PhD in

Computer Science, Control and Geoinformation

PhD Cycle

XXXVII

Innovative Solutions for Speculative Parallel Discrete Event Simulation on Multi-core Shared-memory Machines

Federica Montesano

A.Y. 2023/2024

Tutor: Prof. Francesco Quaglia

Coordinator: Prof. Francesco Quaglia

"I don't know if we each have a destiny, or if we're all just floating around accidental-like on a breeze, but I, I think maybe it's both. Maybe both is happening at the same time."

— from the movie **Forrest Gump**

Ackowledgements

During these intense years, I have sometimes felt overwhelmed by self-doubt, but I mostly felt that I was constantly learning and growing, both as a student and as a person. This would not have been possible without the people around me. I want to thank my advisor, Professor Francesco Quaglia, that with his passionate guidance and positive attitude helped me grow as a student and as a researcher. The environment I found myself in was always free of judgement and focused on teaching me how to become a good researcher, I am deeply grateful for that and I hope I have done it justice. I want to thank Romolo, who has shown incredible patience since my master's thesis and has supported me throughout the PhD. His advice and encouragement helped me overcome many challenges and grow both personally and professionally. He certainly made my experience more enjoyable, and I hope that working with me didn't disrupt his sleeping schedule more than it already is. Thanks to Alessandro, whose witty and cheerful attitude made it so easy for me to gain trust of him and calm my nerves during stressful situations. He never failed to offer valuable advice to improve my work and provided immense support during the final phase of my PhD.

Outside of the academic environment, there is one person that has always been there when I needed him. Thanks to Luca, with whom I shared everything long before the PhD years and who has always encouraged me to do my best. Thanks for proving me wrong everytime I doubted myself, and thanks for patiently listening to my presentations about the same topics over and over again. I finally want to thank my family, that supported me throughout this experience without putting pressure on me.

Abstract

The continuous evolution of hardware has led to the breakdown of Moore's and Dennard's laws, and therefore to the establishment of multi-core shared-memory architectures. This evolution has made a paradigm shift in the design and optimization of High Performance Computing (HPC) applications necessary. In fact, the increasing number of cores, together with the increasing gap between CPU and memory performance, has highlighted significant bottlenecks of memory due to both data movement and memory bandwidth. These issues have been further exacerbated by Non Uniform Memory Access (NUMA) architectures. This thesis addresses challenges and opportunities posed by the evolving architectures, in the context of Parallel Discrete Event Simulation (PDES). The focus of this thesis is to provide innovative solutions for speculative PDES in order to fully exploit the parallelism of modern multi-core shared-memory machines. We focus on modern speculative PDES systems characterized by fine-grain sharing of simulation objects across threads, enabling dynamic workload distribution and load balancing improvement. We tackle the new challenges introduced by this paradigm related to memory management, spanning from memory hierarchy awareness and locality awareness, to operating systems services overhead, to consistency of global data structures accesses. We consider cache and NUMA locality awareness for event processing in order to reduce the costs of cache misses. We tackle

effective memory management techniques for the incremental checkpointing facility, exploiting traditional operating systems services and then developing a new operating system service to further improve the mechanism. Finally, we discuss an effective mechanism to access the committed global state of the simulation through state-swapping with close-to-zero delay to enable output collection. By integrating these optimizations into our target platform, we show significant improvements in the scalability and performance of PDES platforms. This thesis highlights the importance of considering the underlying hardware, emphasizing memory locality, the operating system's impact, and workload balancing. The outcomes of this thesis fill some gaps in the literature, presenting innovative solutions for speculative PDES exhibiting a fine-grain sharing of resources, and addressing the evolving demands of modern computational workloads.

Table of Contents

Ackowledgments 2
Abstract 3

List of figures 11
List of tables 12
List of Code 13
List of Algorithms 14

1 Introduction **15**
1.1 Context 15
1.2 Parallel Discrete Event Simulation 19
1.3 Challenges 23
1.4 Contributions 26
1.5 List of Published Papers 30

2 Preliminaries **33**
2.1 Memory Hierarchy 33
2.1.1 Cache memory 36
2.1.2 Main Memory 39
2.1.2.1 NUMA architecture 40
2.2 Shared-memory Architectures 41
2.2.1 Cache Coherency 42

2.3 Virtual Memory . . . 43
2.3.1 The Translation Lookaside Buffer . . . 45
2.4 Discrete Event Simulation Systems . . . 47
2.5 Parallel Discrete Event Simulation Systems . . . 49
2.5.1 Time Warp . . . 52
2.5.1.1 Local Virtual Time . . . 53
2.5.1.2 Global Virtual Time . . . 55
2.5.1.3 State Saving and Causality Violations Recovery . . . 58
2.5.1.4 Reverse computation . . . 60
2.6 Reshuffle of the PDES Architecture . . . 60
2.7 Simulation Models and Benchmarks . . . 65
2.7.1 PHOLD . . . 65
2.7.2 Personal Communication System . . . 66
2.7.3 Tuberculosis . . . 68

3 State-of-the-art 69
3.1 Memory Locality . . . 70
3.2 Checkpointing . . . 78
3.3 State Trajectory Inspection . . . 84

4 Spatial/Temporal Locality-based Load-sharing 90
4.1 Baseline Architectural Concepts . . . 91
4.1.1 Distance between Threads . . . 91
4.1.2 Simulation Object Memory Layout . . . 91
4.2 Locality Aware Scheme . . . 92
4.3 Workload Management Scheme . . . 98
4.4 Multi-view Event Pool Management . . . 103

4.5 Dynamic Window Management . . . 107
4.6 Dynamic NUMA Placement of Simulation Objects . . . 110
4.7 Experimental Evaluation . . . 113
4.7.1 Test-bed Environment . . . 113
4.8 Benchmark Applications . . . 114
4.8.1 Preliminary Experimental Evaluation for Parameters Setup . . . 116
4.9 Results . . . 119
4.9.1 Results with PHOLD . . . 119
4.9.2 Results with PCS . . . 120
4.9.3 Results with TBC . . . 128
4.10 Final Remarks . . . 128

5 Memory Aware and Lightweight Mechanisms for Incremental Checkpointing 131
5.1 Write-tracking Mechanism via `mprotect()` . . . 134
5.2 Decision Model for the Memory Aware Incremental Checkpointing . . . 137
5.2.1 Estimating Costs of the Buddy-based Approach . . . 142
5.3 Experimental Evaluation . . . 143
5.3.1 Test-bed Environment . . . 143
5.4 Benchmark Application . . . 145
5.5 Results . . . 146
5.5.1 Considerations . . . 149
5.6 Lightweight Operating System Service for Incremental Checkpointing . . . 151
5.6.1 Write-tracking Mechanism via LKM . . . 153

5.6.2 Dirty-page Address Logging Device 160
5.7 Experimental Evaluation 164
5.7.1 Test-bed Environment 164
5.8 Preliminary Experimental Evaluation 165
5.8.1 Results 166
5.9 Benchmark Application 168
5.9.1 Results 169
5.10 Final Remarks 172

6 Effective Access to the Committed Global State 175
6.1 System Architecture 180
6.2 Execution Contexts 183
6.3 Memory Safety of Simulation Object States 189
6.4 State-swapping Activities Distribution 193
6.5 Experimental Evaluation 197
6.5.1 Test-bed Environment 197
6.6 Benchmark Application 198
6.6.1 Compared Solutions and Metrics 199
6.7 Results 201
6.8 Final Remarks 205

7 Conclusions 207

List of figures

2.5.1 Example of a Causality Violation in speculative PDES 51
2.5.2 Example of a Rollback Procedure after a Causality Violation 52
2.5.3 Example for Explaining the GVT 56

4.2.1 Joint exploitation of temporal and spatial locality at cache level. 95
4.2.2 Joint exploitation of temporal and spatial locality at NUMA level. 96
4.4.1 Visual representation of the multi-view shared event pool. 105
4.5.1 Scheme of the state machine for managing W (throughput denotes the event rate). 111
4.8.1 Evaluation of different pipe-size values for the PCS model run with 40 threads. 117
4.8.2 Evaluation of different window-size values for the PCS model run with 40 threads. 118
4.9.1 Speedup with respect to PHOLD sequential execution with different event granularities. 120

4.9.2 Throughput of PHOLD with different event granularities and thread counts. Each label represents the speedup relative to the sequential execution for a given event granularity and thread count. 121
4.9.3 Execution speed with $\rho = 0.25$. 122
4.9.4 Execution speed with $\rho = 0.5$. 122
4.9.5 Execution speed with $\rho = 1$. 122
4.9.6 Results with PCS. 122
4.9.7 Execution speed 124
4.9.8 Event throughput for an individual simulation run . . 124
4.9.9 Results with PCS with 20% of hot-spot cells. 124
4.9.10 Execution speed 126
4.9.11 Event throughput for an individual simulation run . . 126
4.9.12 Workload skew between the two NUMA nodes 126
4.9.13 Average cumulative number of migrations. 126
4.9.14 Results with PCS with moving 20% of hot-spot cells. . 126
4.9.15 TBC execution speed 127

5.1.1 Illegal Write Access after Memory-protection via mprotect() 135
5.1.2 Write-tracking after Memory-protection via `mprotect()` 135
5.1.3 The buddy pages mechanism exploiting write-protection 136
5.5.1 Simulation execution speed 148
5.5.2 Checkpoint sizes 149
5.6.1 Write-tracking after Memory-protection via `track_memory()` 153
5.6.2 Page-fault Not Caused by Write-protection 155

5.6.3 Page-fault Caused by Write-protection 156
5.6.4 Logging Device Architecture 162
5.6.5 Logging Device Architecture 163
5.8.1 Latency of protection and page-fault handling with different memory sizes (log-scale on the y-axis) 166
5.8.2 Latency of memory-write protection (and unprotection) via `mprotect(...)` vs our custom LKM syscalls with different memory sizes 167
5.9.1 Throughput of Incremental State Saving in PCS via `mprotect(...)` vs LKM facilities on varying checkpoint periods and varying inter-arrival times 170

6.1.1 Target timeline of activities along wall-clock-time. . . 182
6.2.1 State diagram for the management of contexts 189
6.3.1 N-bit structure of the per-simulation object lock—N is set to 64 in our x86-64 oriented implementation. . . 191
6.7.1 PCS model - NC-D distribution 202
6.7.2 PCS model - CSA-D distribution 203
6.7.3 Efficiency w.r.t. original USE 204
6.7.4 Relative speedup w.r.t. original USE 205

List of tables

4.7.1 Hardware Platforms 114
4.9.1 Average rollback frequency of PCS 123
4.9.2 Average rollback length of PCS 123
4.9.3 Average rollback frequency of TBC 127
4.9.4 Average rollback length of TBC 127

List of Code Examples

6.1 Register setup for the CSR context. 185

List of Algorithms

4.3.1 Simulation Main Loop 99
4.3.2 Locality-based Load-sharing Management Functions . . 100
5.2.1 Memory Partitioning Algorithm 141
6.3.1 Simulation Object Locking Algorithm 191
6.3.2 Usage of the `potential_locked_object` Variable Kept in Thread-local-storage. 192
6.4.1 Algorithm executed in the CSR context—N is the number of simulation objects, with identifiers in $[0, N-1]$. . 195

Chapter 1

Introduction

1.1 Context

In the post-Moore era [64, 87, 88, 121], it has become necessary not only to run programs faster, but also to tailor them to the hardware on which they run, in order to fully exploit modern machines' capabilities. Furthermore, over the years, the number of processing units, namely cores, has increased, and consequently multi-core machines have become the reference for developing and executing applications. These kinds of machines have become a staple in both academic research and industry to reach higher performance along the path of exascale computing.

However, the growing amount of parallelism has not been counterbalanced by the increase of memory speed, leading to the well-known memory-wall problem [76, 135]. In fact, the performance gap between the processing unit, namely CPU, and main memory, namely RAM,

has been continuously growing, with memory latency remaining a significant bottleneck for application performance [9].
Furthermore, since 2005 Dennard's scaling has broken down [24, 26, 138], and this pushed the manufacturers to focus on multi-core processors, in order to improve the performance of parallel applications, also exploiting hyperthreading, i.e. the possibility of running two or more threads in parallel inside a single CPU-core, in particular on Intel processors [135]. Therefore, the evolution of multi-core machines has led to the rise of massively parallel applications, as well as new programming models focused on parallelism, like Compute Unified Device Architecture (CUDA) in the context of accelerated general-purpose processing, or Cilk and OpenMP, that also support shared-memory computing. The shared-memory computing paradigm, along with multi-processor/multi-core machines, has received much attention among researchers since the 1990s, due to the widespread adoption of programming models like Posix Threads and OpenMP in modern applications. The main characteristic of shared-memory systems is that the entire memory available is accessible in parallel by all the processing units. Consequently, in a shared-memory architecture, caches play a crucial role due to the impact that cache coherency protocols can have on performance, which is one reason why it is hard to develop scalable applications on these architectures [37, 125].
Memory coherency and the other above-mentioned problems related to the performance gap between CPUs and memory are exacerbated by Non-Uniform Memory Access (NUMA) architectures, due to their natural asymmetry and the distance between NUMA nodes. This is because data might be distributed unevenly across the memory banks

placed on distant NUMA nodes, so at each memory access the latency increases, not to mention the pressure on the interconnection between the NUMA nodes. In order to hide this performance gap, industries have designed and produced faster and smaller on-chip caches, to reduce memory latency, assuming that parallel applications use memory efficiently in terms of spatial and temporal locality, which is a risky assumption. In fact, many modern scientific computing applications such as graph processing and sparse linear algebra, require a huge amount of memory, but lack of memory awareness and/or locality, leading to performance degradation due to inefficient data movement [61, 77, 145]. Similarly, Discrete Event Simulation (DES) applications, which are widely used in fields such as network modelling, as well as manufacturing systems and biological systems modelling, healthcare systems management, also face substantial memory access challenges due to their irregular and dynamic memory access patterns. Allowing memory reuse through the consideration of the memory hierarchy and the NUMA architecture can increase the overall efficiency of an application, and this is particularly relevant in the context of High Performance Computing (HPC) applications running on top of multi-core shared-memory machines.

As a matter of fact, these machines turn out to be effective for running simulation applications, in particular Discrete Event Simulation (DES) and Parallel Discrete Event Simulation (PDES), which is a method of running DES models on top of multi-processor/multi-core machines [39]. More details about this kind of applications will be discussed in the next section, to allow better comprehension of the target platforms.

Along with the hardware evolution of multi-core shared-memory machines to allow HPC applications to run more efficiently, an architectural shift of PDES systems has also become necessary [35, 41]. In fact, in earlier times, most of the high performance computing applications, including DES and PDES systems, consisted of cluster-based applications, in which nodes communicated via message passing facilities [16, 80, 112, 123]. Consequently, the focus of many optimizations has been towards the distribution of the workload among the clustering nodes to reduce the communication overhead and delay, but the advent of multi-core machines imposed a reshuffle of PDES systems running on top of them, to fully exploit the inherent parallelism. We point out that optimizing PDES applications on multi-core machines is orthogonal to the optimization of cluster-based PDES, since we can consider each clustering node as a multi-core machine, so it is anyhow a relevant aspect to take into account when optimizing cluster-based or distributed applications. Other than that, further optimizations of PDES running on top of multi-core shared-memory machines have been necessary, taking into account the actual memory layout of these kinds of machines [54, 118]. All of this required a shift on how a PDES system should be devised, since in earlier times it was not common to share data between the processing units, but instead PDES applications typically relied on data partitioning and message passing communication. This thesis will present innovative solutions for speculative PDES on multi-core shared-memory machines. We will discuss the PDES paradigm and how its evolution has been fundamental in the field of both simulation and HPC in the next section.

1.2 Parallel Discrete Event Simulation

Over the years, many solutions and techniques have been integrated in PDES platforms with the aim of improving its capabilities to exploit parallelism of multi-core machines and improving overall performance and scalability. In order to understand the motivations towards these improvements and the goals of this thesis, we are going to provide some basic concepts about PDES.

One basic feature of a PDES system is the concept of Simulation Object (SO), also called Logical Process (LP), which allows representing a part of the simulation state. In this thesis the terms simulation object and LP will be used interchangeably.

Clearly, in a PDES platform there are several simulation objects, and each one of them is an autonomous entity representing a portion of the simulated system as a set of variables completely disjoint among each other. In this paradigm, events destined to different simulation objects can be executed in parallel by different worker threads. The execution of events is managed through event handlers, which are software components responsible for the processing logic (therefore, the update of the state) associated with each event. Each event carries a timestamp, and the event handler ensures that operations are applied consistently, maintaining causality within the simulation. This paradigm is suitable across a wide range of application domains, such as agent-based simulation systems [1, 124, 134], demographic [10, 68, 81, 82, 153] and urban traffic systems [53, 126], and networking applications to model

large-scale systems [12, 22, 54, 70, 94, 113, 118]. The literature highlights the broad interest in PDES systems to address complex systems. We particularly focus on PDES systems that enable speculative execution of events across multiple simulation objects. Speculative execution allows events to be processed out of their strict chronological order across different simulation objects, provided no causality violations occur (e.g., no event with lower timestamp is received after the execution of an event with larger timestamp). Speculation is a well-known approach used, for example, in the context of computer architectures and microprocessors [130], and of database systems [62, 114].

In PDES, speculative processing is also referred to as optimistic synchronization (originally coordinated via the Time Warp protocol) [38, 57], since it assumes that causality violations will not occur, in contrast to conservative synchronization, in which causality violations are completely avoided. In optimistic PDES, some recovery mechanism from causality violations must be considered. We will thoroughly discuss this issue in subsequent chapters, but some control mechanisms for identifying such violations and recover from them are the well-known techniques of rollback, either through checkpointing or through reverse computation. On the one hand, reverse computation techniques provide an anti-handler for each event handler, which reverses the computation of each operation and neutralizes its effects, ensuring that the state of the application is the same as it was prior to the computation. On the other hand, checkpointing consists of periodically saving the state of the simulation in order to support a possible state recovery, by restoring the previously saved state during the rollback procedure. This can either be done by saving the entire state of the

simulation at a certain time instant, namely full state saving, or by saving only the actually modified portions of the state since the last checkpoint, namely incremental state saving. When using checkpointing techniques, it is crucial to ensure that memory is not wasted due to periodically taken checkpoints. This is achieved by implementing a fossil-collection mechanism, which consists of identifying and reclaiming the so-called committed portions of the simulation state (referred to as fossils). These are the states that can no longer be affected by causality violations, allowing an efficient memory usage [151]. This is why the notion of Global Virtual Time (GVT) is important – which will be formally defined in Chapter 2 – as it represents a time boundary to allow separating the committed portion of the simulation state, where causality violations cannot occur, to a not yet committed one. Finally, the GVT value is also referred to as the commit-horizon of the simulation, and we will see how we can exploit it to optimize critical-path operations.

We have introduced some basic concepts of speculative PDES to give a general view of the applications we are targeting, we can now discuss how the paradigm has evolved over the years, to understand the context of the optimizations addressed in this thesis and why they are necessary. In fact, over the years, we have witnessed an evolution of speculative PDES architecture in terms of the workload distribution of LPs across worker threads, motivated by the above discussed evolution of multi-core shared-memory machines [12, 54, 94, 118].

In earlier times, traditional PDES systems presented a workload distribution scheme based on long-term binding: in this scheme, a simulation object was bound to a specific worker thread for a medium/long-term

period of time, that is the non-minimal wall-clock-time period between two subsequent re-assignments of the object to another thread. So, basically in this kind of architecture, worker threads process events from a subset of simulation objects for a medium/long amount of time, until a workload rebalancing occurs. This architecture failed to fully exploit the inherent parallelism of multi-core shared-memory machines, due to the partitioning of simulation objects across worker threads and the periodic rebalancing needed. This is why a new fine-grain sharing of simulation resources was developed, allowing processing events destined to any simulation object by any worker thread. This created a workload scheme based on a short-term binding, lasting no more than a single event execution. In speculative PDES, this technique has the advantage of bringing the computing power of the multi-core machine close to the commit-horizon of the simulation. In fact, in this scheme each worker thread picks at any time the unprocessed event with minimal timestamp destined to any simulation object, regardless of any previous partitioning.

However, while workload-balancing through data partitioning with long-term binding schemes can be accordingly scaled with the increasing number of cores, the short-term based schemes with fine-grain sharing of resources need proper synchronization and coordination across the threads in order to maintain consistency throughout the entire system, and they also should guarantee that no threads block each other while accessing shared data structures. It should also be guaranteed that the threads do not wait for each other due to potential causality constraints between events. These challenges have been effectively addressed in prior research [56, 72, 73], culminating into the development

of a new speculative PDES platform, called USE [54], that leverages the capabilities of multi-core shared-memory machines, becoming a reference platform for speculative PDES running on top of these kind of machines. In the following sections, we will delve into the challenges relative to this PDES paradigm and outline our approach to addressing these issues.

1.3 Challenges

The importance of shared-memory architectures in the context of HPC applications should be clearer, as well as the challenges involved in developing systems that scale effectively on these architecture [37]. Along with this, we described how the evolution of PDES systems has been a necessary step towards the better exploitation of multi-core shared-memory machines [54].
However, some of the limitations of this kind of PDES platform are:

1. Frequent accesses to the global event pool by all the worker threads in the system;
2. Lack of memory hierarchy awareness and locality awareness of the overall load-sharing scheme;
3. Non-negligible overhead of operating systems' services due to memory updates coordination across cores;
4. Possible interference when accessing shared data structures in order to identify a committed portion of the state.

Frequently accessing the shared event pool imposes synchronization costs and also pollutes the caches, failing to exploit locality. Other than that, the event pool is typically a linked-list data structure, whose traversing can become expensive (point 1). Moreover, the general lack of memory hierarchy awareness (e.g. for what concerns event processing) inhibits the exploitation of the caches and the overall memory hierarchy (point 2). In fact, the fine-grain sharing of resources allows the worker threads to constantly switch from one simulation object to another, potentially leading to relevant problems when cache-misses occur. In particular, NUMA-unawareness incurs in more costs when cache misses occur, due to the interconnection network and the latency of remote nodes.

Furthermore, PDES systems exhibiting fine-grained sharing of resources, can suffer from a kind of "side effect" overhead, e.g. when exploiting operating systems services to support classical speculative PDES operations, such as the checkpointing for state recovery from causality violations (point 3). In fact, as mentioned, state saving (namely checkpointing) techniques are necessary, but it is also relevant to reduce the amount of memory saved, that is why incremental checkpointing is used. The crucial problem is the identification of the modified portions of the state, and this is why memory protection services of common operating systems can be exploited. The challenging aspect regards the nature of memory protection services, and their kernel-side activities, i.e. Memory Management Unit (MMU) coordination, coupled with the fully-shared workload of PDES systems, since, due to the data sharing, memory protection mechanisms can cause large overhead, making a more accurate simulation's state management necessary.

In addition to efficient management of simulation object states, critical-path operations, such as the identification of a committed global state, are non-trivial in these types of systems (point 4). This problem has been previously tackled in [21] using a state-swapping approach, that is temporarily swapping the current state of a simulation object with a past committed one, but it targets PDES platforms exhibiting a partitioned workload among threads (namely, not fully-shared). In order to support the operation on fully-shared PDES platforms, we have to devise accurate mechanisms to avoid any possible interference caused by different worker threads running regular simulation operations on the same simulation object targeted by the state-swapping operation. To summarize, the challenges that will be the main objectives of this thesis are as follows:

- Cache locality awareness and NUMA awareness,
- The intrusiveness of operating systems services in terms of memory coordination activities,
- The possible interference and the coordination of the state-swapping for critical-path operations.

In the next section, we will present the contributions of this thesis, which aim to address the described challenges.

1.4 Contributions

The limitations described in the previous section play a central role in the limited scalability of speculative PDES on multi-core shared-memory machines. In fact, memory locality, and more in general memory awareness, is known to be a core aspect enabling scaled up performance.

Optimizations of parallel applications, targeting several memory-related aspects, have been widely investigated through the years, spanning from algorithms to reduce contention in multi-processor shared-memory systems [79], to collision avoidance between transactions [31] and the exploitation of Linux kernel's services as a support mechanism for scheduling transactions [69] in the context of transactional memory, to the enhancement of spatial locality of tasks in parallel applications [50]. Focusing on PDES, some optimizations targeted different aspects of PDES systems: to cite some of them, we can find buffers' management strategies [43], synchronization mechanisms through transactional memory [49], and data structures design for the event pool [30].

