https://doi.org/10.1016/j.physa.2014.09.061`

[158] Stephen J Turner and Ming Qiang Xu. 1991. Performance Evaluation of the Bounded Time Warp Algorithm. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS '92)*, Marc A Abrams and Paul F Reynolds, Jr (Eds.). Society for Computer Simulation, San Diego, CA, USA, 117–126.

[159] Pirooz Vakili. 1991. Using a standard clock technique for efficient simulation.

Operations Research Letters 10, 8 (Nov. 1991), 445–452. `https://doi.org/10.1016/0167-6377(91)90021-G`

[160] Roberto Vitali, Alessandro Pellegrini, and Giornata Cerasuolo. 2012. Cache-Aware Memory Manager for Optimistic Simulations. In *Proceedings of the 5th International Conference on Simulation Tools and Techniques (SimuTools '12)*. ICST, Brussels, Belgium, 129–138. `https://doi.org/10.4108/icst.simutools.2012.247766`

[161] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2009. Benchmarking Memory Management Capabilities within ROOT-Sim. In *Proceedings of the 13th International Symposium on Distributed Simulation and Real Time Applications* (Singapore) *(DS-RT)*. IEEE, Piscataway, NJ, USA, 33–40. `https://doi.org/10.1109/DS-RT.2009.15`

[162] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2012. Load sharing for Optimistic Parallel Simulations on Multi Core Machines. *ACM SIGMETRICS Performance Evaluation Review* 40 (Dec. 2012), 2–11. `https://doi.org/10.1145/2425248.2425250`

[163] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2012. Towards Symmetric Multi-threaded Optimistic Simulation Kernels. In *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation (PADS '12)*. IEEE, Piscataway, NJ, USA, 211–220. `https://doi.org/10.1109/PADS.2012.46`

[164] Tim P Vogels and Larry F Abbott. 2005. Signal Propagation and Logic Gating in Networks of Integrate-and-Fire Neurons. *The Journal of neuroscience: the official journal of the Society for Neuroscience* 25, 46 (Nov. 2005), 10786–10795. `https://doi.org/10.1523/JNEUROSCI.3508-05.2005`

[165] Minh Vu, Lisong Xu, Sebastian Elbaum, Wei Sun, and Kevin Qiao. 2022. Efficient Protocol Testing Under Temporal Uncertain Event Using Discrete-event Network Simulations. *ACM Transactions on Modeling and Computer Simulation* 32, 2 (March 2022), 1–30. `https://doi.org/10.1145/3490028`

[166] Bing Wang, Bonan Hou, Fei Xing, and Yiping Yao. 2011. Abstract Next Subvolume Method: a logical process-based approach for spatial stochastic simulation of chemical reactions. *Computational biology and chemistry* 35, 3 (June 2011), 193–198. `https://doi.org/10.1016/j.compbiolchem.2011.05.001`

[167] Tom Warnke, Oliver Reinhardt, Anna Klabunde, Frans Willekens, and Adelinde M Uhrmacher. 2017. Modelling and simulating decision processes of linked lives: An approach based on concurrent processes and stochastic race. *Population studies* 71, sup1 (Oct. 2017), 69–83. `https://doi.org/10.1080/00324728.2017.1380960`

[168] Frederick Wieland. 1997. The threshold of event simultaneity. In *Proceedings of the 11th workshop on Parallel and Distributed Simulation* (Lockenhaus, Austria) *(PADS '97)*. IEEE Computer Society, USA, 56–59. `https://doi.org/10.1145/268826.268901`

[169] Frans J Willekens. 1999. The Life Course: Models and Analysis. In *Population Issues: An Interdisciplinary Focus*, Leo J G van Wissen and Pearl A Dykstra (Eds.). Springer, Dordrecht, The Netherlands, 23–51. `https://doi.org/10.1007/978-94-011-4389-9\_2`

[170] Pia Wilsdorf, Anja Wolpers, Jason Hilton, Fiete Haack, and Adelinde M Uhrmacher. 2022. Automatic Reuse, Adaption, and Execution of Simulation Experiments via Provenance Patterns. *ACM Transactions on Modeling and Computer Simulation* 33, 1-2 (Sept. 2022), 1–27. `https://doi.org/10.1145/3564928`

[171] Philip A Wilsey, Avinash C Palaniswamy, and Sandeep Aji. 1994. Rollback Relaxation: A Technique for Reducing Rollback Costs in Optimistically Synchronized Parallel Simulators. In *Proceesings of the 1994 International Conference on Simulation and Hardware Description Languages (ICSHDL '94)*, David Rhodes and Philip A Wilsey (Eds.). Society for Computer Simulation, San Diego, CA, USA, 143–148.

[172] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois

Knoll. 2018. Exploring Execution Schemes for Agent-Based Traffic Simulation on Heterogeneous Hardware. In *Proceedings of the 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT '18)*. IEEE Computer Society, Piscataway, NJ, USA, 1–10. `https://doi.org/10.1109/DISTRA.2018.8601016`

[173] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. 2019. A Survey on Agent-based Simulation Using Hardware Accelerators. *ACM Comput. Surv.* 51, 6 (Jan. 2019), 1–35. `https://doi.org/10.1145/3291048`

[174] Bernard P Zeigler, Tag Gon Kim, and Herbert Praehofer. 2000. *Theory of Modeling and Simulation.* Academic Press, London, UK.

[175] Hong Zhang, C M Tam, and Heng Li. 2005. Modeling uncertain activity duration by fuzzy number and discrete-event simulation. *European journal of operational research* 164, 3 (Aug. 2005), 715–729. `https://doi.org/10.1016/j.ejor.2004.01.035`