We aim to propose some innovative solutions for speculative PDES on multi-core shared-memory machines, tackling several memory-related aspects, spanning from memory locality-awareness for event processing, memory awareness in terms of the reduction of intrusiveness and overhead of operating systems' services associated to speculative PDES operations, i.e. the checkpointing operation, to the reduction of the intrusiveness and the delay of critical-path operation, i.e. identification of the committed global state to perform online output collection.

Our approach tackles these aspects from different perspectives:

1. Memory hierarchy awareness in terms of cache and NUMA locality, in order to improve simulation performance in terms of throughput and execution time [86];
2. Write-access awareness and coalescing of written memory pages, favouring spatial locality of memory pages when using operating systems' memory protection services to support checkpointing, in order to reduce the costs associated to these services [74];
3. Lightweight management of the simulation objects' states to reduce the overhead and intrusiveness of the above-mentioned operating systems' memory protection services, related to cross-CPUs interactions [85];
4. Reduction of the intrusiveness and delay of the state-swapping operation used to identify a committed global state of the simulation to do output collection [75].

To briefly illustrate the listed points, the point 1 in the list is tackled through a multi-level NUMA-aware event processing scheme, in order to favour spatial and temporal locality when processing events in a PDES platform running on top of a multi-core shared-memory machine that presents a fully shared workload scheme. We enabled this locality-aware load-sharing scheme exploiting a (virtual time) window-based approach to control the amount of speculative events processed, as a trade-off factor between the locality-aware processing of events and the risk of causality violations (and rollbacks). We have also provided for

a NUMA-aware support, allowing the migration of simulation objects among nodes for workload balancing purposes. The results obtained from this scheme showed higher performance in terms of throughput of the simulation execution, compared to the simulator described in [54], which does not implement any locality-aware event processing scheme. Points 2 and 3 target the exploitation of operating systems' memory protection services to support a classical speculative PDES operation, that is incremental checkpointing, first focusing on a novel incremental checkpointing support compared with the state-of-the-art (point 2), consisting on the exploitation of memory protection services of common operating systems, and then focusing on performance improvements of the memory protection service itself (point 3). In particular, we first considered a write-operations correlation model to improve spatial locality through the coalescing of memory pages when accessing the simulation objects' states to take the checkpoint (point 2) exploiting memory protection services. From this mechanism we achieved higher throughput with respect to the state-of-the-art approaches, and also compared to single-page approaches (namely, without coalescing pages).

We further improved the incremental checkpointing operation leveraging Linux Kernel Modules techniques, and managing memory pages in a more lightweight manner to tackle the above-mentioned overhead (see point 3 in the list) of the operating systems' memory protection services related to cross-CPU coordination, that make the costs non-negligible in PDES platforms exhibiting a fully-shared workload and presenting large state of the simulated model. Through this scheme, we reduced the intrusiveness of operating system services and improved

the performance of the incremental checkpointing facility in terms of latency and throughput.

Finally, the point 4 in the list addresses the reduction of intrusiveness and delay in scenarios where we may need to inspect a past state of the simulation along its trajectory during execution. This problem, known as access to the committed global state, has been tackled through state-swapping, to align with state-of-the-art approaches. We tackled the scenario of a PDES platform running on top of a multi-core shared-memory machine that exhibits a fully-shared workload, which complicates the identification of the committed global state. This is due to potential interference among worker threads targeting the same simulation object for the state-swapping activity and regular simulation execution. We also addressed the delay of the state-swapping operation, which is a crucial point in PDES applications to enable online output collection. The results obtained by the solution proposed in this thesis show a reduced latency of the overall operation, regarding both the state-swapping and the output collection, compared to a classical synchronous approach to perform the operation.

While some solutions have dealt with common aspects to the ones tackled in this thesis, i.e. load-balancing and LPs scheduling mechanisms [25, 107, 146, 148], window-based protocols [6, 90], cache locality awareness [141, 147], state-swapping for identifying a committed global state [21], they are only devised for traditional PDES architectures, either cluster-based or presenting long-term binding, so do not target at all the evolved PDES platforms on multi-core shared-memory machines that exhibit a fully-shared workload, and do not deal with the above-mentioned limitations regarding NUMA architectures. Therefore, we

aim to fill this gap by improving memory management for speculative PDES on multi-core shared-memory machines, tackling memory hierarchy awareness and exploiting the Linux kernel's capabilities, embedding the devised solutions in a state-of-the-art speculative PDES platform.
All the approaches have been implemented and integrated into the open source Ultimate-Share-Everything (USE) speculative simulation platform [54], and all the solutions will be thoroughly discussed, with a particular focus on the analysis of the obtained results and on the Linux kernel based facilities exploited.
We will further discuss some works in Chapter 3, highlighting how this thesis stands out with respect to the state-of-the-art.

1.5 List of Published Papers

The papers covering the aforementioned topics, which will be further discussed in relation to the state-of-the art, aim to propose solutions that fill the gaps in the existing literature, then validated through experimental data. They are listed below in chronological order of publication:

- Montesano Federica, Marotta Romolo, and Quaglia Francesco. "Spatial/temporal locality-based load-sharing in speculative discrete event simulation on multi-core machines." *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* 2022.

- Marotta Romolo, Montesano Federica, and Quaglia Francesco. "Effective access to the committed global state in speculative parallel discrete event simulation on multi-core machines." *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* 2023.

- Marotta Romolo, Montesano Federica, Pellegrini Alessandro, and Quaglia Francesco "Incremental Checkpointing of Large State Simulation Models with Write-Intensive Events via Memory Update Correlation on Buddy Pages." *2023 IEEE/ACM 27th International Symposium on Distributed Simulation and Real Time Applications (DS-RT).* IEEE, 2023.

- Montesano Federica, Marotta Romolo, and Quaglia Francesco, "Lightweight Operating System Services for Incremental Checkpointing in Speculative Discrete Event Simulation on Linux Platforms",*2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)* IEEE, 2024.

- Montesano Federica, Marotta Romolo, and Quaglia Francesco. 2024. Spatial/Temporal Locality-Based Load-sharing in Speculative Discrete Event Simulation on Multi-core Machines. *ACM Trans. Model. Comput. Simul. 35, 1, Article 2 (January 2025), 31 pages.*

Along with the thesis development, the participation to several PhD Symposia from different conferences brought to the production of some papers describing the aspects tackled during the PhD, which helped me incrementally refining the idea behind the thesis. These are:

- Montesano Federica. "Towards the Optimization of Memory and Data Management in Speculative PDES on Multi-Core Machines." *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* 2024.
- Montesano Federica. "Full-Stack Revision of Memory and Data Management in PDES on Multi-Core Machines." *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing.* 2024.
- Montesano Federica, "Towards the Improvement of Memory Management in PDES Systems on Multi-core Shared-memory Machines", *2024 IEEE/ACM 28th International Symposium on Distributed Simulation and Real Time Applications (DS-RT).* IEEE, 2024.

Chapter 2

Preliminaries

We now introduce some theoretical concepts in order to better understand the issues described in the previous chapter regarding the capabilities of multi-core shared-memory machines in the context of HPC, and why some optimizations are needed. We then proceed to introduce the Discrete Event Simulation (DES) paradigm, and consequently Parallel Discrete Event Simulation (PDES), to outline the focus of the innovative solutions proposed in this thesis. We highlight how the evolution of the PDES paradigm has uncovered several issues that need to be addressed in order to fully exploit the aforementioned machines.

2.1 Memory Hierarchy

The importance of considering memory hierarchy when designing applications has grown with advances in the performance of processing units. In fact, we have already mentioned that the performance gap

between CPUs and memory has been increasing for years, and it is even more evident in multi-core machines as the number of cores grow. The increase in core count has lead to higher contention for shared memory resources. This makes the memory access pattern more critical to overall performance. Hence, designers of memory hierarchies have focused on reducing the access time, determined by the cache miss rate, miss penalty and cache access time, but more recently much focus has been put towards energy consumption, since it has been found that in some architectures caches can account for 25% to 50% of total energy consumption, while data movement to/from main memory can account for 62.7%, on average, of total energy consumption for a wide set of applications [9].

Other than that, advances in specialized memory technologies, like High Bandwidth Memory (HBM) and Non-Volatile Memory (NVM), provide new opportunities and challenges for memory hierarchy design. These technologies offer improvements in bandwidth and power efficiency, but they also necessitate more sophisticated memory management techniques to fully leverage their benefits [8, 60, 78].

What we are interested in is how memory hierarchy, up until the RAM level, affects the performance of a parallel application.

Ideally, we would like to have zero latency, infinite capacity and infinite bandwidth to support applications accessing memory in parallel. Other than that, we would like to minimize the costs associated to memory accesses. Unfortunately, these requirements oppose each other, since a larger memory is typically slower, due to the time needed for searching for the required location. Furthermore, modern machines present a

shared-memory architecture, which exacerbate these costs due to multiple cores accessing data simultaneously, as we already mentioned in Chapter 1 and will further discuss in Section 2.2.
The idea behind the organization of memory in a hierarchical way is based on the different latencies of each component: whenever a memory area located on some level h is touched, it is moved to a higher cache level $h + 1$. The reason why this is beneficial lies in a principle called locality of reference [27].
The locality of reference is a principle followed by processors, according to which a set of memory locations is more likely to be accessed in a short period of time. It is based on the predictability of the processor's behaviour. The two main types of locality are spatial locality and temporal locality:

- Temporal locality refers to the reuse of a memory area in a short period of time,
- Spatial locality, also known as data locality, refers to the use of a set of memory areas in relatively close locations.

Hierarchical memory takes advantages of the locality of reference principle. In fact, by having multiple cache levels (e.g., L1, L2, LLC) that store copies of frequently accessed data, processors can quickly retrieve data from faster memory rather than repeatedly accessing slower main memory. This approach reduces the average latency for memory accesses, which is especially important in high-performance applications.

Given the significant energy consumption from memory accesses, considering spatial and/or temporal locality is becoming increasingly important.
In the next subsections, we will describe the main characteristics of the memory system, starting from the caching hierarchy, moving on to the main memory, with a particular consideration of Non-Uniform-Memory-Access (NUMA) architectures, to have a broad comprehension of the challenges of designing memory-aware applications on a multi-core shared-memory machine.

2.1.1 Cache memory

Cache memory is one of the layers in the memory hierarchy, it is closer to the CPU and has lower latency than RAM. In fact, it stores recently (and frequently) accessed data in order to allow faster access from the CPU, in compliance with the locality principle. So, key features of the caches are speed, being faster than RAM, and proximity, since it is located closer to the CPU (or on the CPU itself in some cases). The goal of using caches is to reduce the access time and the traffic to/from RAM in terms of interconnection pressure, in order to increase overall CPU utilization and system scalability. However, in order to achieve this, we must accurately design our applications to take into account these hardware components.
The cache itself is organized in a hierarchical manner:

- L1 cache: it is the first level of cache, typically placed inside the processor. It is the fastest level of cache but also the smallest, its

size is typically of 64KB, while its access time is on the order of 1 ns.

- L2 cache: it is the second level of cache, it might be placed inside the processor or not. If it is not present, it can be shared between processors in some architectures. Its size is about 256KB, and its access time ranges from 3 to 10 ns.
- Last Level Cache (LLC): it is typically shared between all/some processors, depending on the architecture. It is used to increase the performance of the previous levels of cache. Its size ranges from 4 to 64 MB, while its access time ranges from 10 to 20 ns.

Each cache level stores data in fixed-size blocks called cache lines. A cache line typically contains 32 to 128 bytes of data, and when the processor accesses data, it loads entire cache lines from the memory hierarchy. Caches are designed based on whether their content is shared across the levels or unique to specific levels. In an inclusive cache, for example, all cache lines in a higher-level cache are guaranteed to also reside in a lower-level cache. Conversely, in an exclusive cache, each cache level contains unique data, so a higher-level cache can contain cache lines not present in a lower-level cache.
To understand how the cache works and how to improve performance using caches, we must consider the flow of activities of the memory controller when memory is accessed. When a CPU accesses memory and finds some data in the cache (at any level), it is referred to as a cache hit. When, instead, the required data is not present in the cache, it is referred to as a cache miss. There are various types of

cache misses, based on the reason why the miss occurred and on the type of access, e.g. conflicting access on the same location, or first access to a memory location. An inclusive cache has the benefit of having a lower miss latency, because it is more likely that some data missing from a higher level cache is present in some lower level cache. However, one drawback is that its capacity is limited by the LLC's. On the contrary, an exclusive cache suffers from more overhead to solve cache misses, but its capacity is determined combining the capacity of the entire hierarchy.
Anyway, when a cache miss occurs, the required data must be brought into the cache from either a lower cache level or from the RAM, and it is managed through replacement policies. A typical replacement policy implemented is the Least Recently Used (LRU): basically, the controller evicts the least recently accessed block from the cache to make room for the new data brought from a lower level. Modern processors, of course, implement an approximation of the LRU policy, taking into account the design features of different architectures [48, 150].
We have discussed how caches work when a CPU wants to read data, but since it might also want to write data, we need to understand how different levels of cache are kept consistent with each other. There are two main strategies:

- Write-through: when the CPU writes data to the cache, it also issues the update through the hierarchy up to main memory,
- Write-back: when the CPU writes data to the cache, it only updates the copy in the cache. When a cache line is about to be

replaced, it is written back to main memory.

On the one hand, having a multi level cache allows reducing the access time because it is more likely to find data in some cache level, improving performance, CPU utilization and energy efficiency; on the other hand, it adds complexity to the overall memory organization, and if data is not found in any of the cache level it increases latency for cache misses. Furthermore, as the number of cache levels increases, higher costs are associated to the cache coherency protocols. So it is particularly relevant to know how caches work to design and implement parallel applications on multi-core shared-memory machines, in order to fully exploit their capabilities.
In the next subsection, we will describe the last memory level of interest for this thesis, that is main memory, namely RAM.

2.1.2 Main Memory

Main memory is also referred to as Random Access Memory (RAM). Unlike caches, it holds larger amounts of data and communicates directly with secondary storage (e.g., SSDs or HDDs).
The general characteristics of RAM are the increased speed with respect to the disk (or secondary storage), even though it is slower than the caches, and the larger capacity. The types of RAM are: Dynamic RAM (DRAM), which is the most commonly used for RAM systems due to its overall costs; Static RAM (SRAM), typically used for caches; Double Data Rate (DDR), used for improving speed and efficiency in modern RAM designs.

As we mentioned at the beginning of this dissertation, RAM has become a bottleneck in modern applications due to the widening performance gap vs the CPU, one reason being the increasing capacity [76]. In fact, in multi-core architectures, main memory must serve multiple cores simultaneously, leading to contention and requiring memory controllers to manage accesses efficiently both in terms of latency and of bandwidth. On the one hand, efficient memory access ensures that data is fetched and returned to the CPU faster, directly improving application performance. On the other hand, DRAM bandwidth is limited, and excessive data movement can saturate it, and this is especially relevant in Non-Uniform Memory Access (NUMA) architectures, as we will further describe.
The previously mentioned aspects are especially relevant in multi-core shared-memory systems, in which multiple cores compete for DRAM access, and the complete disregard of memory access patterns exacerbates contention and leads to performance degradation. They are also relevant in NUMA architectures, due to their natural asymmetry, since remote memory accesses generate more delay and more pressure on the interconnection network than local ones.

2.1.2.1 NUMA architecture

In a NUMA architecture, RAM is organized in multiple memory banks, namely nodes, at different distance from each other, on which the processors are placed. The address space is shared between the processors, meaning that each processor can access all the memory available, but processors are placed in a way such that they manage a local memory

and a remote memory. This implies that each processor has faster access to its local memory and slower access to remote memory banks. The nodes in the NUMA architecture are connected through an interconnection network, in order to allow remote memory accesses. On the one hand, this allows decoupling memory accesses across processors, on the other hand accessing remote nodes not only make the processors experience a longer latency, but it also creates pressure on the interconnection network. This is why memory placement, locality-awareness and load-balancing are crucial aspects to be tackled for parallel applications running on top of these systems.
NUMA architectures are a particular design of shared-memory architectures, discussed in the next section.

2.2 Shared-memory Architectures

In shared-memory architectures, all processors have access to the entire address space available [37]. This allows them to communicate directly through shared-memory, instead of communicating through message-passing. As mentioned in the previous subsection, NUMA is a particular type of shared-memory system, in which the latencies of processors accessing memory depend on their placement in the architecture. We note that the shared-memory paradigm can also be implemented in Uniform Memory Access (UMA) systems.
Anyway, a shared-memory architecture needs some mechanism to coordinate memory updates in order to maintain consistency across processors, and this is achieved through cache coherency protocols, such

as MESI or MOESI).
The main advantage of shared-memory systems is the ease of communications between processors, due to the avoidance of message passing facilities [59]. However, data sharing forces to devise some synchronization mechanism in order to avoid data races across processors. Furthermore, scalability becomes an issue, since with increasing number of cores memory contention's overhead becomes non-negligible. Careful design is needed in order to solve or reduce the impact of these issues.

2.2.1 Cache Coherency

Cache coherency ensures that multiple copies of data stored in various caches (in multi-processor/multi-core systems) remain consistent. The reason why it is necessary resides in the nature of shared-memory systems. In shared-memory systems, multiple processors or cores can access and modify the same memory locations, leading to potential inconsistencies if updates are not propagated correctly.
Some challenges related to the cache coherency protocols are: possibility of stale data (e.g. if one core modifies a memory block, other cores might still operate on outdated cached copies), write propagation, (since changes made in one cache must be visible to other caches and the main memory) and scalability (since as the number of cores grows, maintaining coherency becomes increasingly complex and resource-intensive). As mentioned, popular coherency models are MESI and MOESI [93], and the main drawbacks of any cache coherency model are the possible latency due to invalidation or synchronization delays, and the bandwidth bottlenecks from frequent communication between

caches [32, 47, 140]. Hence, these issues must be taken into account when designing parallel applications in shared-memory machines.

2.3 Virtual Memory

Virtual memory has been devised to create an abstraction of the memory view, such that the processors can manage a larger contiguous address space regardless of the actual physical memory available. In order to effectively manage memory, there is a Memory Management Unit (MMU), that maps virtual addresses to physical addresses.
The way virtual memory has been traditionally devised is through paging or through segmentation. On the one hand, paging divides virtual memory into fixed-size memory blocks, namely pages, that will be mapped into physical pages, namely frames. Traditionally, the minimum page size is 4KB. The mappings of virtual addresses into physical addresses is maintained into a page table, which is organized in a multi-level manner in modern operating systems in order to cope with the many optimizations of the processor architecture. The actual address mapping, and also other auxiliary information regarding the usage and management of the page, are kept in the lowest page table level, namely page table entry (PTE). The advantage of paging is the reduced external fragmentation rate due to the fixed-size. The disadvantages are the latency of the look-up activity into the page table, and the possibility of internal fragmentation, e.g. when the page is not fully utilized.
Segmentation, on the other hand, divides memory into variable-sized

chunks of memory, namely segments, based on the logical division of code, data, stack segments. One advantage of segmentation is the low latency of accesses, but it suffers from external fragmentation, due to the different size of the segments.
Segmentation and paging are two ways of implementing memory protection, which controls access permissions to specific locations, allowing or forbidding a process from accessing particular memory regions. These two methods can be combined. Protection mechanisms are used for security purposes, or to prevent unauthorized access to kernel-level memory areas. To support such a mechanism, each page is equipped with attributes describing the protection level, namely read-only, write-only and execute-only, and any combination of the three describes the possible management of the pages.
Typically, pages are retrieved and loaded to main memory only when accessed, exploiting memory hierarchy as described above. However, we have discussed how cache misses can occur when pages are not present in any cache level, but it can also happen that a page is not present in RAM. In that case, a page fault occurs, and the page has to be fetched from secondary storage. One problem arising from frequent page faults is the thrashing phenomenon, in which the CPU spends more time swapping pages than actually executing tasks.
Particularly relevant, in the context of page accessing optimizations, page fault management included, is the role of the Translation Lookaside Buffer, as we will describe in the next subsection.

2.3.1 The Translation Lookaside Buffer

The Translation Lookaside Buffer (TLB) is a cache that stores the recently accessed virtual-to-physical address mappings, namely PTEs. In a multi-core system, there is one TLB for each hyperthread. Obviously, its goal is to speed up memory translation for frequently accessed pages, since at each access the MMU first queries the TLB, and if necessary (e.g. in case of a TLB miss) it consults the page table. There are many optimizations to avoid TLB thrashing, namely frequent TLB misses, both hardware and software side (e.g. enlarging the TLB size or prefetching the address mappings).
The TLB can have a considerable impact on performance on multi-core shared-memory machines, and it is tightly bound to memory permissions management. In fact, it not only stores the virtual-to-physical address mappings, but it also records metadata related to the PTEs, such as the permissions.
As stated in the previous section, permissions determine the capacity of an application to perform certain operations on a given memory page, preventing illegal accesses. If a memory access violates the permissions, e.g. a write operation on a non-writeable memory page occurs, a page fault is triggered, namely a protection fault. When memory permissions are changed, the Linux kernel carries out many activities, one of them being the invalidation of all the TLB entries relative to the page, or range of pages, targeted by the fault. This is referred to as TLB shoot down or TLB flushing, and it has to be done in order to maintain the consistency of the mapping across all cores, since an update of the page table occurred (namely, the permissions changed).

In fact, changing the permissions of a memory area means updating the metadata relative to the PTEs in the page table, so the previously stored information in the TLB might become outdated, possibly causing incorrect translations. That is why the TLB shoot down is needed. As long as the system is single-core, it is enough to only flush the local TLB, but in a multi-core machine, the coordination mechanism is more complex. In fact, in such a system, the kernel must notify all the cores to flush their local TLBs for the affected pages, and this is done via the Inter-Processor-Interrupt (IPI) architecture. After the IPIs are sent, each core processes them by invalidating the range of pages in its local TLB and then resuming its previous activity. However, this can cause performance degradation due to the overhead involved in the synchronization process, which requires communication and coordination across cores. Furthermore, as the number of cores increases, the TLB shoot down mechanism can become a bottleneck of applications exhibiting particularly memory intensive operations, leading to poor performance. In order to mitigate the effects of the TLB shoot down by software approaches, a lazy invalidation mechanism might be adopted, with the risk of having incorrect translations in the meantime, or batching the page table updates (e.g. the permission changes) in order to reduce the number of shoot down signals. Otherwise, kernel level facilities might be leveraged in order to have a finer-grain invalidation mechanism, directly operating at page table level.
We note that there are many other reasons why a TLB shoot down might be needed, e.g. after a page is swapped out, or when a context switch occurs.

Given this overview of memory systems, we can dive into the discussion on the discrete event simulation paradigm, target of the solutions presented in this thesis.

2.4 Discrete Event Simulation Systems

Simulation is a paradigm that enables the imitation of real-world scenarios or phenomena to study their evolution and collect results for data analysis without requiring an analytical solution of the system. It allows testing what-if scenarios by only changing the configuration of the system. To achieve this, a simulation model is defined to represent the target phenomenon, along with the data structures and variables that constitute the model's state. The simulation kernel, instead, is the component that drives the model execution and takes care of several aspects such as the correctness of the simulation.
Simulation applications fall into two categories, depending on how the phenomenon evolves through time: continuous simulation and discrete simulation. The former one refers to evolution of the simulated application along a continuous time axis. To determine a system change, mathematical formulas (i.e. time series analysis, differential equations) are used. The latter, which is the one of interest for this thesis, is called Discrete Event Simulation (DES), and refers to the fact that the progress is based on events that occur at discrete points in time (the timestamp of an event), rather than continuously. The system's state only changes when an event occurs, and to enable event execution, a

DES implementation consists of a set of event-handler functions (callbacks), that are executed by the simulation kernel allowing the state of the simulation to change. In order to correctly support the progress of simulation along discrete time steps, causality management is needed, and a virtual clock is adopted. In fact, we highlight that it is important to distinguish virtual time, which is an abstract notion of time relative to simulation execution, from Wall-clock Time (WCT), which is the elapsed time measured by the machine. So, in order to guarantee a causally consistent model execution, events must be processed in timestamp order.
In fact, a DES system is equipped with an event list, namely a priority queue, that contains the events to be processed ordered by their timestamp. The simulation engine picks an event from the event list and processes it, moving the virtual clock forward. When executing an event, other events might be generated, and they are inserted in the event list for future processing. Hence, the event list ensures that the events are processed in time order, maintaining logical consistency in the simulation model.
DES is useful for several reasons, allowing to model complex systems, especially when the state does not change in a continuous manner, but can be described by the occurrence of events. DES is used in a broad range of applications:

- Manufacturing/Logistics: to optimize supply chain, resource allocation, and inventory management.
- Healthcare: staffing and resource management, patient flow, and emergency scenarios simulations.

- Transportation: to analyse traffic patterns, congestion, crossroads management, and optimize routes.
- Telecommunications: for network traffic management, peak traffic and network congestion, resource allocation (e.g. in cloud providers), and server optimization.
- Finance: for risk analysis for investments, financial markets, and fraud detection.

Along the path of hardware evolution, simulation also required executing very large and complex models. Hence, due to the establishment of multi-processors/multi-core systems, DES systems had to keep up, and so the Parallel Discrete Event Simulation (PDES) paradigm was devised, allowing simulations to be run in parallel across multiple processors or computing nodes. In fact, parallel simulation can be executed by multiple processing units residing on the same machine, or across multiple nodes geographically distributed, in a cluster-based manner. One of the goals of parallelizing the simulation is to reduce the overall execution time compared to a sequential DES execution, namely to achieve higher speedup.
In the next section, we will discuss the PDES paradigm.

2.5 Parallel Discrete Event Simulation Systems

As already mentioned in Chapter 1, in a PDES system the model's state is partitioned into many Logical Processes, namely LPs, and they communicate with each other by exchanging messages (namely

events). Each LP represents a portion of the simulation state, and consists of a set of private variables completely disjoint to the other LPs. Therefore, an LP is an independent entity that implements its own event-handlers to execute events, generates events destined to other LPs (or to itself) and updates its own state. A parallel implementation of the DES model involves executing multiple LPs, and therefore their event handlers, in parallel across multiple processors, cores, or interconnected machines. [12, 16, 35, 39, 41, 54, 80, 94].
Each LP maintains its state to keep track of the changes caused by event execution, and does not interfere with the other LPs' states, since the communication among LPs happens via message exchange (e.g. when it generates events destined to some LP). Events destined to each LP must be processed in virtual time order, this is why each LP keeps a Local Virtual Time (LVT) variable, which represents the current virtual time for that LP, in order to avoid any causality violation. Moreover, each LP has a queue of events to process, ordered by timestamp, similarly to what would happen in the case of sequential DES systems. This guarantees that for each LP, causality of event processing is preserved.

However, while in sequential event processing it is enough to have a timestamp ordered queue of events in order to guarantee causality, in parallel event processing some synchronization mechanisms to guarantee global causality and consistency are needed. There are two main categories of synchronization algorithms: conservative and optimistic. The former one guarantees that no event is processed out of order, and so that no causality violation ever happens. In fact, if an LP contains

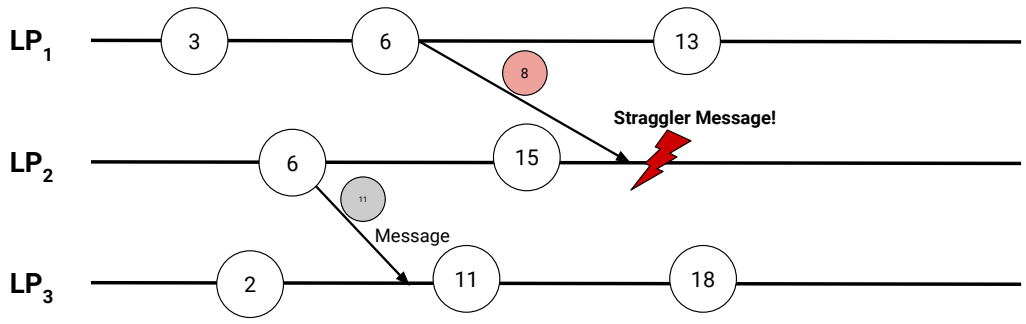

FIGURE 2.5.1: Example of a Causality Violation in speculative PDES

an unprocessed event e_t and it can guarantee that no event will be received with timestamp $t' < t$, then it can safely process e_t. This requires having a lookahead, which is a simulation attribute that defines the minimum temporal distance in the future before an LP can generate events, ensuring correct causality. The latter method, instead, allows processing events out-of-order regardless of causality violations. But since it is likely to receive out-of-order messages violating causality, i.e. as shown in Figure 2.5.1, the simulation engine must support the reconstruction of a correct state in order to possibly recover from causality violations. In the above-cited figure, we can see how causality violations can occur: in particular, each LP is processing its events in timestamp order, and it is sending/receiving messages to/from other LPs. When LP_2 receives the event with timestamp equal to 8, its local time is set to 15, since it is the timestamp of the last executed event. Hence, it receives an out-of-timestamp order message, called *straggler*, which causes a causality violation. In the next subsection, we will thoroughly discuss optimistic simulation, implemented by the Time Warp algorithm.

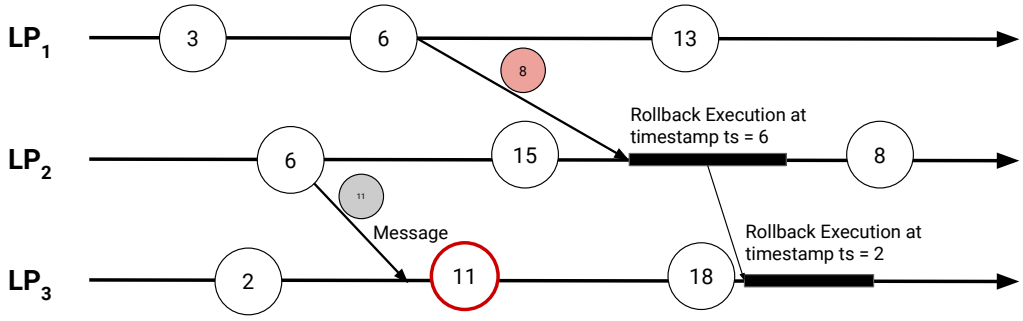

FIGURE 2.5.2: Example of a Rollback Procedure after a Causality Violation

2.5.1 Time Warp

The Time Warp algorithm was introduced by Jefferson in 1985 in order to implement virtual time [57]. Virtual time is a paradigm used to organize and coordinate distributed systems by defining a shared notion of time and synchronization among them. We have already specified that at each LP, events are handled in timestamp order. The main idea of Time Warp is based on processing an event as it occurs, assuming it is in global order, updating the state and generating events destined to other LPs. If the processing turns out to be incorrect (e.g. an LP receives a straggler message as in Figure 2.5.1), a rollback procedure is executed in order to re-align simulation execution to a causally consistent state prior to the straggler event. Then all the processed events after the causality violation occurrence are cancelled, and their effects are neutralised, before re-processing events in the correct time order. This scheme is exemplified in Figures 2.5.1 and 2.5.2. When introducing causality violations, we described Figure 2.5.1 to show what a straggler message is. In Figure 2.5.2, we show how to recover from

causality violations, as just hinted. In particular, in this figure, LP_2 has received a straggler message, so in order to re-align its state to a correct one, there is a rollback execution to, for example, timestamp 6. It is also necessary to neutralise the effects of sending and executing the event with timestamp 11 on LP_3, since it occurred in a no more valid trajectory, this is why there is also a rollback execution for LP_3. This figure also shows that cascade rollbacks can happen.

Many implementations of Time Warp came in succession over the years, considering PDES running on top of shared-memory multi-processor/multi-core machines [13, 17, 38, 96]. Time Warp variants, such as Bounded Time Warp, have also been compared to conservative synchronization algorithms to further characterise Time Warp's performance [29].

In the next subsections, we will discuss the virtual time paradigm, namely Local Virtual Time and Global Virtual Time, and the state recovery mechanisms, namely checkpointing and rollback.

2.5.1.1 Local Virtual Time

As previously mentioned, each LP maintains a local view of virtual time, known as Local Virtual Time (LVT), which measures the progression of event processing for that individual LP independently of the others. Each LP has an input queue where all received events from other LPs are stored in increasing order of their timestamps and are progressively picked for processing. This is an ideal execution, in which it is assumed that no out-of-order messages are received. However, we

mentioned that this might not be the case because of different processing rates of the LPs, so when an LP is picking an event from the input queue, it is actually assuming that it will not receive any other message with a lower timestamp than the one it will be processing, as envisaged by the optimistic algorithm. As long as this assumption holds, the execution proceeds correctly, but if it does not hold, it encounters a causality violation, and the event causing the violation is called a straggler (see Figure 2.5.1). In order to solve this problem, the LP has to roll back to a past state with a virtual time lower than the timestamp of the straggler event, possibly cancel intermediate side effects and then execute forward to re-align the simulation state, as already described in Figure 2.5.2.

In a parallel and distributed environment, rollbacks can have cascaded effects due to LPs sending messages to others. In fact, it might happen that the LP in a rollback phase, has sent some messages to other LPs before receiving the straggler message, causing side effects in their execution, and so on. All the in-transit and received messages must be either cancelled/unsent or their effects must be reversed. Time Warp manages this without freezing the simulation. For every event, there exists an anti-event or anti-message, which is a copy of the original event except for the sign. In fact, a correctly processed event has a positive sign (+), while an anti-event has a negative sign (−). When an LP sends a message, a positive copy of that message is sent to the receiver's input queue, while the negative copy is maintained by the sender and used in case of rollbacks. In fact, if a causality violation occurs, after the rollback operation, the side effects of that violation must be reversed, and there are two situations that might occur:

- The original event e_x was sent but not yet processed. If the timestamp associated with the received anti-message is greater than LVT_x, the original event has not yet been executed along the LP_x's trajectory, so it is enough to discard e_x. In this case, no rollback is necessary, but the anti-event is enqueued, causing a message annihilation, leaving the receiver unaware that the event ever existed.
- The original event e_x was sent, and it has been partially or totally processed but the receiver. So, if the anti-message has a timestamp t' such that $t' \leq LVT_x$ (the $=$ refers to the case of exactly processing the event relative to the anti-message), the state of LP_x has already been subjected to some changes, and some side effects have occurred. So a rollback is necessary on LP_x and the anti-message annihilates the original event, and this also happens in a cascade manner for other LPs if necessary.

The operation of rollback is crucial in optimistic synchronization, and there exists two main approaches to tackle this issue: state saving, which is the one of interest for this thesis and can be full, incremental or hybrid approaches combining the two (see [36, 90, 103–106, 115–117, 129, 131, 149]), and reverse computation [15, 19].

2.5.1.2 Global Virtual Time

The local virtual time management just described manages the advance of time as seen from the LP's perspective, but in order to coordinate

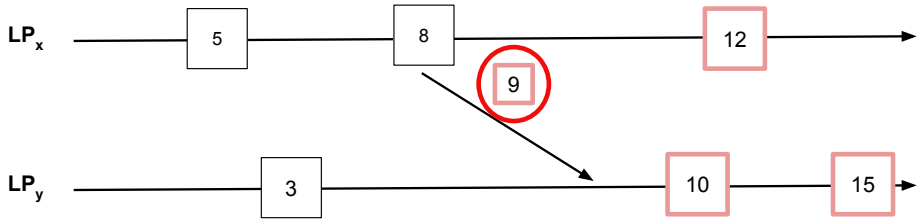

FIGURE 2.5.3: Example for Explaining the GVT

multiple LPs in terms of simulation progress, to guarantee the simulation termination and to manage the resources of every LP, we must define a global control mechanism. This is called Global Virtual Time (GVT) [42], and it is defined as follows, and depicted in Figure 2.5.3:

Definition 2.1. The GVT is defined as the minimum between (i) all virtual times in all virtual clocks currently in the system, and (ii) the virtual times of all messages in transit and not yet processed.

Since the GVT never decreases, it serves as a bound for the virtual time to which any LP can roll back at some point. The GVT is generally considered a measurement of simulation progress, since in the case of successful event processing it increases, and it is viewed as a commit horizon, since it is not possible to roll back to a time lesser than the current GVT.
A fundamental aspect regarding the usage of the GVT in Time Warp is memory reclamation, in order to reduce the memory usage for saving the state. In particular, any message relative to an already processed event, whose timestamp is lower than the GVT can be discarded, as it is impossible to receive an event with lower timestamp. As for the

past states in the state queue, all but one saved state before the GVT for each process can be discarded, and their memory can be reclaimed: this is known as fossil collection. Hence, the GVT is fundamental information used to recognize and discard obsolete data that is no longer useful for the simulation, and also to commit the events produced in the speculative execution, guaranteeing simulation progress. Over the years, many optimizations regarding the computation of the GVT have been studied, especially to exploit shared-memory machines [55, 95]. Furthermore, the GVT computation is fundamental for another core aspect characterizing speculative PDES, that is the identyfication of the committed global state. In fact, a committed state, namely S_x, of any simulation object o_x always has virtual time lower than or equal to the last computed GVT. However, the same object might have a current state S'_x with virtual time larger than the GVT value. State S'_x cannot be used for inspecting the execution trajectory of that simulation object and for producing output data, since it is still susceptible to possible inconsistencies due to out-of-order processing of events, which can always occur beyond GVT. Hence, when we compute a new GVT value, we would need to observe a past state of o_x in order to perform a correct inspection of the execution trajectory of the simulation object. This is relevant in the context of online output collection and/or predicate detection.

2.5.1.3 State Saving and Causality Violations Recovery

Along with the definition of virtual time and the notions of local virtual time and global virtual time, the Time Warp paradigm has been conceived to manage possible causality violations. In fact, in the previous subsections it has been mentioned that out-of-order processing of events across LPs might give rise to out-of-timestamp order errors, causing violations solvable by a rollback procedure to a past state for the target LPs.

In order to support the rollbacks, we must periodically save the state of the LPs, and this is particularly relevant in optimistic PDES. Several approaches have been studied over the years, from full state saving, i.e. periodically saving the entire state of each LP (see [36, 90, 103, 104, 106, 115, 129]), to incremental state saving, i.e. periodically saving only the modified portions of the states (see [98, 116, 120, 149]), to reversible computing, i.e. implementing a reverse-event for each event handler (see [15, 19]).

The state saving technique, in general, refers to the activity of saving a copy of the state of any LP_i after the execution of the event e_t, occurring at timestamp t, and this is called checkpoint. Obviously, taking a checkpoint after every event execution is not only a time-consuming operation, but it also affects memory usage, so the sparse state saving technique is usually adopted. In this way, the checkpoint is taken periodically. On the one hand, it allows reducing the memory used; on the other hand, if a causality violation occurs and rollback must be performed, the realignment phase might be longer, e.g. if the period between two checkpoints is too large. In fact, we mentioned that after

restoring a state, we must re-align the state of the simulation by executing the already processed events, straggler included. This operation is called coasting forward and consist of re-processing the events in a silent manner, without re-generating events. Many works focused on effective policies to tune the checkpoint period (see [103, 106, 115]).
A state saving technique that allows reducing the memory footprint is the incremental state saving. The crucial aspect of the incremental state saving is to identify the portions of the LP's state that have been modified during the event execution, in order to build a checkpoint that is useful to restore the state during the rollback procedure [98, 116, 120, 149]. This is used to reduce the size of the taken checkpoint and save clock cycles with respect to the full state saving. When a rollback must be performed, in this case, the chain of incremental checkpoints is traversed backwards to identify the correct state from which to begin re-aligning the simulation's trajectory. At the same time, the incremental checkpoint can be taken periodically as well, to optimize the trade-off between the checkpoint cost and the state restoration cost.
Furthermore, a series of incremental checkpoints can be alternated with infrequent full checkpoints, which helps to discard obsolete logs preceding the last computed Global Virtual Time (GVT) in a speculative simulation run. In particular, an infrequent full log with a timestamp less than (or equal to) the GVT value enables the discarding of all the preceding incremental checkpoints—since the restoration of some state will find all the requested parts to be restored by backward traversing the incremental checkpoint chain up to that full checkpoint at worst.

2.5.1.4 Reverse computation

For the sake of completeness, we briefly describe the second main approach to recover from causality violations, called reverse computation [14]. This is based on the idea that for each event e there is a reverse event e_r capable of restoring the previous state, by executing in reverse order the operations of e. Of course, this is an effective and simple approach when events execute basic operations such as arithmetic operations, but it can be become more difficult for events presenting if-else structure or non-reversible operations, especially if they involve memory pointers.
However, many solutions can be devised to take care of these problems, one of them described in [19], but they will not be further discussed, since it is out of the scope of the thesis.
In the next section, we will discuss why and how PDES architectures have been evolved over the years.

2.6 Reshuffle of the PDES Architecture

Having described all the building blocks of DES and PDES, we can now focus on the software architecture and on its development and evolution through the years.
Classical PDES architectures consisted on the partitioning of the simulation in multiple kernel instances, each of them managing a subset of LPs. The programming model is referred to as multiple sequential executions on different machines. This partitioning was devised in order to exploit cluster-based machines, where multiple nodes operating

independently were interconnected via a network. Over the years, a reshuffle of the architecture has been necessary due to the establishment of multi-processor/multi-core machines [12, 13, 35, 38, 41]. This favoured the transition from distributed memory systems to shared-memory systems. According to this shift, the system consists of multiple simulation kernels, each of them managing multiple cores, leveraging the multi-thread paradigm, in which each worker thread (WT) executes the simulation (the terms "worker thread", "thread", and "core" are used interchangeably in this thesis). This scheme applies to a single or multiple machines. The exploitation of finer-grained parallelism allows increasing the parallelism and avoiding the overhead needed for inter-node communication, thanks to the exploitation of shared-memory machines architectures [94, 118].

However, a further observation has to be made: in fact, in both the above-described cases, the binding between a WT and a set of LPs is long-term, meaning that a WT executes events destined to one of its own subset of LPs and manage its states, until there are processable events. Conversely, an explicit rebalancing is required to change the binding between a WT and one or more LPs. This is inherently limiting the performance of PDES, since it does not fully exploit the computing power of the underlying hardware for simulation operations. Other than that, such a paradigm fails to exploit the capabilities of multi-core shared-memory machines, not allowing WT to actually share data and access shared data. Specifically, the long-term binding approach can lead to inefficient load-balancing, since each WT might have a different amount of events to process destined to their subset of LPs, leading to uneven workloads. Moreover, in such a system each WT could access

the entire memory available, but it is instead restricted to accessing the subset of LPs bound to them.

The opposite idea to the long-term binding paradigm used by traditional PDES systems implements a fine-grain sharing of resources, and it is called *share-everything PDES*. This avoids any partitioning of simulation objects among threads and creates a new scheme based on a short-term binding. It introduces a fundamental shift towards the workload distribution of PDES systems, since it allows any thread to dispatch any simulation object in the system at any time along simulation execution, enabling effective load-balancing and finer-grain sharing of resources. Hence, in the so-called short-term binding, the time period of the binding is equal to the processing time of an individual event [56].

The share-everything paradigm is therefore based on the idea that all the threads running the simulation can actually share data, in particular individual events destined to any LP. This is referred to as fine-grain sharing. In this paradigm, the LPs are not the main work unit, as it was in the previously described paradigm, but rather a container of work units, namely the events. The core concept is to deliver the computing power of the underlying machine to the next minimum timestamp event to be processed, fully exploiting multi-core shared-memory machines. This method ensures that the system continuously processes events without the risk of a bottleneck caused by the WT-LP binding, thereby enabling greater parallelism and speed. In order to do so, in [54] a non-blocking global queue of events has been devised, as well as non-blocking algorithms to manage the scheduling and dispatching of events, and the housekeeping tasks such as fossil collection. The

non-blocking data structure allows threads to freely access and process events in a non-sequential, concurrent manner, without halting each other, while the fossil collection, still managed in a non-blocking manner, guarantees the correct deletion of obsolete states efficiently. So, as a first step to evolve the entire PDES paradigm, the evolution of the pending queue of event has been necessary, also considering NUMA architectures [72, 73, 110].

This enabled higher efficiency due to the improved load-balancing, that also allows having lower rollback probability due to the continuous processing of events in the proximity of the commit horizon. Moreover, it avoids over-speculation and the divergence of virtual time across LPs.

Another important characteristic of this simulation platform regards the memory layout. In particular, the platform described in [54] supports models developed using the C programming language, and exploits the compile time interception of the calls to the malloc library API via the `-wrap` flag. This is done in order to make chunks hosted on a specific memory area available for use by an individual simulation object. In its turn, this memory area resides in a set of pages mapped in the address space of the PDES engine via the `mmap(..)` system call. At the level of the PDES engine, it is therefore possible to offer an API to set the state of the memory allocation system in order keep track, through Thread Local Storage (TLS) variables, of the simulation objects being executed. The allocation system can also select object-specific mapped memory areas for delivering the chunks. This mechanism allows to effectively manage the simulation objects states at page level.

However, the limitations of PDES platforms devised following this

paradigm can be disclosed as follows:

- They present a global pool of events that is shared among threads, and it has to be traversed and updated by each WT in the simulation. Even if the pool is managed by a non-blocking algorithm, that reduces the synchronization overhead, the cost of the traversal grows linearly with the number of events. It is therefore necessary to reduce the frequency of accessing it.
- The memory hierarchy is not taken into account for event processing. In particular, frequently switching between LPs' states when picking events to process increases the costs of cache misses, especially considering NUMA architectures.
- Operating systems' services might add non-negligible overhead, for example in the context of the MMU's internal operations. In fact, memory management entails cross-core coordination at kernel level when memory updates occur, as we discussed in Section 2.3, which involves also to broadcast TLB shoot downs. The main point regards the invalidation of multiple TLBs due to the memory permission changes performed by the operating systems' protection services. This multiple invalidation causes non-negligible overhead due to the fine-grain sharing of resources. We also note that this activity is endemic due to the kernel level activities performed by the memory protection services (namely, the IPIs to signal and coordinate all the TLBs).
- Accessing shared data structures also presents critical aspects with respect to the identification of a committed global state of the

simulation. In fact, fine-grained sharing of resources makes it more complicated to distinguish portions of state still susceptible to causality violations from the committed ones.

Overall, these limitations indicate that the PDES platforms based on this paradigm face some scalability challenges, that can hinder their ability to fully exploit the potential of multi-core shared-memory architectures.
In the next section, we will present the simulation models that represent the target for evaluating the solutions proposed in this thesis.

2.7 Simulation Models and Benchmarks

A simulation model is the representation of the target real-world scenario/phenomenon to be studied. In the evaluation of the solutions devised in this thesis, the target models of our experimental evaluation consist of a synthetic benchmark, PHOLD, and two real-world models, Personal Communication System (PCS) and Tuberculosis (TBC).

2.7.1 PHOLD

PHOLD is a synthetic benchmark [40], in which the execution of an event leads to updating the state of the target simulation object, which keeps track of statistics such as the number of processed events and the average simulation time advancement. It also leads to executing a classical CPU busy-loop for the emulation of a given event granularity. There are two types of events:

- Regular events, whose processing generated new events of any type,
- Diffusion events that do not generate new events when being processed.

The number of diffusion events generated by regular events, denoted as Fan-out, can be set as a parameter. This model do not represent any real-world scenario, but rather it abstracts an event-driven system with minimal complexity. It is usually used to test scalability and overhead of PDES platforms.

2.7.2 Personal Communication System

The Personal Communication System (PCS) application models a cellular network for personal communication [20, 58]. Each simulation object models the evolution of an individual hexagonal cell. Each cell can handle a number of N channels, modelled via power regulation and interference/fading phenomena. The records associated with channels are dynamically allocated/released upon start/end of calls, and are kept in a list that is managed by the object simulating the cell. Upon call setup, power regulation is performed, which involves scanning the aforementioned list of records to compute the minimum transmission power allowing the current call setup to achieve the threshold-level signal-to-interference-ratio (SIR). Each record is released when the corresponding call ends or is handed off to towards and adjacent cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Data structures keeping track of fading coefficients are

also updated while scanning the list, according to a model defining meteorological conditions and their variations. The set of configurable parameters includes:

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

These parameters affect the channel utilization factor, defined as $\rho = \frac{\tau_D}{\tau_A \cdot N}$. The value ρ impacts on the granularity of events, since the more the busy channels, the more power-management records are allocated and consequently scanned/updated while processing events. At the same time, higher values of the channel utilization factor lead to higher memory requirements for the state footprint of individual simulation objects. Also, CPU and memory demands are bounded depending on the number N of per-cell managed channels. In fact, when a call-setup operation is requested due to a call or handoff (switching between cells) arrival, if all the channels are already busy, then the call is dropped, mimicking the real world scenario where communication is interrupted whenever the base station has no available resources.
This model is useful for resource allocation optimization or traffic analysis, and in this thesis it has been also used to model memory intensive executions for testing the proposed solutions, as we will see in the next chapters.

2.7.3 Tuberculosis

The Tuberculosis (TBC) model has been developed at the Barcelona Supercomputing Center to simulate the spread of tuberculosis in the Barcelona area [81]. It is based on agents, namely individuals, circulating in an area represented by several simulation objects. Each simulation 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 simulation object that models the region. Specifically, the object maintains a record for each agent currently residing in the corresponding region. The different agents model individuals that may be in one of the five possible states, based on the evolution of TBC infection: *healthy*, *infected*, *sick* (i.e. with active TBC), *under treatment* and *cured* (i.e. completed treatment). 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). This model is useful to study the spread of the infection among the population, and also to help to plan treatment protocols and strategies.
In this thesis, it has been used for further testing the locality-aware solution presented in Chapter 4.

Chapter 3

State-of-the-art

Provided a broad overview of the systems we are targeting, and the challenges emerged due to the evolution of both the hardware and the simulation platforms, we can now dive into the literature. We will describe what has already been tackled regarding speculative PDES systems on multi-processor/multi-core shared-memory machines, which aspects have been dealt with in a more limited manner and how we aim to fill the gap.
We will focus on aspects previously described as critical regarding speculative PDES running on top of multi-processor/multi-core shared-memory machines, such as memory hierarchy and locality awareness, incremental checkpointing optimization and reduced delay of critical-path operations such as the identification of a committed global state, highlighting the differences between state-of-the-art approaches and the solutions presented in this thesis.

3.1 Memory Locality

Memory management of speculative PDES systems has been a challenge since its inception. In particular, some efficient mechanisms to handle allocation/deallocation and reclamation of buffers were needed, especially in the context of PDES running on top of shared-memory machines. The work in [43] tackles this issue, and it also discusses strategies for reducing memory overhead and avoiding bottlenecks caused by buffer contention in shared-memory environments. It further focuses on rollback efficiency and memory reuse, namely fossil collection. Still regarding memory reuse, the work in [136] aims to reduce the number of message copies to exchange data between threads, by developing a reversible memory allocation mechanism to reduce memory pressure and also avoid copying the message contents. These works focus on memory management from the allocation and reclamation points of view, which is still orthogonal to the proposal of this thesis, even though the cited works do not target NUMA architectures.
The work in [118] discusses how the setup of intra-node facilities, based on shared-memory, can allow multiple Message Passing Interface (MPI) ranks running on a same machine to reduce their communication overhead. This is relevant in the context of the evolution of PDES systems discussed in Section 2.6, exploiting multi-processor/multi-core shared-memory machines to achieve higher performance, especially for large-scale simulation. Along this path, the proposal in [143] introduces an architecture for speculative PDES where multiple threads

running on a shared-memory machine support a communication mechanism based on operating systems' top/bottom half primitives with reduced intrusiveness. This aims to support a dynamic adaptation to different workloads requirements/fluctuations, improving flexibility and scalability. Additional studies have been focused on the analysis of general architectural redesigns when shifting optimistic PDES (Time-Warp based) to shared-memory multi-processor/multi-core machines (see [38, 147]). Experimental results have shown that optimistic PDES on multi-processor/multi-core shared-memory machines can achieve higher performance under certain constraints, such as balanced workload. Both works also discuss some strategies to tackle challenging issues such as rollbacks and fossil collection, that have been also mentioned earlier as potential bottlenecks of PDES platforms. The first work, [38], provides a baseline for better understanding optimistic PDES on multi-core shared-memory machines, while the work in [147] provides further improvements and challenges related to memory hierarchy awareness. This shows that optimising PDES platforms on multi-core shared-memory machines is still a topic of interest.

Other proposals have been oriented to the definition of new approaches for event processing, like cross-state events involving multiple simulation objects [97], or the inclusion of state attributes that are accessible to other simulation objects through the exploitation of the transactional memory support [17]. The first work aims to reduce the causality violations costs, by explicitly tracking the state changes relative to event execution across simulation objects, instead of solely rely on the timestamps for detecting the causality violation. Despite aiming to reduce causality violations, which is also one of our objectives, we do

not explicitly track state changes, but we leverage the locality of reference principle to effectively process events. The second work enhances the Time Warp paradigm for multi-core shared-memory machines to improve resources utilization by exploiting transactional memories.
This thesis proposal differs from all these studies, since none of them is oriented to the short-term binding of simulation objects to worker threads, namely the evolution of PDES systems that brought to the development of the share-everything platform described in Chapter 2 [54]. Moreover, the above-cited works do not consider cache-aware association of simulation objects to threads, NUMA-aware placement/access of/to simulation objects' states and batch-processing of events of specific simulation objects along any thread. These are instead the main aspects dealt with in part of the thesis, namely for what concerns memory hierarchy awareness and cache/NUMA locality awareness for event processing (see Chapter 4).
As for solutions oriented to load sharing in multi-core machines, in combination with long-term binding of simulation objects to worker threads, we can find model-based approaches. In [142] the authors allow the calculation of an effective distribution of simulation objects among threads under the hypothesis that the future of the simulation run will have similarities with respect to the last observed execution phase. In particular, the authors allow dynamically redistributing simulation objects across threads, also considering, in the migration decision, information related to rollbacks, in order to minimize the costs of the speculative execution. For this scenario, the literature also offers approaches where the state of the simulation object is dynamically

migrated across the different NUMA nodes in the shared-memory machine [96] to allow a more effective management of the virtual pages used not only for the live state image of the simulation object, but also for the event buffers and for recoverability. The aim is to reduce memory latency of optimistic PDES due to the lack of awareness of NUMA architectures, devising a memory affinity mechanism based on Linux kernel facilities to allow reduced overhead for load balancing. However, none of the previous solutions has been tailored to short-term binding of simulation objects to worker threads, which is instead a core aspect we tackled in this thesis. This type of binding has been investigated in [54]. As we already mentioned, in [54], the authors exploit a fully-shared event pool in order to enable any worker thread to CPU-dispatch any simulation object at any time along the simulation execution. Hence, a simulation object is kept locked by a specific worker thread only for the time interval related to the processing of an individual event. In this work, the accesses to the shared event pool have been based on a non-blocking algorithm, which favours scalability [73]. The major limits of this work are related to the fact that spatial locality, or more in general memory hierarchy awareness, is not taken into account. Hence, a worker thread can continuously switch across different simulation objects, with no attempt to reuse the same memory areas. Also, the accesses are NUMA unaware, hence they can generate both delay, because of the latency for accessing remote NUMA nodes, and excessive pressure on the NUMA interconnection, limiting performance and scalability. As we pointed out, these are the baseline problems we tackle in part of the thesis, specifically in Chapter 4, in combination with the reduction of the amount of memory locations

accessed by a worker thread when managing the shared event pool.
The work in [148] provides an improvement for load-balancing in PDES on top of multi-processor/multi-core shared-memory machines. It combines a classical medium/long-term binding scheme based on persistence, namely past data related to the workload, and work-stealing. The stealing operation is put in place if the last rebalance has led to imperfect partitions. This might occur because of errors in the prediction of the future workload, that will be actually generated by the simulation objects when the rebalance occurs. In this solution, the simulation objects are still grouped and remain bound to a specific worker thread, up to some steal operation or a periodic rebalance. Hence, the core ideas of this approach are still not suited for locality improvement with fine-grain sharing of the workload among worker threads. Additionally, spatial locality is not taken into account by the authors, hence they do not explicitly address the improvement of cache and NUMA usage, which are instead core aspects in this thesis.
The work in [141], provides a solution for improving the efficiency of cache usage in speculative PDES systems. This solution is based on redirecting cache-adverse operations, like checkpointing, which leads to the invalidation of other information kept into the cache, to a specific cache partition. In fact, operations like state saving/restoring, access data on a periodic basis, and they might cause eviction of cached data related to the actual working set of the application logic, causing a performance drop. In practice, this solution offers the advantage of keeping some zone of the cache less affected by cache replacement caused by write intensive operations related to data that are likely not re-accessed for a while (like it occurs for a checkpoint), reducing the

costs of cache misses. The work in [14], provides an analysis showing how checkpointing and reverse computation have different impacts on speculative PDES performance on a shared-memory machine because of their different effects on the caches and on the TLB architecture, particularly analysing the cache coherency impact on this kind of architecture. These proposals are however not tailored to the optimization of cache management in the context of fine-grain sharing of simulation objects among worker threads, and are also not tailored to the improvement of the effectiveness of memory accesses in NUMA architectures when cache misses occur. Furthermore, they focus on cache-adverse operations such as the checkpointing, rather than more broad cache-awareness for what concerns forward event processing. These aspects are instead central for this thesis.

As for the enhancement of spatial and temporal locality, the work in [33] has introduced an event execution strategy called demand-driven PDES: it exploits the inherent locality patterns of event execution according to which only some parts of the model are actively sending/receiving messages at certain time periods. The authors, therefore, identify idle threads (with no events to process) and temporarily de-schedule them from the CPU until they become active (with events to process) again. However, this work refers to platforms based on long-term binding, so it is not suitable for platforms exhibiting fine-grain sharing of resources, since in the latter scheme worker threads are always busy executing events for any available simulation object. Furthermore, the authors explicitly de-schedule some threads, therefore directly operating on the lifecycle of threads to avoid the idleness. In the solution proposed in this thesis, the lifecycle of worker threads

is not touched, but it is instead adopted a multi-level data structures scheme to pick events to process to satisfy spatial locality at some cache level.

As a matter of fact, the approach presented in this thesis is related to classical mechanisms that have been used in operating systems in order to CPU-schedule the different threads. In more detail, the perfect load sharing approach, which has been used in Linux[1], enables a same thread to consume its residual ticks according to a batching scheme. This enables the thread to exploit the caching system in a more effective manner, compared to the scenario where multiple threads that still have ticks to spend on the CPU are dispatched in an alternate manner. The proposal in this thesis also exploits a kind of batch-processing for enabling a worker thread to process events of the simulation objects, reducing the alternance and favouring also temporal locality (namely, a touched memory area is likely to be touched again soon). However, since we are targeting optimistic PDES platforms, the solution presented in this thesis also takes into account aspects that are not considered at the level of the operating system technology, like the need for avoiding timestamp order violations as much as possible, in order to reduce the incidence of rollbacks.

As an additional point, this thesis also provides a multi-view mechanism for handling the globally shared event pool, which allows reducing the amount of memory areas touched by a thread while selecting the events to be processed. In fact, as previously stated, the global queue of events is a bottleneck due to the frequent scan operations needed to pick an event to process. We want to reduce the likelihood of having to

[1] Available at https://mirrors.edge.kernel.org/pub/linux/kernel/v2.4/

scan the global queue with our locality-aware scheme. Several works have been proposed which try to optimize the usage of event-pool data structures, either shared or not, in the field of PDES systems (see, e.g., [30, 107, 110, 111, 137]). Also, hardware transactional memory has been exploited in speculative PDES on multi-core machines, particularly for consistent management of the accesses to a shared event pool by the worker threads [49]. However, the multi-view approach has not been considered by previous studies. Furthermore, this thesis is still aligned with the trend related to non-blocking management of shared data structures on multi-core machines. This enables providing the advantages of increased locality of the operations by threads, avoiding at the same time the penalty due to thread blocking phases when managing the shared event pool.

Considering the general field of parallel computing, there are works that focus on spatial locality by making each thread work on a reduced size portion of data, sometimes automatically selected at the compilation level [50]. This work specifically targets programs written in Cilk. Other solutions, such as the ones tailored to task-level parallelism (i.e. OpenMP), separate tasks in hot (already partially processed by a thread) and cold (not yet processed) ones, and make threads process their hot tasks with higher priority when they become ready again for processing (see, e.g. [128]). One of the goals of the work in [128] is to effectively manage task priorities, in order to avoid having tasks being blocked by lower priority tasks, improving the overall utilization. Other works cope with the reduction of shared-data access conflicts, which can have negative effects on the

caching hierarchy [79], aiming to improve the efficiency of synchronization mechanisms on multi-processor/multi-core shared-memory architectures. Data-sharing access conflicts have also been considered in the case of applications relying on transactional memory [31]. In this work, the authors devised an approach to detect and solve transaction conflicts, aiming to minimize the likelihood of conflict via a scheduling mechanism, and providing a resolution scheme that prevents unnecessary re-execution of transactions. All the solutions described do not tackle the combination of spatial and virtual time locality, as instead it is considered in this thesis suited for speculative PDES. Furthermore, we highlight that in the context of optimistic PDES targeting fine-grained sharing of resources, the combination of spatial and temporal locality for processing events on NUMA architectures have not been tackled.

3.2 Checkpointing

As discussed in Chapter 2, speculative PDES some recovery mechanisms are needed in order to solve possible causality violations. This has been traditionally tackled through state saving techniques, or through reverse computation. Since the focus of part of this thesis is to propose solutions for the checkpoint operation, we will analyse what has been done over the years in this context.
Several works have investigated how to exploit infrequent (e.g. periodic) checkpointing techniques to optimize the tradeoff between the cost of checkpointing and the cost of restoring a state that was not

checkpointed [36, 90, 103, 104, 106, 115, 129]. Other authors have proposed the usage of acceleration based on hardware support to execute the memory copy operations required to pack the state of the simulation object in the checkpoint buffer [44, 109]. All these solutions target the case of non-incremental checkpointing; hence, they can be considered orthogonal to the solution we provide in this thesis.

Regarding works targeting incremental checkpointing, [149] and [98] propose techniques based on binary-level instrumentation, while the work in [116] proposes the usage of operator-overloading schemes in the context of object-oriented programming. The objective of these works is to support the fine-grain identification of write accesses to the state of the simulation object in order to save reduced zones of the state layout. These solutions can be extremely useful for scenarios of (very) reduced volumes of write accesses to the object state. In contrast, as part of this thesis, we investigate the opposite scenario of incremental checkpointing, also for the case of non-minimal size zones updated by the execution of the events. We note that the approach proposed in this thesis also scales well with a reduced size of the state updated, this is because the amount of state updated must be considered in the context of the checkpoint interval. This means that even though an event can update a small portion of the state, large portions of the state can be updated in a whole checkpoint interval made of multiple events.

While the instrumentation-based approach enables the identification

of dirtied memory areas at granularity lower than page size, its disadvantages come from the likelihood of multiple executions of the write-tracking probe for a same memory zone. More in detail, the trampolines and the code blocks for determining the dirtied memory areas are executed each time a memory-update instruction takes place, independently of whether the target memory has already been recorded as a dirty zone to be checkpointed. This leads to spending CPU-cycles for executing useless memory-write tracking activity, a problem that becomes even more relevant when the incremental checkpoint is taken infrequently, after a large number of processed simulation events.

Still for incremental checkpointing, the solution in [11] provides support for identifying all the memory updates through the reliance on Performance Monitoring Units (PMUs) offered by modern processors. Differently from this proposal, we can keep track of all the memory writes targeting a specific page, or a group of pages, via the interception of a single memory update, after which the write protection is eliminated, rather than the interception of all the occurring memory updates. Moreover, we can further scale down the costs of managing memory updates through an approach based on buddy pages.

Other works have studied how to compare or mix incremental and non-incremental checkpointing [91, 99, 117, 131]. This has been done via the reliance on performance models, that can indicate the convenience of one or another technique in a specific phase of the execution of the simulation. This thesis is still orthogonal to these techniques and could be exploited for building model variations to assess the best-suited technique (the one proposed in this thesis or a different one) to be used during any phase of the model execution.

Rollback and state reconstruction have also been supported using reverse computation techniques [15, 19], where backward execution steps are used to rebuild the past state image that is requested when a timestamp-order violation is detected. In the proposal in [15], the backward steps are executed via the implementation of reverse event handlers, while in the proposal in [19], they are executed via the runtime generation of machine instructions for the reversing of memory updates. These techniques have one major target: reducing the need for large memory (for saving state information) while still enabling the possibility of restoring a past state. However, as also discussed in [19], they need to be integrated with checkpointing to avoid problems such as the excessive number of backward steps for state reconstruction. Therefore, this thesis is still orthogonal and likely combinable with these solutions.

Regarding the management of incremental checkpointing in speculative simulation via the exploitation of operating system write-protection mechanisms, the work in [120] provides an approach for High-Level Architecture (HLA) federations. In this work, the authors present a solution where speculative execution is supported transparently for the case of federate simulators designed to exploit the HLA conservative (non-speculative) interface. Differently from what we propose in this thesis, this solution has been mainly oriented to the compile-time/runtime identification of the operating system pages that in the address space are part of the federate state (rather than the Run-Time-Infrastructure of HLA) and need to be write-protected for supporting incremental checkpointing. However, each page is treated independently of the others regarding write access interception and memory

copy for checkpointing. Differently, in this thesis, we address the costs related to the management of operating system services, in particular to the reduction of the number of `SIGSEGV` signals (caused by illegal accesses in write mode) to be processed, after calling the `mprotect()` system-call, thanks to the exploitation of a correlation model of memory updates based on buddy-pages.

The reliance on operating system services for page write protection has been largely exploited in the context of checkpointing (full or incremental) for fault tolerance (e.g. [3, 52, 63, 65, 101, 102]). However, several of the proposed solutions leave the operating system kernel with the task of opening the write permission to pages (e.g. after a copy-on-write event). At the same time, most of these solutions have the principal target of reducing the intrusiveness for materializing the checkpoints (namely, the page copies) on stable storage. In this thesis, we cope with a different scenario since the checkpoints supporting rollback for out-of-timestamp order event processing in speculative simulation do not need to be transferred to stable storage. In fact, the out-of-timestamp order processing of events does not lead to main memory loss, as instead it occurs in the case of faults. The work in [89] exploits operating systems' memory protection services to track memory writes, devising a framework that captures all dynamic memory allocations in order to manage the checkpointing request for fault-tolerance purposes in large-scale scientific applications through checkpoint-restart. Despite leveraging a similar technique as we do in part of this thesis, the scope of the work is quite different. In fact, while the work in [89] uses memory protection services to track written memory pages and to analyse the access patterns in order to support

asynchronous checkpointing across multiple nodes in a distributed system, in this thesis we use memory protection services to track written pages, and we also devise a buddy-pages scheme to correlate the written pages and avoid redundancies of the state saving facility. The goals of the works are therefore different, as we aim to reduce the costs associated to the checkpoint saving phase by grouping multiple pages through a correlation model. In fact, as part of the solution adopted for improving the incremental checkpointing, we exploit the memory access profile, and the correlation of memory writes on different buddy pages, to optimize the incremental checkpointing operations. This is fundamental in speculative simulation because of the intrinsic need for (frequent) state restoration, hence relatively frequent checkpoints, in the event of out-of-timestamp order speculative event processing.
Additionally, we not only leverage existing operating system services, but we also focus on providing innovative operating system services oriented to the lightweight and scalable tracking of page updates. In particular, while the literature approaches oriented to fault tolerance are based on determining if any thread of an application has accessed whichever page of the address space in write mode [89], and therefore they require mechanisms like IPI in order to support the cross-CPU coordination when protecting pages from updates by multiple threads, the solution proposed in this thesis is suited for tracking write-accesses by a specific thread to pages destined to host the state of a specific simulation object. Hence, we avoid at all the coordination of the different MMUs of the CPUs on-board of the machine when a checkpoint interval is started for a given simulation object. At the same time we support the cross-thread sharing of the accesses to the pages hosting

the state of a specific simulation object while still tracking memory-write accesses, which is relevant when the objects are re-assigned to threads (e.g., for load balancing/sharing) along their checkpoint intervals.
For scenarios where the frequency of checkpoints needs to be higher, such as, for example, the reliance on checkpointing for automatic error recovery in server-side applications, the work in [144] provides an approach in which the hot pages of an address space are checkpointed prior to their actual write access. The solution proposed in this thesis has some points in common with the solution in [144], since we also rely on anticipated page-level marking/checkpointing vs the actual write. However, we also provide support for optimizing the opening of memory write permissions based on the notion of buddy pages. This is done by relying on the correlation of the memory write accesses along sequences of simulation events (see Chapter 5).
As a further optimization of the incremental checkpointing in speculative PDES, in this thesis we provide for innovative Linux Kernel Modules techniques, to tackle the memory tracking at the level of the PTEs. The goal is to reduce the intrusiveness of the aforementioned `mprotect()` system-call in terms of cross-CPU coordination (see also Chapter 2).

3.3 State Trajectory Inspection

A crucial aspect of speculative PDES systems, discussed in Chapter 2, regards the computation of the Global Virtual Time (GVT), that has

been extensively studied over the years, (see [42, 45, 55, 100, 132]) in order to optimize its computation and adapt it to the emerging shared-memory platforms.

This topic, and the other ones covered by the literature regarding load-balancing and state management in speculative PDES (see Sections 3.1 and 3.2), are preparatory to one aspect that has been dealt with in a limited manner in the PDES community, that is the access to the committed portion of the execution trajectory, as also mentioned in Chapter 2, e.g., in order to determine specific properties that are matched (or not matched) by the model that is being simulated and/or to produce output data on-line.

In fact, the management of output data produced along the execution of simulations, as well as the dynamic evaluation of predicates and the representation of statistics, and their variations, related to the simulation trajectory represent increasingly relevant aspects. They are among the core aspects to be managed when exploiting simulation in complex scenarios like the one of providing a support for making decisions in a pandemic like COVID-19 [71]. In this area, some recent works have focused on exploiting mechanisms for an effective rendering and for the on-line analysis of simulation data [34, 67, 68, 119]. These solutions are suited for processing data after they have already been produced. Compared to these proposals, the presented approach in this thesis is orthogonal since we provide mechanisms operating at the level of the speculative PDES engine in order to support the on-line data production process with low overhead and high timeliness, in particular considering the need for accessing state information that belongs to the past committed portion of the speculative simulation run and is

no longer materialized into the states of the simulation objects.
The work in [152] provides a solution for the reduction of the storage and processing cost related to data management in agent-based simulations in the Cloud. This solution avoids storing of large amounts of raw output data by exploiting stream data processing, which is used to generate the result dataset along the simulation model execution. Like the aforementioned literature studies, this work is still focused on the management of data after their production, while part of this thesis is focused on effective mechanisms for the production of data, in particular when considering speculative simulation and considering the relation between current/committed state images and their memory layouts.
As for the management of simulation output streams, the work in [4] presents a solution for enabling a speculative simulation engine to support applications where the handler that executes events with no assurance of their causal consistency can also produce output data, for example by relying on the standard `printf()` function. In this solution, the output data are then annihilated or are actually reversed on the standard output channel for their commit, depending on whether the execution trajectory of the simulation object that produced them is finally committed, or is rolled back. This solution is essentially oriented to transparency, and has the major objective of enabling a simulation model developer to use standard libraries for output data production when writing the simulation software that will be processed optimistically. In this thesis, we tackle a different scenario, where the production of output data does not take place in the handler that processes the events. Output data are instead produced by a different

handler, which receives in input a past and committed state snapshot of a simulation event. In this scenario, there is no cost for producing output data that are eventually retracted, which makes it more suited for extremely reduced intrusiveness of the output activities with respect to actual simulation processing. We also tackle an effective way of identifying and access the committed global state to allow output collection, as we will discuss in Chapter 6.

The global state identification has been a relevant problem in distributed systems for decades, and in work [7] it is thoroughly described. In particular, while in [7] the authors focus on the identification of a committed global state in the context of a distributed system based on message-passing, we are in the context of speculative PDES, and the committed global state refers to a state which is considered safe (i.e. not subject to rollback). Similarly to what proposed in this thesis, [7] needs to rely on snapshotting the state of the application to maintain consistency across the nodes in the system, the proposed solution in this thesis leverages common checkpointing support to inspect the committed global state. The work in [7] gives some fundamental concepts on the relevance of accessing a committed global state, even though in the context of speculative PDES we have to consider several challenges related to the speculative nature of event processing and also related to the multi-core shared-memory machines we are targeting, that allow us to exploit a fine-grain sharing of resources.

In order to effectively access the committed global state for trajectory inspection, we must make sure that the state being accessed cannot suffer from causality violations, so its virtual time must be lower than the current GVT value. Since we also want to guarantee progress of the

simulation, freezing the simulation in order to wait for a simulation object's state to be outdated is not a feasible approach. In the literature, state-swapping has been studied in order to tackle this problem [21]. In this work, the authors provide a heuristic mechanism for identifying a committed consisted global state (i.e., with no events not recorded in the history which has generated some other event that is instead recorded in the same history), and provide an algorithm for making the worker threads reconstruct the local state of each object which belongs to the consistent global state. This algorithm is based on the above-mentioned state-swapping, which allows temporarily swapping the current state S'_x of a simulation object with a committed one S_x which is rebuilt in memory (possibly in the same memory area where S'_x stands) using common approaches for state recovery in speculative PDES. In this solution, S_x is typically the closer state that has been passed through by o_x before the new GVT value. This enables the alignment of the inspected state to the latest committed logical time of the simulation. However, in the scenario tackled by the literature, each simulation thread has its own set of managed objects (namely, the long-term binding based approach). This implies that when thread T_i needs to swap the state of the simulation object o_x that it manages, there will be no other thread capable of touching the state of o_x concurrently. This is because of the binding between threads and simulation objects, that in this case guarantees that no interference across threads will occur when swapping the states. Hence, it is not suitable for modern speculative PDES platforms to be run on top of multi-core shared-memory machines that support the full-sharing of the workload among all the worker threads [54, 56]. In this scenario,

the realignment to the committed state value of an object o_x needs to avoid interference with respect to the regular operations carried out on that object by some other worker thread. We note that similar considerations apply to simulation engines that can run on a cluster of shared-memory machines, where fine-grain workload sharing among worker threads can be adopted within each node in the cluster.
Furthermore, in [21] there is no specific mechanism for contextually moving all the threads to the management of the output data production phase. These are crucial aspects for reducing the latency of output production, which are instead the main targets in this thesis. As an additional core aspect, the literature does not consider that a thread which is faster than others in swapping its simulation objects states can then resume processing the events of these objects before other threads, possibly leading to over-optimism in their execution. We instead tackle this issue in this thesis by exploiting the IPI architeture to enable a prompt and simultaneous notification to all the threads that it is time to do the state-swapping operation.
The work in [2] addresses issues concerning global termination conditions in simulation applications. This is done via categorization of non-trivial termination predicates, and via the introduction of algorithms suited for detecting predicates in different categories. These algorithms implicitly assume the availability of complete state histories of the simulation objects for evaluating the termination conditions. The solution proposed in this thesis is instead focused on effective simulation-engine mechanisms for managing the phase related to the re-construction of some committed (global) state, for which the check of predicates can be carried out on-the-fly.

Chapter 4

Spatial/Temporal Locality-based Load-sharing

As mentioned in previous chapters, efficiently executing speculative PDES applications on multi-core shared-memory machines presents several challenges, for example related to workload distribution and memory efficiency. Addressing these challenges is critical for improving the performance and scalability of simulation systems, especially in the context of PDES platforms exhibiting fine-grain sharing of resources, such as the one described in [54]. In such systems, in fact, there is a general lack of memory awareness in terms of access patterns throughout the memory hierarchy, and this fails to exploit the inherent characteristics of shared-memory machines. This is why we developed an innovative event processing mechanism, based on a spatial/temporal locality-based load-sharing scheme to process simulation events, also leveraging a dynamic window-based approach to manage the batch processing of events. We also consider NUMA awareness, devising a

simulation object migration scheme in order to effectively balance the workload in unbalanced simulation models.

4.1 Baseline Architectural Concepts

4.1.1 Distance between Threads

We define a notion of distance between threads, that has a relevant role on how we tackle memory locality and memory hierarchy awareness. We denote with $D_{i,j}$ the distance between thread T_i and thread T_j, which is clearly symmetric among the two threads. Also, we denote as $MaxD$ the maximum distance we have among two threads in the PDES system, which corresponds to the distance between two threads running on two processing units that only share the RAM level of the memory hierarchy on UMA machines, or are pinned on distant NUMA nodes in a NUMA machine. On the basis of the above considerations, we also have that $0 \leq D_{i,j} \leq MaxD$ for any couple of worker threads T_i and T_j.

4.1.2 Simulation Object Memory Layout

We consider a general memory usage approach for simulation objects, where dynamic-memory allocation, e.g. the usage of malloc services, is adoptable while handling the state layout of an object. Also, any dynamic allocation request is served by a cached allocation sub-system that pre-allocates the memory segments destined to host chunks of a

given simulation object's state. Furthermore, the segment destined for the allocation of chunks belonging to the state of a given object is based on the mapping of specific virtual pages in the address space of the simulation application. Hence, different simulation objects will have their states hosted by disjoint sets of virtual pages. This enables locating an object state—its virtual pages—on a NUMA node X, or migrating it to a NUMA node Y with no impact on the physical memory positioning of other simulation objects—just thanks to the disjointness of the sets of pages hosting them. The NUMA-oriented version of Time Warp presented in [96], already adopts a similar scheme for managing the states of the objects, which we simply adhere to in this thesis.

4.2 Locality Aware Scheme

The above-defined notion of distance between worker threads takes into account spatial aspects related to the memory locations that are used while running the simulation. In particular, if one thread T_i has touched a given area of memory recently, then any other thread T_j whose distance from T_i is low will more likely find such memory area hosted in higher-level caching components. This can favour processing activities by T_j. At the same time, if the simulation object for which T_j has picked its next event to be processed is no longer kept into some caching component (e.g. because of cache replacement with other data), then having this object currently located on the NUMA node close to the processing unit where T_j is pinned can still favour the processing activities of T_j, and at the same time can indirectly favour

the activities by other threads because of the reduction of NUMA interconnection traffic.

Beyond that, one important aspect is how temporal locality affects speculative PDES simulations. In particular, we want to avoid that this kind of locality-aware processing makes logical clocks of simulation objects managed by different threads diverge. If clocks diverge, we might have synchronization issues due to the generation of rollbacks along the speculative execution, and consequently performance degradation [133].

In order to avoid this, we introduce a window W of simulation time to decide the actions of a thread. Even though the use of a time window has been considered in the literature [6, 28, 29, 90, 108], we consider the window W according to a new perspective. In fact, the role of our window based mechanism is to increase spatial locality, combined with the afore-mentioned distance $D_{i,j}$ between threads, and the NUMA placement of simulation objects. We do not exploit window for throttling purposes, namely for stopping events' injection into the system, rather, we exploit it for scheduling simulation objects with events' timestamps still falling in the window range with higher priority compared to others. In Figure 4.2.1 we show a scheme of our idea combining temporal locality along virtual time and spatial locality. In this scheme, thread T_i starts processing the WHITE events destined to some simulation objects, leaving the BLUE events not processed in the meanwhile, since they are destined to another simulation object, event though they have a lower timestamp. This gives rise to a batch processing of WHITE events, and it likely avoids that the state of the target simulation object is replaced in the higher level of the cache

hierarchy because of the switch to the processing of events, by thread T_i, from another simulation object. This favours the likelihood of serving memory accesses though the higher levels of the caching hierarchy along the batch of 3 WHITE events. Now, let's imagine that one WHITE event produces a YELLOW event, still falling in the window W. Also, T_i has previously scheduled a simulation object to process another YELLOW event, which is now committed since the GVT is ahead of its timestamp. It is likely that the state of the target simulation object is still present in some cache level. At the same time, another thread T_j becomes available, and since it shares some high cache level with the thread T_i, it can take advantages in terms of memory access latency by picking the YELLOW event that falls within the window, rather than picking the BLUE event. In other words, T_j can still benefit from spatial locality favouring the access to the state of an object that was previously fetched into a close cache component, before scheduling other simulation objects not recently accessed by threads at lower distance from Ti. The just described scenario focuses on the scheduling of simulation objects, in particular on the improvement in the exploitation of simulation objects' states that were recently fetched in close cache components. As mentioned, we also tackle the optimization of memory access due to cache misses, which is particularly relevant in NUMA platforms. The window is also exploited to decide whether a thread available for processing events has to schedule a simulation object located on the same NUMA node on which the thread is pinned. In Figure 4.2.2 we see this scenario: a thread T_j is available for event processing, but, differently from the case in Figure 4.2.1, cannot exploit a simulation object that has been previously cached in a close

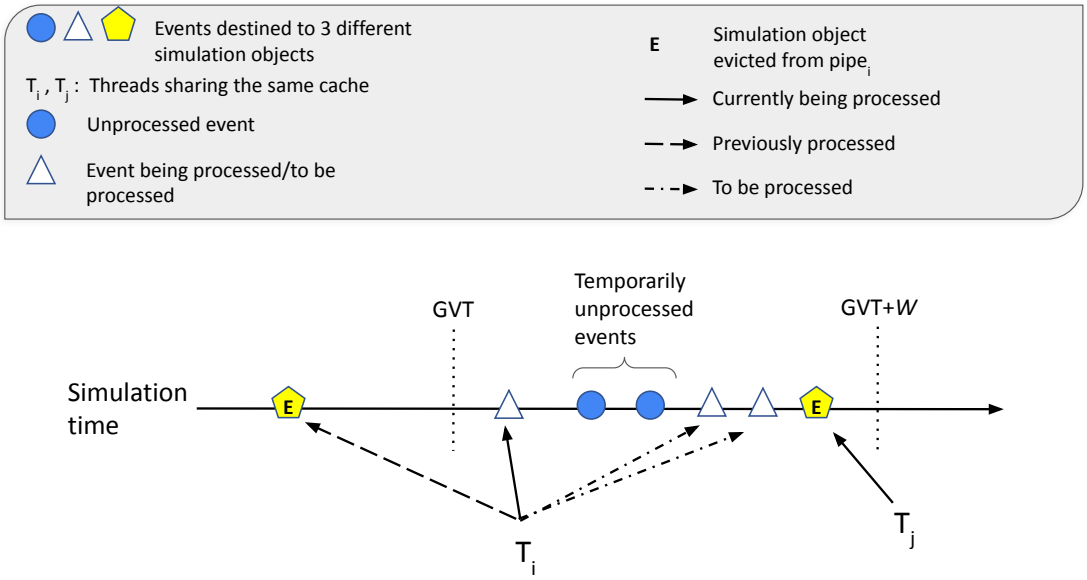

FIGURE 4.2.1: Joint exploitation of temporal and spatial locality at cache level.

caching component. Hence, it needs to schedule any other simulation object, and this would translate to traversing the global event pool. Instead, we see that T_j decides to schedule a simulation object to process a BLUE event falling into the window W, because the simulation object is placed on the same NUMA node the thread is running on, leaving the WHITE events unprocessed, since they are destined to a simulation object on a far NUMA node, despite having lower timestamp. The expected benefit is the reduction of the actual latency for event processing due to a reduction of the miss penalty. At the same time, leaving the WHITE events unprocessed can favour their picking from another thread T_i pinned on that NUMA node. As mentioned, we adopt a short-term binding of simulation objects to worker threads to carry out a sort of batch processing, but in order to enable the

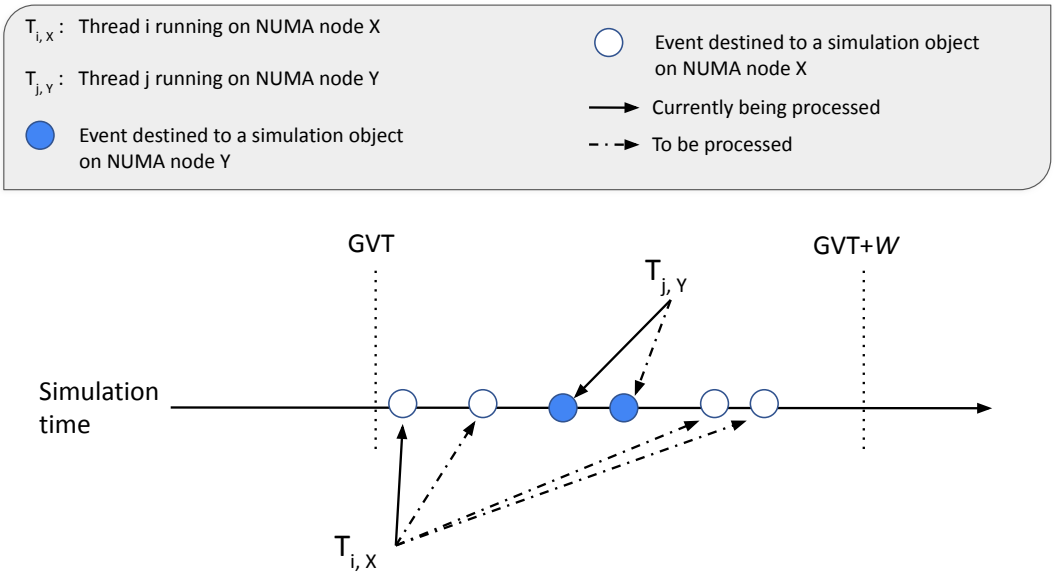

FIGURE 4.2.2: Joint exploitation of temporal and spatial locality at NUMA level.

exploitation of spatial locality and the improvement of caching effectiveness, worker threads do not just keep one simulation object bound to them. Rather, they keep a set of simulation objects, whose events have been processed by those threads recently. The objects' management is carried out via a pipe of simulation objects, denoted as $pipe_i$, associated with each worker thread T_i. Each time the worker thread picks an event destined to some simulation object from the shared event pool, that simulation object is put into the associated pipe as the standing element, independently of the object being already in the pipe. In this way, the order according to which the simulation object appears in the pipe, denotes the time distance according to which the worker thread has managed that object, leading to fetch its state from a higher level of the caching system. Since the pipe is represented with

an array in our implementation, and we will discuss how its size has been parametrized, it has a bounded size value, which means that in order to save a simulation object as a standing element, some other element might be discarded. This implies that the discarded object is no longer bound to the worker thread, and it might be managed by another worker thread. As we can see in 4.2.1, this is what happens with the object associated to the YELLOW events. We can further exploit the knowledge of the recently evicted simulation objects from the pipe, because it is possible that their state information might still be present in a lower cache level. To keep these simulation objects into account, to further improve the overall caching hierarchy access, each worker thread T_i also manages an additional pipe, called $evicted\_pipe_i$, in which recently evicted simulation objects bound to T_i are put. This works exactly like the pipe, but when an object is discarded from the evicted pipe, it is simply eliminated because of the lack of space.
Of course, improving caching management and exploiting spatial/temporal locality on these kinds of PDES platforms impose some constraints on the amount of batch processing actually beneficial. In fact, as discussed in previous sections, the GVT computation is a crucial problem in PDES systems, especially considering shared-memory architectures and fine-grained workload sharing. To briefly recall, the advancement of the GVT is determined by the change of the state in the shared event pool, and so represents overall simulation progress. Consequently, we have a frequent move forward of the GVT, and therefore a frequent move forward of the simulation time window [GVT, GVT + W]. So, the core idea of our scheme is the one of processing at thread T_i as many events as possible associated with the simulation objects in

$pipe_i$, as long as the timestamps fall within the interval [GVT, GVT + W]. Clearly, when the GVT stops advancing, since we temporarily did not process events destined to other simulation objects, the content of the pipe needs to be updated. So, each worker thread T_i can either pick a different simulation object from an evicted pipe associated to some other worker thread T_j close to itself, or consider simulation objects located on the same NUMA node it is pinned on.

4.3 Workload Management Scheme

In Algorithm 4.3.1, we show the simulation loop executed by each worker thread. At the beginning of each iteration, the thread attempts to pick an event from a simulation object that allows exploiting spatial and temporal locality. This is done by invoking LocalityAwareSchedule() to extract a simulation object in the pipe. If the function returns no event and no object, meaning that it is not possible to exploit the locality aware scheme, the thread simply picks the event with the smallest timestamp, following the classical rule, by invoking SmallestTimestampSchedule(). Anyway, if an event e is picked, associated with an object o, before processing the event a call to UpdatePipes() is made, which updates the pipes (*pipe* and possibly *evicted_pipe*) associated to the worker thread, listed in 4.3.2. Then, a check on the event e's timestamp must be done, because it is possible that the event e is in the past with respect to the current logical time of the object o. If this is the case, a rollback must be executed. We will not discuss here how the rollback procedure is carried

out, since it does not concern the spatial/temporal locality scheme, but it will be thoroughly discussed throughout the rest of the thesis. At the end of each iteration, a call to GVTOperations() is made, to manage the advancement of the simulation.

Algorithm 4.3.1 Simulation Main Loop

```
1:  procedure SimulationLoop
2:      while ¬ simulationComplete() do
3:          event e, object o ← LocalityAwareSchedule()
4:          if e = null then
5:              e, o ← SmallestTimestampSchedule()
6:          end if
7:          if e ≠ null then
8:              UpdatePipes(o)
9:              if timestamp(e) < o.LocalVirtualTime then
10:                 Rollback(e, o)
11:             end if
12:             event new_events ← Execute(e)
13:             InsertIntoPendingEventPool(new_events)
14:         end if
15:         GVTOperations()
16:     end while
17: end procedure
```

The Algorithm 4.3.2 shows the workload management using the load-sharing locality aware mechanism we developed. Every time a simulation object is chosen to process an event, it is temporarily locked and then unlocked when the processing is finished. The locking actual implementation is based on try-lock primitives, which does never lead a thread to actually block its computation if the object is not available, e.g. if it has been already locked by another thread. So, the function LockAndGetNextEvent() might return successfully an event to process, locking the simulation object, or "null"; the function

Algorithm 4.3.2 Locality-based Load-sharing Management Functions

```
1: function LocalityAwareSchedule( ) return (event, object)
2:     thread_id i ← GetCurrentThreadId()
3:     bool found_obj ← TRUE
4:     while W ≠ 0 ∧ found_obj do                ▷ This loop should be executed if and only if the window is set
5:         found_obj ← FALSE
6:         for k ← 0 to pipe_i.size do           ▷ Iterate over each object in the pipe of the current thread
7:             object o ← pipe_i[k]
8:             e ←GetNextEvent(o)                ▷ Obtain the next event of the object
9:             if e ≠ null ∧ timestamp(e) ≤ Limit() then   ▷ Check if the event timestamp belongs to the window
10:                return (e, o)                 ▷ An event with timestamp in the window has been found
11:            end if
12:        end for
13:        uint distance ← 0                     ▷ No object has an event with timestamp falling in the window
14:        while distance < MaxD ∧ ¬found_obj do ▷ Scan the evicted pipes of other threads according to their distance
15:            thread_id th_pool[] ← GetThreadsAtDistance(distance++)
16:            for h ← 0 to th_pool.size do      ▷ Multiple threads might have the same distance from thread i
17:                thread_id j ← th_pool[h]
18:                found_obj ← ChooseObjectFromOrderedSet(evicted_pipe_j) ▷ If returns TRUE, an object from
19:                                              the evicted pipe of j has been added to the pipe of i
20:                if found_obj then break       ▷ A new object has been found
21:                end if
22:            end for
23:        end while
24:        if ¬found_obj then                    ▷ No objects have been found, so look for in the local NUMA node
25:            object numa_objs[] ← GetAllObjectsFromLocalNumaNode()   ▷ Retrieve objects bound the
26:                                              local NUMA node
27:            found_obj ← ChooseObjectFromOrderedSet(numa_objs)   ▷ If returns TRUE, an object from the
28:                                              local numa node has been added to the pipe of i
29:        end if
30:    end while
31:    SquashPipe(i)  ▷ No object has been found in any pipe, so release any object in the pipe of the current thread and return
32:                                              to the main simulation loop
33:    return (null,null)
34: end function
35: function Limit( ) return simtime_t
36:    return GVT+W
37: end function
38: function ChooseObjectFromOrderedSet(object objs[]) return bool
39:    for k ← 0 to objs.size do                 ▷ Look for an object having the next event falling within the current window
40:        o_x ← objs[k]
41:        e ← LockAndGetNextEvent(o_x)
42:        if e ≠ null ∧ timestamp(e) ≤ Limit() then   ▷ An event has been found
43:            UpdatePipes(o_x)                  ▷ insert the object in the pipe of the current thread
44:            return TRUE
45:        end if
46:        Release(o_x)
47:    end for
48:    return false
49: end function
50: function UpdatePipes(object o) return void
51:    thread_id i ← GetCurrentThreadId()
52:    if o ≠ null then
53:        object o' ← pipe_i.insert_or_update(o)   ▷ insert the object in the pipe of the current thread, or update the pipe
54:                                              putting the object as the standing element
55:        UpdateEvictedPipe(i, o')
56:    end if
57: end function
58: function SquashPipe(thread_id i) return void
59:    while pipe_i.size > 0 do
60:        o ← pipe_i.removeAt(0)
61:        UpdateEvictedPipe(i, o)
62:    end while
63: end function
64: function UpdateEvictedPipe(thread_id i, object o) return void
65:    if o ≠ null then                          ▷ an object has been evicted from pipe_i
66:        Release(o)
67:        evicted_pipe_i.insert_or_update(o)
68:    end if
69: end function
```

RELEASE() is used to unlock the simulation object. The function LIMIT() always returns the value of GVT + W, and it determines the simulation time bound up to which an event can be processed according to the locality scheme (cache or NUMA based). It is important to note that the value returned by LIMIT() is volatile, because the real GVT value can be moved forward by some thread while another thread has executed LIMIT().

The function LOCALITYAWARESCHEDULE() is the core of the locality aware scheme. When a thread T_i calls it, it tries to pick an event kept in $pipe_i$ with the timestamp lower than the value returned by LIMIT(). If this is not true, T_i tries to pick an event from $evicted\_pipe_i$ or from an evicted_pipe of some other worker thread, still considering LIMIT() as the upper bound. This is done considering closer threads in terms of the previously defined notion of distance. Otherwise, thread T_i tries to get a simulation object located on a close NUMA node. When it picks a simulation object, it puts it into $pipe_i$ and starts processing the event.

The attempt of picking events from other sets different from $pipe_i$, is made calling the function CHOOSEOBJECTFROMORDEREDSET(), which tries to pick one of the objects from the set received as an input parameter. This set is an array, and it can be the evicted_pipe of another thread (close in distance to the calling thread), or an array of objects located on the same NUMA node on which the calling thread is pinned. Clearly, all the simulation objects standing into $pipe_i$ have been previously locked by T_i.

If none of the previous attempts succeed, then T_i simply gets a simulation object from the shared event pool with the classical lowest

timestamp policy. We highlight that, in this case, it might happen that the picked event e has its timestamp larger than the value computed by LIMIT(), and this is completely fair since we are not actually exploiting locality. As a final note, the shared event pool is also accessed to pick events if the window size W is not greater than zero, and we will discuss the importance of the window size tuning soon, since it is relevant for the exploitation of locality in combination with simulation progress guarantees.

In the implementation, the acquisition of a simulation object present in some evicted_pipe is managed through a while loop, even though it ends as soon as we successfully get a non-null simulation object whose event falls within the time window. Such an iteration might need to pass through all the eviction pipes of all the other threads, potentially requiring numerous iterations. To reduce these costs, the iteration can be carried out by not scaling the considered distance up to $maxD$. This will favour the picking of simulation objects relatively closer to the calling thread. As an example, we might run the iteration by searching only on the eviction pipes of the threads that share L1/L2 caches, limiting the number of iterations to, e.g., the number of hyper-threads or multi-cores present in a single processor.

Another aspect to consider is that the eviction pipes are accessed by both the owner thread, in write and read mode, and the other threads, in read mode. However, no actual synchronization, in terms of critical sections, is requested. In particular, since simulation objects' identifier are represented as basic data types, namely unsigned integer, can be read/written atomically in a circularly managed array by relying on

common processor support. At the same time, the reading of a simulation object identifier that is currently being evicted by the owner of the eviction pipe will simply lead the reader to consider as pickable for processing a simulation object that is less favourable in terms of spatial locality, respect to the new one that is being inserted into the eviction pipe.
Additionally, simulation objects cannot suffer from starvation due to unlimited delay for processing their events. In fact, if an object o is not dispatched on any worker thread for some time, one of its events will become the minimum-timestamp event in the shared event pool. In this case, the GVT will not advance until this event is processed, and all unprocessed events targeting other simulation objects will eventually fall beyond GVT+W, forcing the fetch of events from the shared event pool.

4.4 Multi-view Event Pool Management

We mentioned that our target PDES platform presents a fully-shared simulation workload among worker threads, and this makes the shared event pool crucial. It requires synchronization mechanisms in order to provide scalability for concurrent accesses. This has been typically tackled by relying on non-blocking algorithms, in particular lock-free [46, 66, 73].
As we enhance locality with our load-shared scheme, further improvements are needed regarding the data structures used by the threads in order to reuse per-object metadata, especially in the situation of

batch processing of events from the same simulation object. The goal is to reduce the impact of accessing the shared event pool, which is still present and might limit the performance reached by such a load-sharing scheme. Typically, lock-free event pools are designed as linked data structures, that are known to be ineffective in terms of cache utilization, and this worsens with its size and with a higher degree of concurrency. Consequently, traversing a fully-shared large-size event pool might reduce the effectiveness of any approach that tries to enforce locality when scheduling the simulation objects. Furthermore, lock-free algorithms make use of Read-Modify-Write (RMW) instructions, that can have a significant negative impact on caches. For these reasons, our approach exploits a multi-view shared event pool, such that a thread can choose a properly selected view of the event pool that suites to its current activities. Each of these views give rise to different memory access patterns, leading to different impacts on the memory hierarchy.

In one of the views, which we refer to as *local*, a worker thread can retrieve the next event to be processed for a specific simulation object bound to it, namely one of the objects in the pipe, without traversing the shared event pool. The worker thread needs to fully traverse the shared event pool only when it needs to look for an event not destined to any of the simulation objects bound to it, and we refer to this view as *global*.

We then exploited a multi-layer solution to build the multi-view event pool, in order to preserve the lock-free management. At the bottom layer, we use the lock-free shared event pool used by the USE platform [54]. This solution has been shown to be effective and scalable when

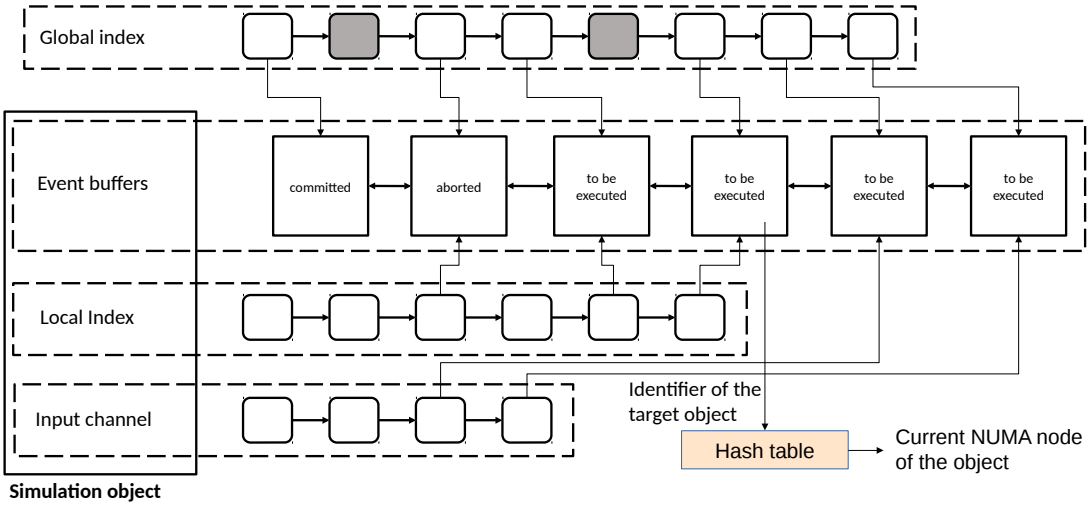

FIGURE 4.4.1: Visual representation of the multi-view shared event pool.

a thread needs to traverse the event pool according to the global view (see Chapter 2). At the same time, an additional layer has been devised to provide the support for the local per-simulation-object view of the events in the pool. The scheme is shown in 4.4.1. The shared event pool can be accessed by the means of either the *global index*, from which each event in the pool targeting any simulation object can be retrieved, or by *N local indexes*, where N is the number of simulation objects in the system, which allow a worker thread to only traverse events targeting one specific simulation object. Since any simulation object is processed by one worker thread, specifically the one that took the lock on the object, its events and metadata, including the local index, are respectively processed and updated by one thread at a time. This means that the local index can be implemented according to any sequential specification, allowing us to have a very simple and effective way to retrieve the next event to executed destined to a simulation object. One important thing to notice is that both the global and local

indexes maintain references to the actual events in the shared event pool, so they are not required to be coherent to each other, minimizing the need for synchronization when updating them. Additionally, no particular order is required for updating on or the other index, unless there is some specific need by the simulation platform. For instance, we choose to first update the global index because we exploit it to assign a tiebreaker to simultaneous events with a FIFO policy.

In order to allow worker threads to target any object when producing new events for it, even though the object is not currently bound to the thread producing the events, the newly generated events are not directly inserted into the target local index. Rather, they are inserted into an *input channel* associated with the target simulation object, and it is then lazily processed by a thread when updating its pipe. More specifically, newly generated events are first inserted in the shared event pool by updating the global index, and then in the input channel of the target simulation objects. Whenever a simulation object is managed by a worker thread, its input channel is flushed and its content (the events) is used to update the local index. In our implementation, the input channel is a simple non-blocking stack, which has the nice property of allowing $O(1)$ insertions and $O(1)$ bulk removals with an individual RMW instruction.

The event pool is then linked to a hash-map that enables to retrieve the current NUMA node hosting a given simulation object. In particular, the identifier of a simulation object, which is kept by any event buffer into the pool, can be used as an access key to the map. The global index in combination with the hash-map is exploited to pick an event, whose timestamp falls into the window w, to process targeting

a simulation object whose state is located on a close NUMA node. In our implementation, the hash-map exploits the fact that the access keys, namely the simulation object identifiers, are in a known interval of values ranging from 0 to N-1, with N being the number of simulation objects in the system, hence completely avoiding collisions and guaranteeing $O(1)$ access cost. Also, since the probability of finding an object located on the close NUMA node increases while traversing events on the global view, simply because the number of explored objects does not decrease, our solution is expected to guarantee a reduced complexity while managing the NUMA-aware picking scheme.

4.5 Dynamic Window Management

As pointed out, the window W is used to identify, starting from the GVT, a simulation time interval in which batch processing is possible. It helps to determine the batch of events destined to a certain simulation object, which can be processed out of order with respect to the timestamps of events destined to other objects. Also, W has a role in determining which simulation object is more favourable to be dispatched in terms of spatial locality when a worker thread needs to establish a new short-term binding, and it should also help to identify what events can be currently be executed without increasing too much the probability of causality violations – and consequently of rollbacks. The core issue for choosing the size of W is related to the fact that running with a size larger than zero gives rise to runtime dynamics that might be different from those related to the size zero. However,

if we run with a value associated to W larger than zero along any execution phase, we are not able to determine whether changes in the runtime dynamics, e.g. rollback occurrence, are exclusively caused by the value of W or are also due to specific runtime dynamics, such as event exchanges or variations in event granularity, at the level of the simulation model. As a consequence, the configuration with $W = 0$ represents a reference to take into account along the various phases of the simulation model execution.

To cope with this aspect, we manage the window W through a state machine, where the value zero is infrequently assigned to it, based on specific conditions met during simulation execution. Also, in the interval between two assignments to zero, the state machine tries to enlarge the window's size as much as possible, until no negative effects are generated on performance due to the increase of the amount of rollbacks. The enlarged value is kept until the next reset of the window size is carried out. In other words, while running in an interval between two resets, we focus on the effects of the window size on the rollback incidence. However, we end the current interval as soon as we recognize a change of the model execution dynamics, standing aside pure synchronization. In the above-mentioned state machine, two parameters are taken into account: the first one is related to the determination of changes in the rollback occurrence, either caused by the window size or by specific variation of the executed model, while the second one is related to the step of increment of the window W adopted.

A consideration must be done: this locality-aware approach for sharing the workload presents performance improvements even in the case of a slight increase of rollback occurrence caused by the non-negative

window size, thanks to the reduction of the average memory access latency experienced along the execution. The rollback pattern is complex, since it is based on both the frequency of rollbacks occurrence, and the length of the rollback phase, namely the amount of events undone. Depending on how the rollback support is implemented, e.g. checkpointing or reverse computation [15, 19, 103, 104], and on model specific features, such as the granularity of the simulation events, these two aspects can have a complex impact on performance. In order to capture the joint effect of rollback frequency and rollback length on performance, in combination with the actual locality-awareness approach, we based the state machine for selecting the window size W on the speed of commitment of the simulation. Hence, we decided to consider the event rate (the number of events committed per each wall-clock-time unit) as the parameter that enables determining whether the increase of W, after a reset, is generating negative effects. However, the event rate does not allow us to identify the effects on performance caused by changes in the size of W in scenarios where performance is also affected by some change in the execution profile of the simulation model, so we also consider the average granularity of simulation events. If the granularity considerably changes, we are in a scenario where a new setup of W may need to take place.

Denoting with er_x the event rate at its x-th observation, and with δ_y the average event granularity at its y-th observation, the state machine resets W each time the following condition holds:

$$\left(\frac{er_x}{er_{ref}} < 0.9\right) \quad OR \quad \left(0.6 > \frac{\delta_y}{\delta_{ref}}\right) \quad OR \quad \left(1.4 < \frac{\delta_y}{\delta_{ref}}\right) \tag{4.1}$$

where er_{ref} and δ_{ref} are respectively the reference event rate and the reference event granularity, both measured in the observation interval the follows the last reset of the window.
By Equation 4.1, we decided to reset the window size more aggressively when the event throughput (i.e. event rate) decreases vs er_{ref}, with respect to changes in the event granularity. In fact, for the event granularity we tolerate changes, positive or negative, up to 40%, because our approach is expected to provide changes in the event granularity while increasing W thanks to the locality-aware approach and the improvement of caching/NUMA effectiveness.
One final aspect regards the stabilization of the window size value W before any reset is applied. We simply use a hill climbing algorithm vs the event rate, stopping the climbing scheme as soon as we observe a maximum, as shown by the state machine scheme n figure 4.5.1. The step to increase W is based on a parameter Δ which we set to the average increment of the local virtual time of the simulation objects related to the processing of any new event. This helps to increase the size of W by having the opportunity to increase the size of the batch of events destined to an individual simulation object along any step of the climbing procedure.

4.6 Dynamic NUMA Placement of Simulation Objects

The previously described locality-aware mechanism already benefits performance, but NUMA locality can still be improved if threads are

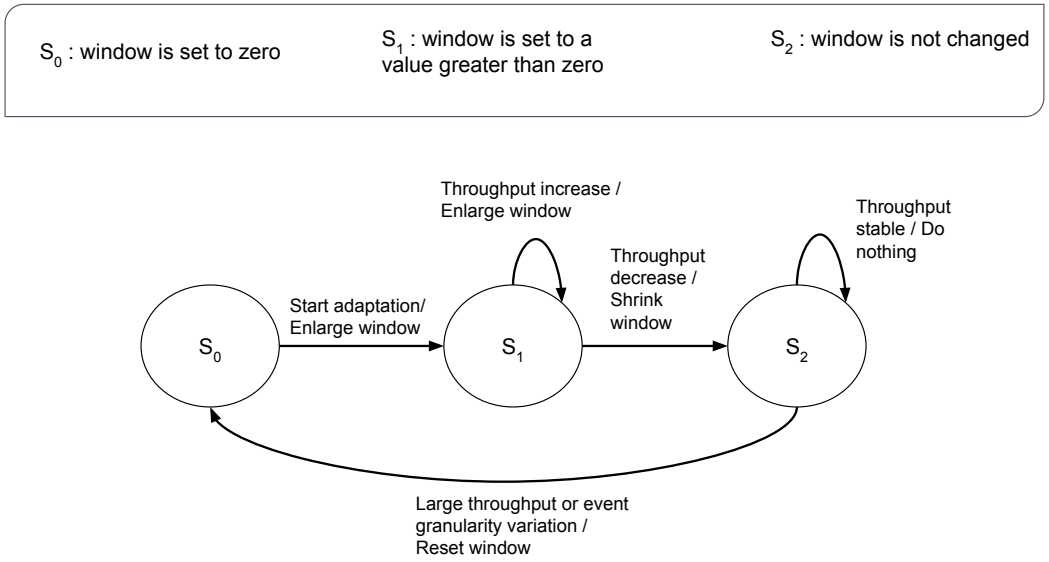

FIGURE 4.5.1: Scheme of the state machine for managing W (throughput denotes the event rate).

not penalized in the possibility of choosing a simulation object residing in its close NUMA node, rather than one hosted by a remote node). To reach this objective, our solution also embeds a scheme that enables a balanced distribution of event processing activities among the different NUMA nodes.

We indicate with ϕ_x the average inter-arrival time of events to the simulation object o_x along virtual time, and with CPU_x the average CPU time for processing an event destined to o_x. Based on these two parameters, we can introduce a weight w_x for simulation object o_x, computed as:

$$w_x = \frac{CPU_x}{\phi_x} \tag{4.2}$$

This weight represents the per-simulation-time-unit incidence on CPU caused by the processing of events destined to the object o_x. The goal

is to keep the sum of the weights of all the simulation objects hosted by a node balanced among all the NUMA nodes.
In particular, indicating with w_{NODE_y} the weight related to the simulation objects hosted by NUMA node Y, which we can compute as:

$$w_{NODE_y} = \sum_{\forall o_x \ on \ NUMA \ node \ Y} w_x \tag{4.3}$$

we decided to periodically perform a migration of simulation objects among NUMA nodes in order to get the distance between the maximum and the minimum value of w_{NODE_y} values lower than a threshold. According to the greedy approximation policy [23], the threshold distance we consider is 30%. However, we did not choose to perform aggressive migrations of simulation objects to match the threshold, since moving virtual pages across NUMA nodes via operating systems services has a cost. In fact, the CPU would synchronously work on data move tasks, like physical page allocations on the target node and data copies from the source node, as soon the service of the operating system is called, namely `move_pages()`. Rather, we decided to achieve a well-balanced configuration over time according to a lazy approach, by just moving, at each re-balance point, a small fraction of the simulation objects included in the simulation model, set to 5%, although we enable the migration of at least one object, which still enables the re-balancing for smaller models. The selected simulation objects to be moved are the one with the maximum weight (low ϕ_x and/or high CPU_x), which enables a bigger step towards the object of re-balancing the weights among nodes in the NUMA architecture.

4.7 Experimental Evaluation

4.7.1 Test-bed Environment

As stated in Chapter 1, all the solutions presented in the thesis have been integrated and tested into the USE simulation platform [54], an open-source high performance PDES engine for multi-core shared-memory machines, which provides non-blocking progress in both virtual and wall-clock-time. The former is achieved by having an innovative implementation of the previously described Time Warp protocol [57], revisited for the case of shared-memory execution. The latter is guaranteed by exploiting fine-grain thread synchronization consisting of individual atomic instructions, ensuring scalability while accessing shared data and metadata within the simulation engine.
USE is the reference platform for fine-grain workload sharing, and it has shown several improvements compared to platforms based on the classical medium/long-term binding of simulation objects to worker threads. In particular, USE has shown to reduce the incidence of rollbacks and to deliver higher performance compared to other PDES reference platforms run on top of shared-memory machines. For this reason, in this section, we will use the original version of USE as the baseline for assessing how the locality-aware scheme can further improve performance. In the original version of USE, threads are pinned to processing units and the placement of memory frames for the state of the simulation objects is determined by the default Linux kernel's policy *first touch*, according to which a frame is allocated within the NUMA node of the processing unit running the thread that touches a

TABLE 4.7.1: Hardware Platforms

Processors	2 x Intel Xeon Silver 4210R
Cores (Logical)	20 (40)
NUMA Nodes	2
RAM	160GB
L1	640KB 8-way
L2	20MB 16-way
LLC	27.5MB 11-way
Operating System	Ubuntu 22.04.02
Linux Kernel	v5.15.0

memory page for the first time. We included a layer that keeps track of memory pages pre-reserved for hosting the state of a specific simulation object also including an initial binding to a NUMA node, and we use the `move_pages()` service if the migration to another node is requested.

We carried out the experiments on the platform whose details are listed in Table 4.7.1, providing an adequate level of parallelism of 40 hyper-threads. The cache layout is retrieved automatically at compile time by accessing the sysfs pseudo-filesystem at `/sys/devices/system/cpu/cpu*/cache`. Finally, the platform is equipped with 160GB of RAM arranged in 2 NUMA nodes. The NUMA configuration is still retrieved automatically at runtime via the NUMA library.

4.8 Benchmark Applications

To evaluate the effectiveness of the proposed solution, we have performed an experimental evaluation that relies on three different simulation models, namely PHOLD [40], Personal Communication Systems

(PCS) [20, 58] and Tuberculosis (TBC) [81]. These applications have already been discussed in Chapter 2, but to briefly recall: the first one, which has almost no memory access to data when events are processed, has been used to estimate the potential overhead of our approach under different configurations in terms of event granularity. Conversely, PCS and TBC allow us to show the benefits given by enabling the locality-aware mechanism for caches and NUMA nodes.

As for the PHOLD model, we set the Fan-Out parameter to 1, and this leads to scenarios where the average number of events in the event pool is stable, but there are punctual fluctuations. The timestamp increments are drawn from an exponential distribution with mean set to one simulation-time unit. Finally, the busy loop generates a different event granularity at different tests, namely $5\mu s$, $10\mu s$, $25\mu s$, $50\mu s$, $100\mu s$, $200\mu s$ and $400\mu s$. The CPU busy loop is run with no need for memory access to data and data cache and/or close NUMA node exploitation.

Regarding the PCS model, we set $\tau_D = 120s$ and $\tau_H = 300s$, respectively representing the expected duration call and the residual residence time of the device inside a cell. The number of channels N and the inter-arrival time τ_A have been varied in order to evaluate scenarios where the utilization factor has different values, and is balanced or unbalanced across the cells. The channel utilization has as impact on both CPU and memory demand for executing the event handler. Also, the unbalanced scenario requires simulation objects migrations in order to balance the model execution activities across the different NUMA nodes, according to our approach.

Finally, we configured the TBC model to conduct a simulation of a

medium-sized city. Specifically, we used 1024 simulation objects, each covering a square region having a mean of 500 agents as population, for a total population of half a million people. 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% is sick or untreated. Despite the significantly small amount of infected people, the frequent agents' movements leads the simulating environment to a pandemic, with clear recurring phases of peak infection and low disease spread. We simulated a total of 6000 days of the evolution of the TBC model.

4.8.1 Preliminary Experimental Evaluation for Parameters Setup

Throughout these sections, we did not assume knowledge of certain crucial parameters for the locality-aware approach, such as the pipe size and the window size. We investigated several configurations for the size of both the pipe and the evicted pipe, and the actual strategy to set the window size W. For this purpose, we exploited the PCS model as the reference, and we ran it on top of our locality-aware solution varying the pipe size from 1 to 8 (so, in order to keep the bound for 1-8 simulation objects for each thread) and with multiple static window values W.

The results show that the optimal size for both pipes is independent of the window value W, and of the utilization factor of the cells within the simulation model. For this reason, we report a subset of the obtained results in Figure 4.8.1, namely for $W = 3.2s$ and $\rho = 0.5$. In

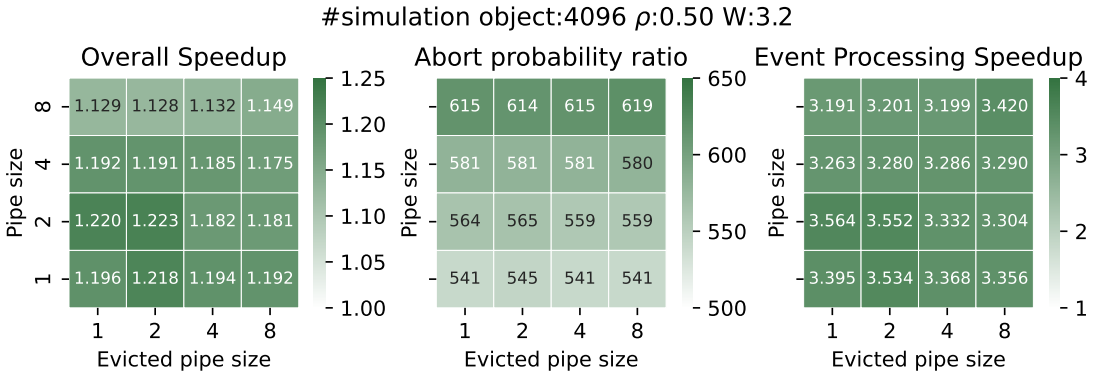

FIGURE 4.8.1: Evaluation of different pipe-size values for the PCS model run with 40 threads.

particular, we show three heatmaps where each tile represents a specific combination of pipe and evicted pipe sizes. The chart reports three different metrics from left to right: the leftmost one reports the overall speed-up provided by our solution with respect to the baseline load-sharing mechanism offered by the original version of USE; the heatmap at the center shows the abort probability ratio, namely the ratio between the abort probability measured with our solution and the baseline platform – the abort probability has been computed as the ratio between the rollback occurrences and the total number of executed events; finally the rightmost one reports the speed-up obtained just for event processing. The speed-up of event processing shows that the locality-aware scheme has a positive and relevant impact on event processing, which is at least 3.19x faster, with all the tested pipe sizes, having the maximum improvement for size equal to 2.

The abort probability ratio, on the other hand, is affected by the pipe size. In particular, the larger the pipe size (vertical axis), the higher the

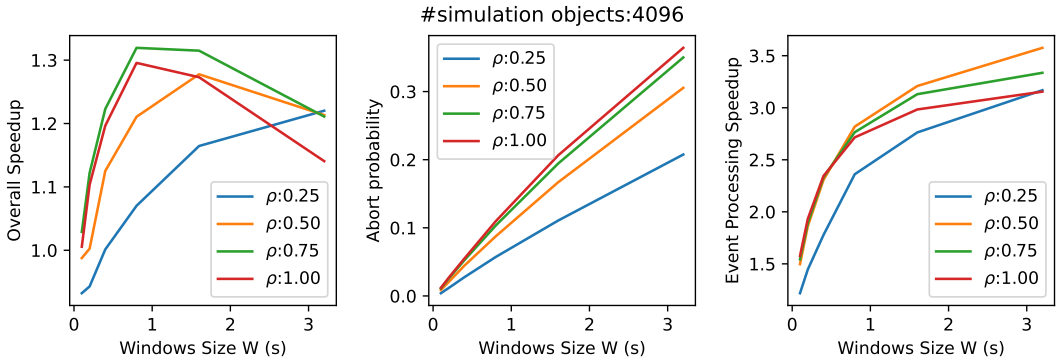

FIGURE 4.8.2: Evaluation of different window-size values for the PCS model run with 40 threads.

abort probability ratio. This reflects the fact that excessively increasing the number of simulation objects bound to a given worker threads leads to a scenario where the computing power is less likely allocated for higher priority events. However, the evicted pipe size has a reduced effect on the abort probability ratio, showing that its choice is not critical. Clearly, the higher the abort probability ratio, the less effective our solution, and this is evident by observing the overall speed-up (the leftmost heatmap), that is maximized when the pipe size is equal to 2. Consequently, we kept both the pipe sizes equal to 2 in our subsequent evaluation. In Figure 4.8.2 the results obtained with a static window W are shown. In particular, the leftmost chart shows the speed-up while varying W from 0.1 to 3.2: as expected, there is an optimal value for a given workload, which leads to balanced trade-off between the gains obtained in terms of locality while executing simulation events (the rightmost plot) and the abort probability (the centremost plot). This observation justifies the viability of the hill-climbing algorithm we use to manage the window size W during simulation execution, starting from $W = 0$. When the system is stable, namely the queue size and

throughput, the hill-climbing starts by exploring the search space, and it stops when a local maximum has been found. According to the empirical evidence, shown in Figure 4.8.2, this approach can identify the optimum.

4.9 Results

4.9.1 Results with PHOLD

In PHOLD, as described in Chapter 2, the event handler spins for a given amount of time, thus wasting mostly clock cycles. Consequently, no benefits in terms of memory locality are expected. However, this allows us to evaluate the impact of the proposed locality-aware scheme, in particular the slowdown for managing additional data structures, such as the pipes, the local index and the NUMA-based hash-map for determining which simulation object is more favourable to pick.

Figure 4.9.1 resumes the results in terms of overall speed-up. In particular, it reports the speed-up achieved by the baseline USE and the proposed approach, denoted as *cache+NUMA opt.*, with respect to the sequential execution. The data show that scalability is not hindered and not affected by the additional work needed for managing the locality-aware scheme. In particular, there is no significant difference between the performance of the baseline USE and the new solution over a broad range of thread counts and event granularities. We also report, in Figure 4.9.2 the absolute throughput values for the different setups we used for event granularity and thread count.

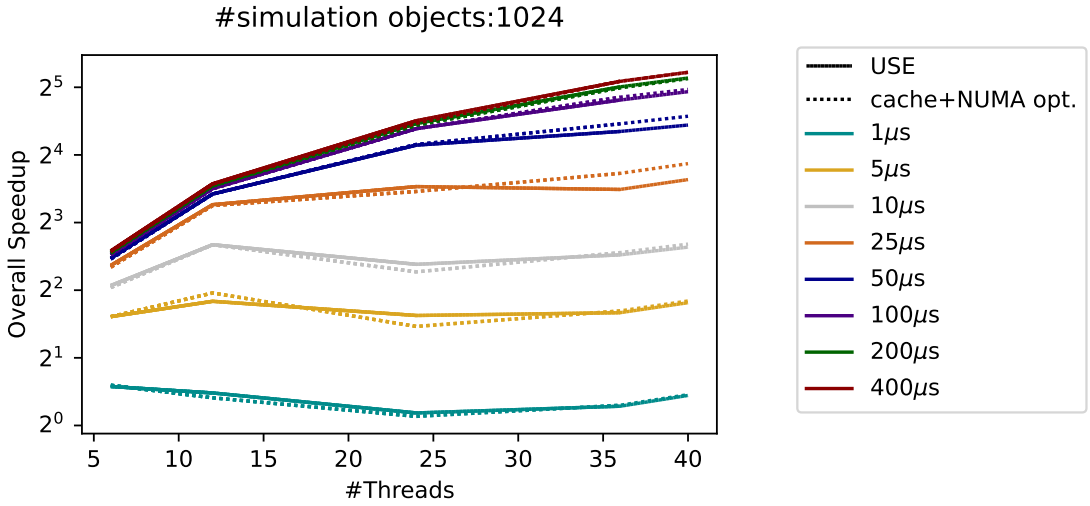

FIGURE 4.9.1: Speedup with respect to PHOLD sequential execution with different event granularities.

4.9.2 Results with PCS

For what concerns the PCS model, we have tested different model sizes using 256, 1024 and 4096 simulation objects, namely cells, each one managing 1000 channels, and also with different utilization factors (0.25, 0.5, 1.0). This allowed us to explore the benefits of our approach for different levels of disjoint access parallelism characterizing the simulation model execution. We evaluated the performance comparing our approaches, the locality-aware scheme (denoted as *cache+NUMA opt.* and the cache locality-aware scheme without NUMA awareness (denoted as *cache opt.*, compared to the baseline load-sharing scheme implemented in USE. Figure 4.9.6 reports the performance achieved when running the PCS model at full concurrency, which is 40 threads on the target machine. We report the average value of the committed events' throughput observed at steady state calculated over 12 runs

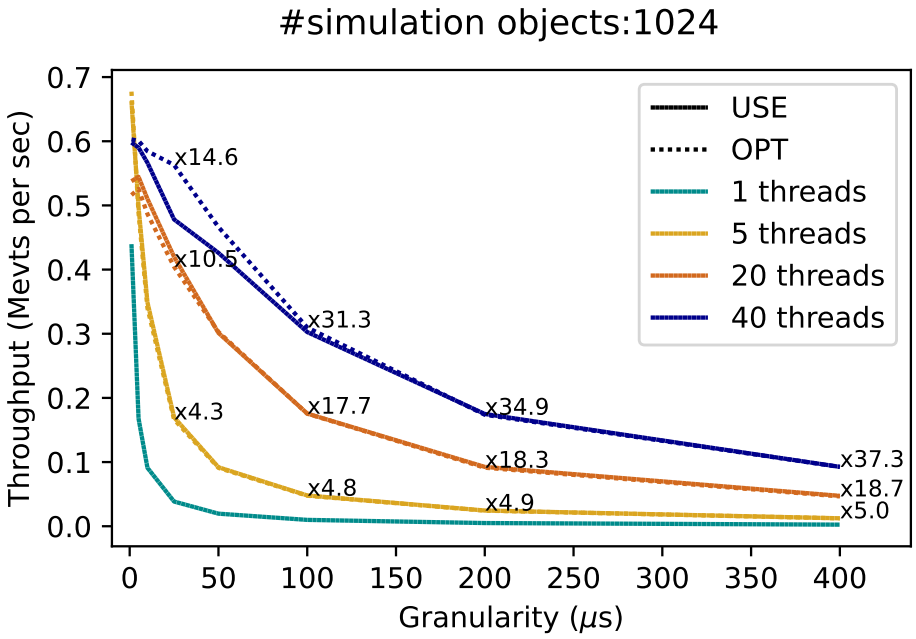

FIGURE 4.9.2: Throughput of PHOLD with different event granularities and thread counts. Each label represents the speedup relative to the sequential execution for a given event granularity and thread count.

(represented with a triangle), its median (represented as an orange line), its 80% confidence interval (represented as a box) and its minimum/maximum values (represented as error bars).

We observe that for smaller simulation model sizes our approach turns out to be less effective in the attempts of exploiting spatial locality when increasing the size of the window W. We recall that this simulation model has no lookahead, thus not favouring our window-based approach. This is evident looking at the leftmost plot, showing the results for the model size equal to 256 simulation objects, where our solution gives rise to an execution speed that is essentially the same, or slightly better, than the one offered by the baseline.

Conversely, increasing the model size allows a higher effectiveness of

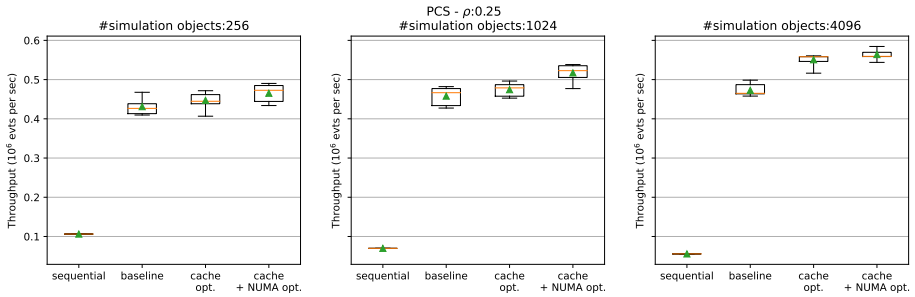

FIGURE 4.9.3: Execution speed with $\rho = 0.25$.

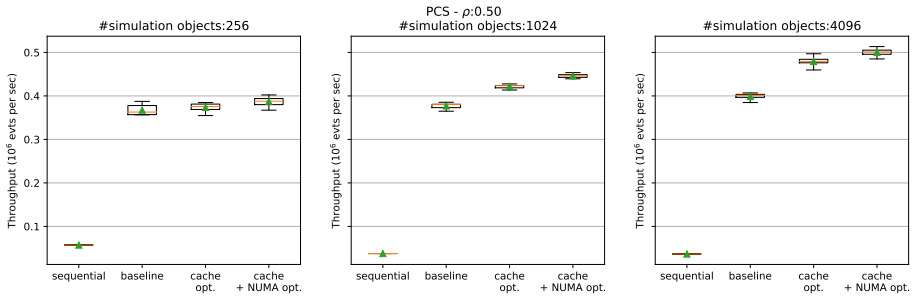

FIGURE 4.9.4: Execution speed with $\rho = 0.5$.

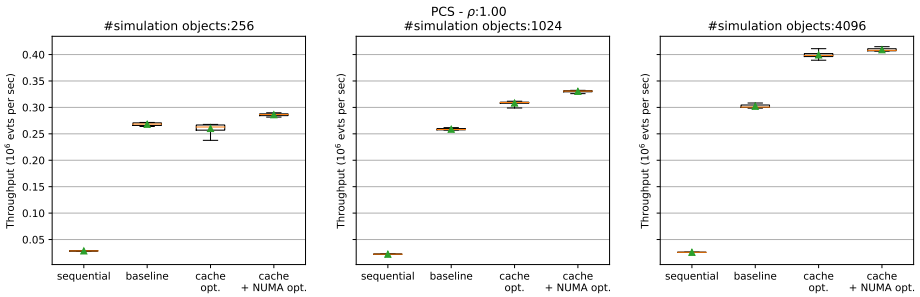

FIGURE 4.9.5: Execution speed with $\rho = 1$.

FIGURE 4.9.6: Results with PCS.

our solution, providing an improvement of the execution speed-up up to 30%. This maximum gain is achieved when the model size is set to 4096 simulation objects and the utilization factor is equal to 1. This is also in support of the fact that the proposed solution is well-suited for heavier-loaded models, such that the density of simulation events per-object along simulation time is higher. Also, the plots show how

TABLE 4.9.1: Average rollback frequency of PCS

UTILIZATION FACTOR	$\rho = 0.25$			$\rho = 0.50$			$\rho = 1.00$		
#SIMULATION OBJECTS	256	1024	4096	256	1024	4096	256	1024	4096
baseline	0.71%	0.19%	0.05%	0.77%	0.22%	0.05%	0.91%	0.24%	0.05%
cache opt.	2.12%	2.68%	3.21%	2.80%	3.29%	2.94%	2.81%	3.24%	2.55%
cache+NUMA opt.	2.62%	3.14%	3.16%	2.75%	3.49%	3.03%	2.95%	3.28%	2.64%

TABLE 4.9.2: Average rollback length of PCS

UTILIZATION FACTOR	$\rho = 0.25$			$\rho = 0.50$			$\rho = 1.00$		
#SIMULATION OBJECTS	256	1024	4096	256	1024	4096	256	1024	4096
baseline	1.06	1.02	1.01	1.08	1.02	1.01	1.08	1.02	1.01
cache opt.	1.97	2.55	3.84	2.06	3.55	4.14	2.13	4.01	5.08
cache+NUMA opt.	2.08	2.50	4.04	1.99	3.25	3.97	2.06	3.34	5.01

the combination of cache and NUMA optimizations enable in various cases the improvement of the execution speed compared to the scenario where the NUMA optimization is not considered.
For completeness of the analysis, we also report in Table 4.9.1 and Table 4.9.2 respectively the average rollback frequency and the average rollback length observed when running the different solutions. As the data show, the exploitation of spatial locality, that we remind is achieved by processing events not according to strict timestamp order within a time window W, leads to slightly increasing the incidence of rollbacks. However, it is limited, and the final trade-off between clock cycles spent for rollback operations and reduced clock cycles required for the forward processing of events provides the advantages seen in Figure 4.9.6.

The results discussed above have been obtained running a balanced configuration of the simulation model, where each cell, namely simulation object, has the same density of incoming calls. In order to test the effectiveness of the proposed solution for what concerns the

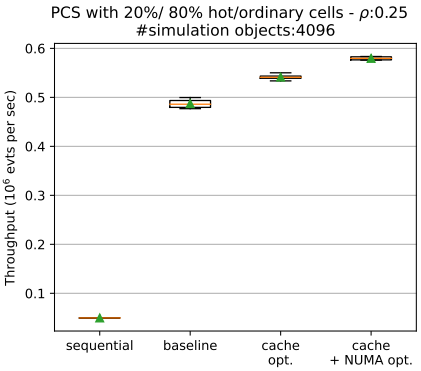

FIGURE 4.9.7: Execution speed

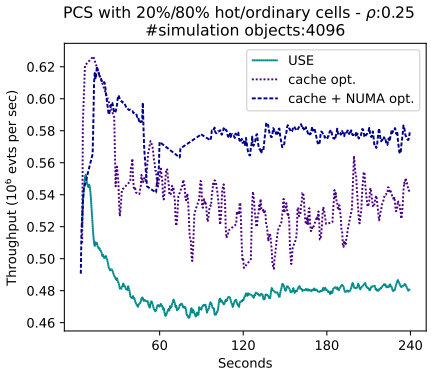

FIGURE 4.9.8: Event throughput for an individual simulation run

FIGURE 4.9.9: Results with PCS with 20% of hot-spot cells.

migration across NUMA nodes, we performed experiments where PCS is configured with two types of cells: hot-spot and ordinary cells. The first type represents the 20% of the simulated cells, and they are characterized by a higher density of calls than ordinary cells. During the simulation, the utilization factor of hot-spot and ordinary cells is respectively 0.5 and 0.25. In order to stress-out our NUMA rebalancing policy, we initially placed all the hot-spot cells on the same NUMA node, also loaded with ordinary cells. This kind of assignment makes a single NUMA node initially handling the 75% of the simulation workload. Figure 4.9.7 shows that, for this unbalanced configuration of PCS, our approach provides 18% speed-up compared to the baseline USE, with the NUMA optimization active. In these experiments, the NUMA rebalancing policy across NUMA nodes is enabled, as described in Section 4.6, to be performed periodically at each wall-clock-time second along the model execution. Limiting the optimization to the caching system (the *cache opt.* one), would not enable reaching such

performance benefit, but we note that the NUMA-aware optimization indirectly favours the cache-aware one, since more balance spreading of the accesses among NUMA nodes allows improving the overall exploitation of a larger amount of cache lines in the caching hierarchy. Furthermore, the automatic management of the migration of simulation objects across NUMA nodes offered by our approach allows removing the need for accurate setup of NUMA placement of simulation objects at startup time, which might be difficult in some cases, for instance with complex and/or non-isotropic models. We also report, in Figure 4.9.8, the relative throughput over time of each load-sharing scheme with respect to the throughput of events of the baseline USE for one of the runs. Initially, both the baseline and the cache-aware solutions are able to effectively exploit locality, however the model size grows while moving to the steady state configuration of the workload of calls to the wireless cells, and the cache benefits are eventually reduced (around 30 seconds for both the baseline and the cache-aware solution). Instead, enabling the NUMA-aware solution, including the migration policy of simulation objects, makes the initial part of the execution more expensive, but it allows achieving a more effective steady state due to the exploitation of cache and NUMA locality, according to the scheme we devised and thoroughly described. To widen the study of NUMA migrations and how they affect performance, we report in Figure 4.9.14 data related to a configuration of PCS where the hot-spot cells constantly vary. The variation takes place periodically, every 1000 simulated time units, and the newly selected hot-spot cells initially reside on the same NUMA node. In this experiment, a total of 4 variations of hot-spot cells take place along a simulated time of more

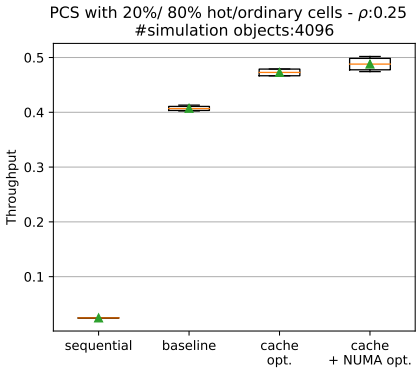

FIGURE 4.9.10: Execution speed

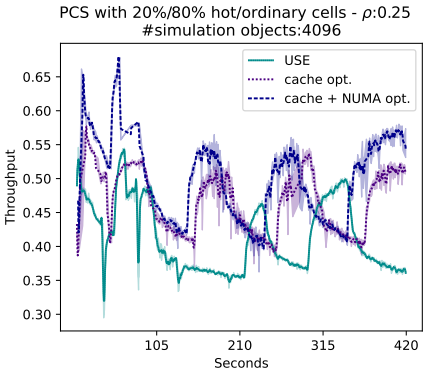

FIGURE 4.9.11: Event throughput for an individual simulation run

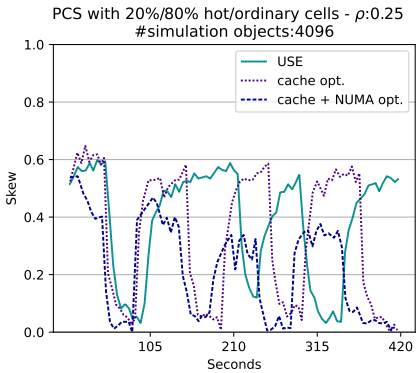

FIGURE 4.9.12: Workload skew between the two NUMA nodes

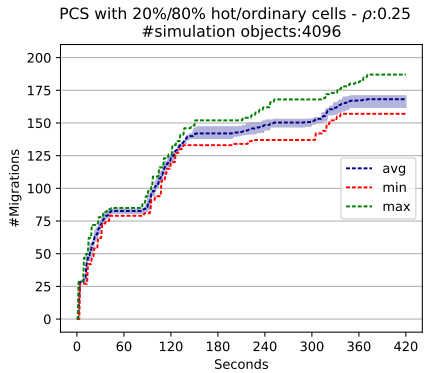

FIGURE 4.9.13: Average cumulative number of migrations.

FIGURE 4.9.14: Results with PCS with moving 20% of hot-spot cells.

than one hour. We report the execution speed observed over different runs in Figure 4.9.10, and also details related to an individual run, in Figure 4.9.11. As for the execution speed, we clearly see an improvement of 20%, when both cache and NUMA optimizations are active. In Figure 4.9.14 we also show the variation of the execution speed over time for an individual run, which highlights how our solution, especially if considering the NUMA optimization, allows achieving clear

event rate peaks when new hot-spots are selected, and how a well-balanced re-distribution of their states across the two NUMA nodes in the underlying machine favours the execution latency.
To further assess the workload balancing capabilities of this approach across NUMA nodes, in the right plot of Figure 4.9.12 we show the variation of the percentage skew of the execution time of the threads across simulation objects hosted by the two NUMA nodes for one of the runs of the PCS model. As we see, the configuration with active NUMA optimization allows keeping the skew close to zero, compared to both the baseline simulator USE and the configuration with the only cache optimization enabled.

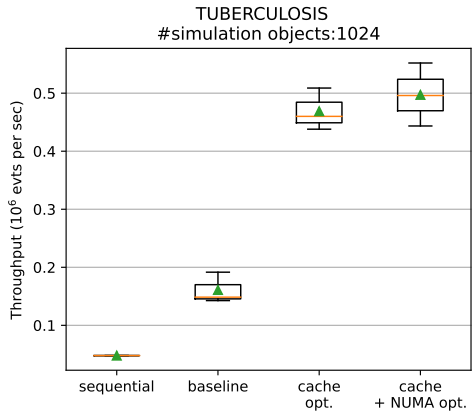

FIGURE 4.9.15: TBC execution speed

TABLE 4.9.3: Average rollback frequency of TBC

baseline	0.0010%
cache opt.	1.5610%
cache+NUMA opt.	1.6323%

TABLE 4.9.4: Average rollback length of TBC

baseline	1.13
cache opt.	1.51
cache + NUMA opt.	1.57

4.9.3 Results with TBC

The results for the model Tuberculosis (TBC) are shown in Figure 4.9.15. The data show the aggregate results over 20 runs. The advantages achievable for TBC through our solution are similar to the ones we have previously observed with the PCS model. In particular, the exploitation of both cache and NUMA optimizations allows our solution to be up to 38% faster than the baseline USE on the average. We also note that for this model, the events are very fine-grained (about 3μs), which gives rise to a reduced speed-up over the sequential run with respect to the PCS model results. Nevertheless, even in this scenario, our solution makes the parallel run be significantly more effective than the baseline USE, in terms of delivered performance. Also, as shown by data in Table 4.9.3 and Table 4.9.4, the reliance on the window W for the batch processing of events gives rise to negligible increase of the rollback frequency, which still favours the ability to exploit locality along the forward phase of execution of the simulation.

4.10 Final Remarks

The literature showed how speculative PDES systems can be reshuffled to better exploit the capabilities of multi-core shared-memory machines. The previously mentioned architectural reshuffle allows simulation objects to be bound to a specific worker thread for a short period of time, bounded by the execution time of an event. This architectural organization offers the advantage of improving efficiency and reducing

the likelihood of rollbacks, since the computing power is always in the proximity of the commit horizon of the simulation. However, despite the gain in efficiency, having a very short-term binding hinders locality, preventing the worker threads to fully exploit the underlying memory subsystem and caching hierarchy. This aspect is particularly relevant in NUMA systems, given the large impact, both in terms of latency and pressure on the interconnection, caused by memory accesses that need to be served by a distant NUMA node.

Following these consideration, the above-described approach provides a new workload-sharing mechanism to improve spatial and temporal locality exploiting the caching hierarchy and NUMA locality in order to improve overall memory utilization. In order to do so, we allowed a thread to prefer simulation objects either lastly executed by itself or by a thread sharing some cache level, or currently deployed on the same NUMA node. This binding holds until events targeting such simulation objects are not too far from the commit horizon of the simulation, a condition for which we provide an autonomous mechanism based on a dynamic virtual time window accordingly managed. Also, the scheme supports a mechanism to dynamically rebalance the simulation activities across NUMA nodes, achieved through the migration of simulation objects, which allows not penalizing any thread running on a processing unit on any NUMA node in terms of possibility to pick for execution a memory-latency favourable simulation object. In order to support this full scheme, we have developed a multi-view fully-shared event pool to improve locality during the worker thread activities that manage a set of events.

Overall, the proposed load-sharing scheme allows to continuously build

and maintain short-term binding between simulation objects and worker threads, favouring both cache and NUMA effectiveness. The results with both synthetic and real-world simulation models have shown that our approach introduces negligible overhead and significantly improves performance, even in the context of non-isotropic simulation models.

Chapter 5

Memory Aware and Lightweight Mechanisms for Incremental Checkpointing

In the previous chapter, we showed how to leverage memory hierarchy awareness in order to optimize event processing in speculative PDES platforms running on top of multi-core shared-memory machines. This allowed us to better exploit caches and NUMA nodes, in particular reducing the costs of misses, to improve the overall performance of the simulation applications, without directly impacting the simulation objects states.

However, efficient state management is necessary in the context of speculative PDES, since causality violations can occur and some checkpointing mechanisms are needed. This is particularly critical in PDES dealing with a large state simulation model and write-intensive workloads. Hence, efficient checkpointing techniques become paramount to

ensure correctness of the speculative execution. As mentioned in Chapter 2, checkpointing techniques can be distinguished between the full checkpointing, in which a copy of the entire state is performed, and the incremental checkpointing, in which only the modified portions of the state are saved. We focus on the latter one, since the tracking of the modified portions of the state is a crucial issue in the above-mentioned scenario, i.e. large-state and write-intensive simulation application.
The solutions explored to support the incremental checkpointing did not exploit write-protection services offered by common operating systems, as described in Chapter 3, but they were founded on the instrumentation of code at the binary level. The lack of consideration in the literature, except for what concerns fault tolerance, is motivated by the reduced granularity of the state of the simulation objects versus the page grain of the memory protection services. However, modern machines allow us to execute simulation with larger state and write intensive models, making write-protection services suitable for improving the effectiveness of the checkpointing.
Furthermore, the classical approach tackling incremental state saving, based on instrumentation of models, as described in Chapter 3, showed some limitations: in fact, for certain simulation workloads, i.e. large-state and write-intensive models, the instrumentation-based approach introduces an overhead during the event execution caused by the (multiple) interception of memory write instructions, which might increase the average event granularity, hampering performance or the energy footprint. Hence, the advantages that can be achieved, e.g. in terms of reduction of the size of checkpoints thanks to the incremental approach, might require excessive costs in terms of CPU cycles.

Based on these observations, we investigated the possibility of exploiting the write-protection services of common operating systems, in particular `mprotect()`, for building an incremental checkpointing solution suited for optimistic simulation in Posix systems, e.g. Linux. The proposed approach leverages memory update correlation through a buddy-pages scheme to identify and efficiently manage changes in the simulation state, in order to also reduce the costs associated to the write-tracking mechanism and the checkpoint operation. As a following aspect, we highlight the performance limits of this approach, due to the kernel-level activities of the `mprotect()` system-call. In fact, the `mprotect()` system-call needs to synchronize the memory updates (i.e. the change of permissions) across all cores, and this means that the Translation Lookaside Buffers (TLBs) must be flushed across all cores, with consequent non-negligible costs. Furthermore, the need for a signal handler to manage write interceptions, causes frequent user/kernel switches to detect the access to a single page. The performance bottlenecks of the `mprotect()` led us to develop a lightweight operating system service to support incremental checkpointing.
In particular, we present the design and implementation of a fully new system call, which enables protecting one or multiple pages from write operations without the need for interactions across the different CPUs in the machine and the threads they are running. Avoiding TLB shoot downs leads to drastically reduce the number of CPU cycles required for installing the protection for write activities by that thread on the pages. As a second step, we provide a system level service that allows acquiring information on what pages have been written in a work-deferred mode. Hence, we fully avoid the need for running

signal handlers when the page-write operations occur, eliminating the user/kernel switches' overhead. Rather, we retrieve the whole pool of dirtied memory pages just when the checkpoint operation needs to be carried out, also enabling the retrieval of multiple items (page identifiers) via a single user/kernel interaction. We developed this solution relying on Linux Kernel Modules (LKM) on Linux x86 processors, and we highlight the fact that the solution and software we will describe could be ideally exploited and/or adapted in/to contexts other than PDES, where checkpoint/restore activities may have a non-negligible impact on the critical path of the application execution (e.g., in terms of their frequency of occurrence).
Therefore, the techniques introduced in this chapter are directly relevant to the goals of this thesis, particularly in the context of improving the efficiency of speculative PDES on multi-core shared-memory machines.

5.1 Write-tracking Mechanism via `mprotect()`

As mentioned, we want to keep track of the modified portions of the state in order to efficiently support the incremental checkpointing operation. We want to avoid the multiple write-tracking problem of the instrumentation-based approaches, so we exploit the `mprotect()` system-call of Linux-based operating systems. We couple our system-call with a signal handler that manages the `SIGSEGV` signal. After protecting a memory area, when a denied memory access touches a protected memory area, the control flow of the program executing the

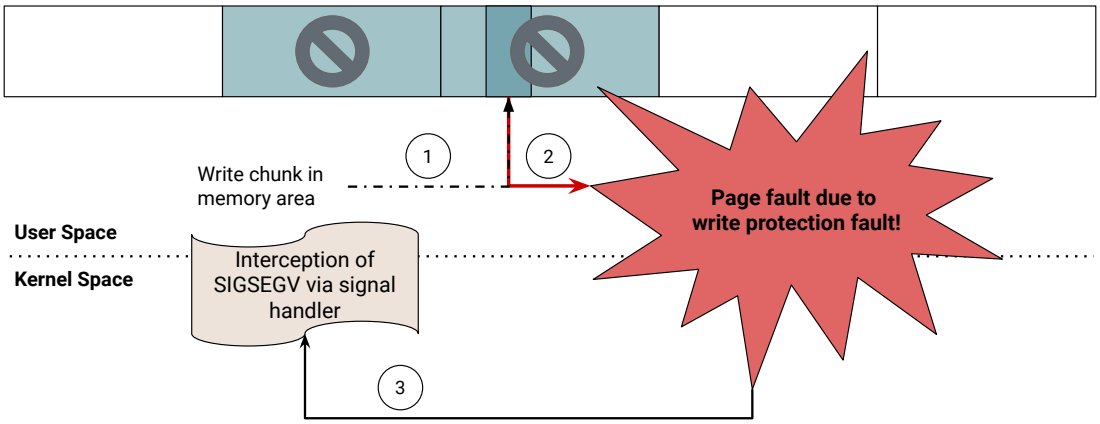

FIGURE 5.1.1: Illegal Write Access after Memory-protection via `mprotect()`

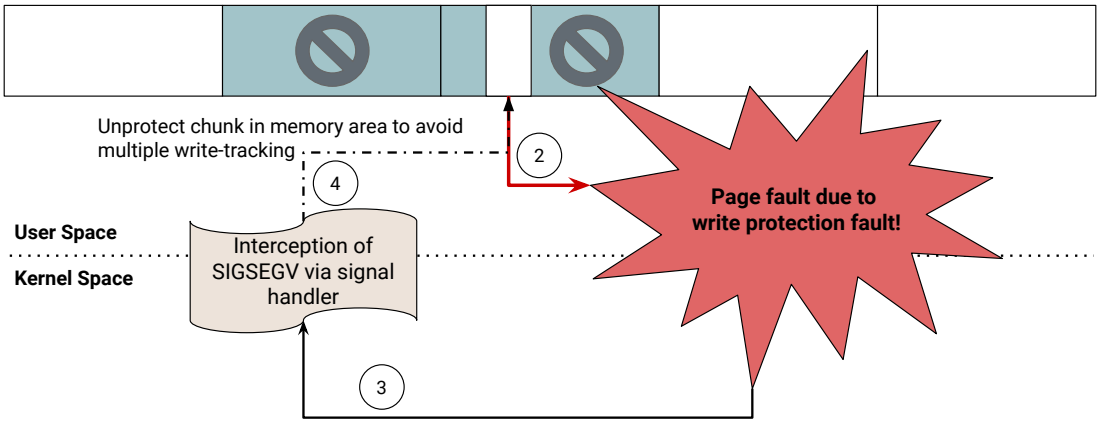

FIGURE 5.1.2: Write-tracking after Memory-protection via `mprotect()`

access is deviated due to the delivery of the `SIGSEGV` signal. We exploit such a mechanism to give control to a signal handler as soon as the first write touches a protected segment (Figure 5.1.1). At this point, such a handler can update a metadata table indicating what zone has been dirtied depending on the current write access, and then it relinquishes control to the original flow, which resumes from the offending instruction. We avoid the multiple write-tracking problem, present in the instrumentation based approach, by reopening the write access to

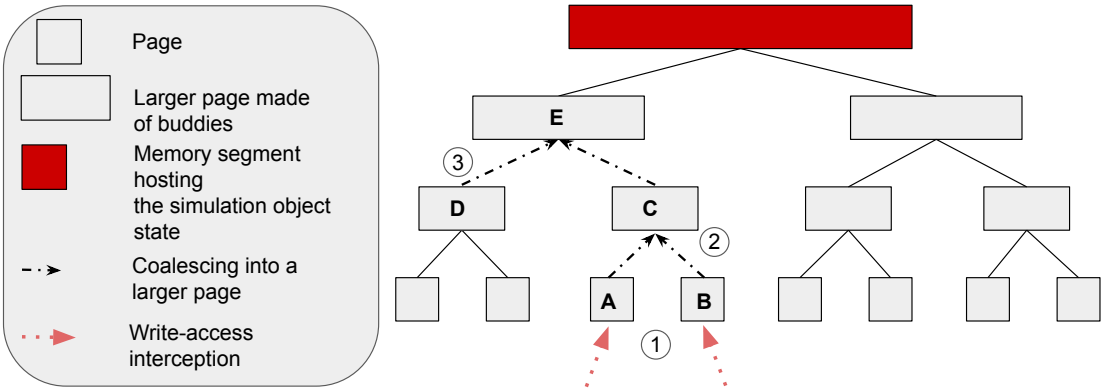

FIGURE 5.1.3: The buddy pages mechanism exploiting write-protection

the page after such an interception via SIGSEGV (see Figure 5.1.2). This includes updates that act on the same target memory locations already updated by some previous event on that same page. We then focus on improving the single page-based approach considering memory awareness of the simulation objects' states, introducing the notion of buddy pages in the memory segment used for hosting the state of a simulation object, as shown in Figure 5.1.3: two pages, namely A and B, are buddies if they are contiguous and aligned in the segment layout. If these pages are both written consequently to an event execution, or in general along a complete checkpoint interval, then we group them into one larger page, namely C. We then manage the larger page C in terms of (1) the checkpoint operation, (2) the write-protection service exploitation to re-open the write access and discard subsequent interceptions for the coalesced page. The coalescing of pages can be iterated for C and a same-size page, namely D, in order to further reduce the memory protection costs for intercepting single-page accesses. In fact, optimizing the usage of operating systems services, namely reducing the number of calls to `mprotect()` and the number of `SIGSEGV`

signals intercepted, is crucial in optimistic simulation, because out-of-order errors are endemic and require checkpointing not to be executed too infrequently. The buddy pages scheme is coupled with a decision model to help with the write-correlation operations, described in the next section.

5.2 Decision Model for the Memory Aware Incremental Checkpointing

We showed the developed mechanism to reduce the intrusiveness of the instrumentation-based approach in terms of instructions in the simulation model, exploiting operating systems' services to protect memory, i.e. `mprotect()`. This leaves the simulation model unaltered and allows us to track memory write operations without incurring in the above-mentioned problems related to the multiple write-tracking for the same memory area. We also showed a buddy-pages scheme devised to reduce the costs associated to the signal handler associated to the `mprotect()` and the cost of saving the state. In order to further reduce the overhead of managing single page accesses, we devised a mechanism to correlate write operations on memory pages.
In particular, we considered a pre-reserved memory area M to host the state of the simulation object, and we then partitioned it into groups of pages according to a decision model that partitions the pages in a way to consider them as a unique zone to be managed by a single `mprotect()` call and by the checkpointing facility. Since the `mprotect()` system call can manage multiple contiguous pages at the

same time, we would like to compute the partitioning that minimizes the overall costs, namely the clock cycles needed for the interception of the `SIGSEGV` signal and the reopening of the write access on the target memory area, and the checkpointing costs.
Assuming that these costs can be accurately estimated, finding the best partitioning can be formulated as the following optimization problem:

$$\min_{\mathcal{Z}} \sum_{i \in \mathcal{Z}} C_{P,i} + C_{L,i} \tag{5.1}$$

where $\mathcal{Z}$ is a partitioning scheme of the memory area M, $C_{P,i}$ and $C_{L,i}$ are the costs for protecting/unprotecting and logging the partition i of the segment, including the cost for tracing its access via the `SIGSEGV` handler.
A naive algorithm for solving this optimization problem involves enumerating all possible partitioning schemes and choosing the minimum-cost one. However, such an approach requires exponential time. In fact, the total number of partitioning schemes evaluated is equal to 2^{n-1}, where n is the number of memory pages within the area M. This can be shown considering that: (1) the number of points in which we can split the memory area is equal to $n-1$; (2) each point provides two possible choices; namely, it can be considered or not. Consequently, we can encode the choice of each individual separation point as a bit within a string of $n-1$ bits that can assume 2^{n-1} different values. For this reason, we opted for considering a reduced set of partitions, namely those that can be generated according to the buddy-system scheme—in fact, our solution focuses on exploiting buddy pages and their correlated accesses in write mode.

The buddy scheme imposes that (1) partitions contain a number of pages that is a power of two, and that (2) the starting address of each partition is aligned to its size within the area M. These constraints make each partition being composed of two halves, called buddies, that are aligned to their size, contiguous to each other, and composed of two smaller buddies with the same properties. Imposing that partitions are generated according to the buddy-system specification makes the number of partitions to be considered linear to the number of elements in the set. In fact, we can consider the n memory pages of the area M as the leaves of a complete binary tree with $2n - 1$ nodes. Each non-leaf node of the tree mentioned above represents a non-minimal partition that can be protected/unprotected with an individual `mprotect()` call. However, even though the number of admissible partitions increases linearly with the number of pages within the memory area, enumerating and evaluating all partitioning schemes is still unfeasible. In fact, it can be shown that they increase exponentially in the number of pages.

The main benefit of resorting to a buddy-system scheme is that the new problem formulation has an optimal substructure, namely, the optimal solution can be built from the optimal solutions of sub-problems. Intuitively, the optimal solution is either protecting/unprotecting the whole segment of memory m or it is the union of the optimal solutions for the left and right halves of m , denoted as m_L and m_R , respectively. In particular, we can easily show that:

$$C^*(m) = \min\left(C_{P,m} + C_{L,m}, C^*(m_L) + C^*(m_R)\right) \qquad (5.2)$$

where $C^*(i)$ is the cost of the optimal solution for a memory segment i . The following theorem proves Equation 5.2.

Theorem 5.1. *The optimization problem in Equation 5.1 with partitions compliant with the specification of the buddy system has the suboptimal structure shown in Equation 5.2.*

Proof. We prove the statement by reduction to absurd. Let us assume that an optimal solution with cost $O^* < C^*$ exists. Such a solution cannot contain a single partition that includes the entire segment m. In fact, such a case imposes $O^* = C_{P,m} + C_{L,m}$, contradicting the hypothesis. The buddy system specification imposes no partition within the optimal solution that crosses the middle of the segment. Consequently, any partition of the optimal solution belongs entirely either to the left half m_L or to the right half m_R of the segment m. It follows that $O^* = O^*_L + O^*_R$, where O^*_L and O^*_R are the costs associated to m_L and m_R. Furthermore, since the O^* is optimal, we know $O^*_L + O^*_R < C^*(m_L) + C^*(m_R)$, suggesting that $O^*_L < C^*(m_L) \vee O^*_R < C^*(m_R)$. This contradicts the hypothesis that both $C^*(m, L)$ and $C^*(m, R)$ are costs associated with the optimal solutions for m_L and m_R. □

Proving the presence of a suboptimal structure allows us to build a simple algorithm to pinpoint the optimal partitioning scheme. Initially, we map each page to a unique partition. Then, we check each pair of buddy pages and, if costs associated with a pair are higher than considering an individual partition including both, the two buddies are moved into an individual contiguous partition whose size is twice

Algorithm 5.2.1 Memory Partitioning Algorithm

```
1:  procedure FindOptimalSolution( )
2:      cost_t tree[NUM_OF_PAGES ·2]
3:      bit_t optimal[NUM_OF_PAGES ·2]
4:      int start ← NUM_OF_PAGES
5:      int end ← 2 · start
6:      for int i ← start to end − 1 do
7:          tree[i] ← compute_cost(i)
8:          optimal[i] ← 1
9:      end for
10:     while start ≥ 1 do
11:         end ← start
12:         start ← start/2
13:         for int i ← start to end − 1 do
14:             tree[i] ← compute_cost(i)
15:             cost_t child_sum ← tree[i/2] + tree[i/2 + 1]
16:             if tree[i] < child_sum then
17:                 optimal[i] ← 1
18:             else
19:                 tree[i] ← child_sum
20:             end if
21:         end for
22:     end while
23: end procedure
```

a single page. If there is any couple of new larger buddies, we check again if it is convenient to merge them, repeating the process until no buddies of any size can be merged. Since there are $2n - 1$ admissible partitions and each is considered once, the algorithm runs in linear time.

Algorithm 5.2.1 shows the pseudocode of the proposed approach. It relies on an array-based representation of a static binary tree, where each node at index i has its left and right child at index $2i$ and $2i + 1$, respectively. The root is placed at index 1. First, the algorithm initializes each leaf—corresponding to a memory page—by computing

its costs and by associating a partition to each individual page. Then, it proceeds by scanning each level of the tree until the root is reached. Whenever a partition has associated costs lower than the sum of its two halves, it is marked as a candidate for belonging to the optimal solution. The partitioning scheme is finally computed by identifying all the highest-level partitions marked as candidates to be included in the optimal solution, which enables covering all the memory pages in the area.

5.2.1 Estimating Costs of the Buddy-based Approach

The algorithm presented in the previous section assumes that the costs $C_{P,i}$ and $C_{L,i}$ for respectively protecting and snapshotting a memory region i are known. This can be achieved by running a micro-benchmark which evaluates the cost of protecting and copying each different-sized partition. However, such an approach allows us to roughly estimate the synchronous costs of `mprotect()` invocations and the costs of logging when a write access occurs to the protected region. Clearly, the cost associated with logging a never-written region is zero. Consequently, we redefine $C_{L,i}$ as the expected cost for logging the region i, which can be computed as $C_{L,i} = P_w(i) \cdot C_C(|i|)$, where $P_w(i)$ is the probability that at least one write access targets i, $|i|$ is the size of the memory region i, and $C_C(s)$ is the cost for copying the content of a region whose size is s.

We keep track of $P_w(i)$ for any admissible partition i as the frequency of first-write accesses in a time window. In particular, we store the number of first-write accesses N_i for each partition i using a complete

static binary tree. The root of this tree keeps N_m of the whole segment m, its left and right children keep N_{m_L} and N_{m_R} for the left and right half of the segment, and so on until we reach the leaves, which track N_p for each memory page p. At this point, $P_w(i)$ can be computed as N_i/N_m when $N_m > 0$, that is, as the ratio between the number of first-write occurrences targeting the partition i and the total number of first-write accesses targeting the whole memory segment. Clearly, if $N_m = 0$, $P_w(i) = 0$ for any partition i.
Since we need to traverse the tree from a leaf to the root at each first-write access, keeping updated the static binary tree requires $k \log_2 n$ time, where n is the number of memory pages and k is the number of first-write accesses.
The estimation of costs is exploited to re-determine the partitioning of the memory area periodically by re-executing Algorithm 5.2.1. This also enables us to deal with simulation models where the access patterns of the events to the state layout change over time.

5.3 Experimental Evaluation

5.3.1 Test-bed Environment

Although the proposed solution is usable in generic speculative simulation systems, we studied its integration in the USE platform [54].
As mentioned in previous sections, we used DyMeLoR [139] to integrate our new incremental checkpointing support in USE. In particular, USE offers a standard programming model for the event handler, which

enables the handler to rely on dynamic memory allocation/deallocation, as typically offered by standard libraries. Whenever a simulation object needs to allocate memory for its state, it requests a chunk from a memory allocation service. DyMeLoR intercepts this request and selects a non-busy chunk from a pre-allocated memory area to satisfy the request. At the same time, the pre-allocated storage for chunks of different sizes is embedded into different portions (hence different pages) of the whole memory area destined for the usage of the simulation object. It is worth noting that the pages pre-reserved for hosting the data chunks are only memory mapped. This means that if a simulation object does not actually use them, they will not take up space in RAM and will require no checkpointing (i.e. memory copy) operation for storing them.

Through this architecture, we maximize the locality of the memory chunks to be used when requesting memory of a given size (which fits the size of these chunks). At the same time, the locality of operations on chunks of different sizes deals with different pages. This can be extremely useful in scenarios where the simulation model may have different memory access profiles (read vs write) on data structures, relying on the linkage of different-size chunks. The incremental checkpointing solution presented in this thesis can operate in this scenario by determining what page, which host the chunks that are accessed in write mode in a correlated manner, can be grouped together when forming the partitions based on the decision model presented in Section 5.2.

As for the underlying hardware, we exploited a machine equipped with an Intel i7-12700K with 12 CPU cores (20 Hardware Threads) and

64GB of DDR5 RAM. The operating system is Ubuntu 22.04 (kernel version 5.19).

5.4 Benchmark Application

The goal of the presented solution is not to overstep existing solutions, but instead to consider a specific scenario not covered by the literature. This is the workload of models with large-state and with events consisting of write intensive zones of the state (hence of set of pages). Checking with models used for evaluating optimizations in speculative engines for parallel simulation, we identified PCS (Personal Communication System), see Chapter 2, as a good test-bed for assessing our incremental checkpointing solution. As already described, in this model, each simulation object is in charge of simulating a wireless coverage area (a cell), and each device currently active in this area requires keeping entries in multiple lists belonging to the state of the simulation object. In particular, there is a list that includes basic information about an active device, such as the identification of the channel used for communicating. However, the model can be configured to keep additional lists depending on the level of granularity according to which the wireless communication needs to be simulated. In particular, another list is used to manage the power assigned to each device call, also depending on a time-variable fading factor. Furthermore, the state of each simulation object keeps an area for storing statistical data referring to different simulation time periods, which are used for the production of output data by the simulation. In our setup of

the model, we used all its facilities, hence having multiple linked lists within the state of each simulation object. Also, the model has events requiring write-intensive access to the state, in particular to the list of power information records—to update it based on variations of the fading that impacts each call.
We have run this model by considering cells equipped with 1000 channels each, where the workload of calls leads to a probability of busy channel of the order of 50%. The overall size of the state of each simulation object resulting from this configuration is of the order of 80KB (20 pages). The total number of simulation objects has been set to 256, leading to simulate a total count of wireless channels equal to 256000.

5.5 Results

We report data comparing the solution based on exploiting the `protect()` systems-call with an incremental checkpointing solution where we intercept memory-write accesses by the event handler to the state of the simulation objects via a macro. This is an ideal solution where instrumentation takes place at the software-source level, and enables the compiler to generate an executable with minimal number of machine instructions required for the write-access tracing mechanism. Moreover, we rely on the incremental log of dirty chunks, which is an alternative compared to the incremental log of dirty (buddy) pages we exploit in the proposed solution. In the plots, we refer to the competitor as *ISS-instrumentation* while we refer to our solution as *ISS-buddy*. For completeness, we also report data gathered through a page-level

protection scheme with no actual partitioning of memory depending on the access pattern to buddy pages. We refer to this solution as *ISS-page*, and we note that it is useful in order to show the effects of our optimization technique exploited in *ISS-buddy*, which is based on the decision model we have presented in Section 5.2. For fairness in the comparison, we executed each run by using the same checkpoint interval for all the techniques tested, exploiting infrequent full checkpoints in all the cases. In particular, we set the USE runtime system to take a full checkpoint each 10 checkpoint operations, while all the others are incremental. Additionally, given that the USE environment has been designed in order to optimize the usage of the CPU in speculative simulation, in particular by generating executions that highly likely suffer from very minimal rollback, we have decided to set the checkpoint interval to be used in all the tested techniques to the value 80. This is a kind of limit value for enabling memory recovery upon GVT calculation, which is well suited for scenarios where the rollback frequency is very minimal and the state size of the simulation objects is large, as for the case of our test-bed application. All runs have been executed on top of 20 CPUs (namely hyperthreads) of the underlying machine, and we have observed that the model is executed with a rollback frequency less than 1%.

We report in Figure 5.5.1 the speed of the simulation execution when running with the three different solutions. In particular, we report the number of committed events per unit of wall-clock-time. Each value is computed as the average over 10 different runs, each executed by relying on different seeds for the pseudo-random generation. The results show how *ISS-buddy* enables performance improvement compared to

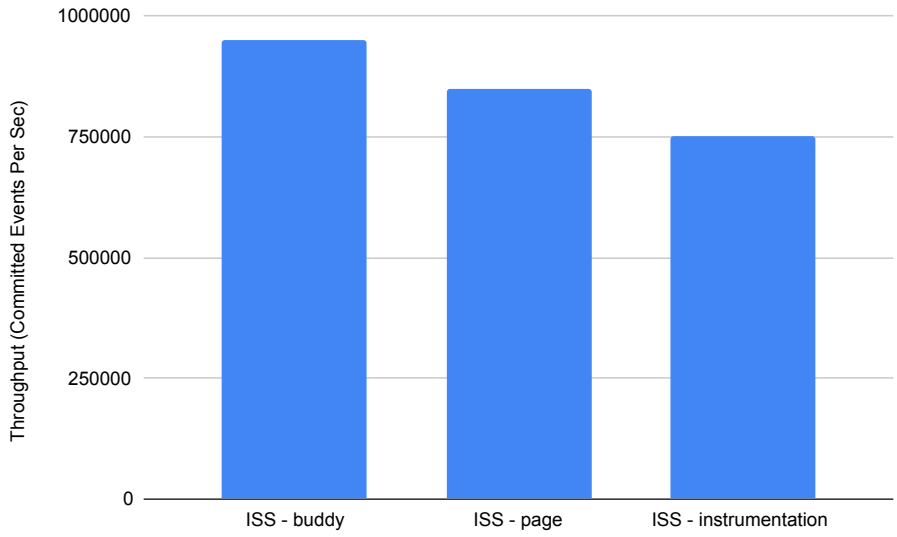

FIGURE 5.5.1: Simulation execution speed

ISS-instrumentation, in particular by enabling the simulation run to commit 22% more events per wall-clock time unit. Furthermore, it allows 12% improvement of the number of committed events per wall-clock time unit when compared with *ISS-page*. This is an indication of the effectiveness of the partitioning scheme based on buddy pages for the reduction of the impact of operating system services. Finally, in Figure 5.5.2 we report data related to the size of the incremental checkpoints for the three techniques, comparing it with the size of the full checkpoint that is infrequently exploited in all scenarios. As the plot shows, the incremental checkpoint with minimal size is achieved through *ISS-incremental*, which however shows the worst performance as we discussed. At the same time, the incremental checkpoint that is achieved through the *ISS-buddy* technique is definitely lower than the size of the full checkpoint. This indicates how *ISS-buddy* still

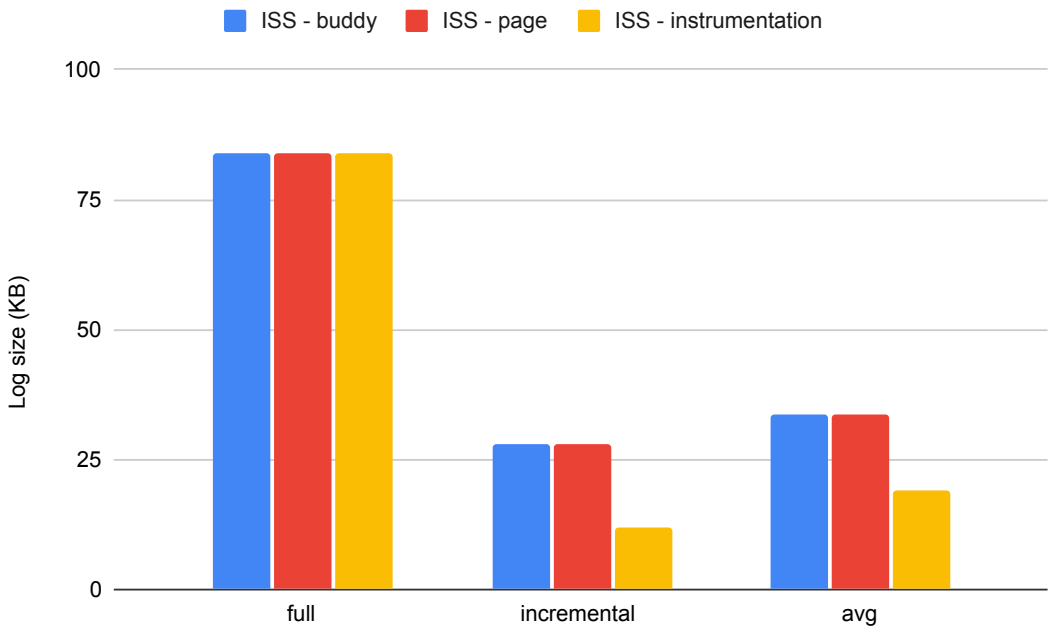

FIGURE 5.5.2: Checkpoint sizes

supports large memory usage reductions compared to the full checkpointing technique. Also, this reduction is similar to the reduction that would have been achieved by relying on *ISS-page*, which, as discussed before, introduces a higher cost than *ISS-buddy* in terms of CPU cycles requested for supporting the incremental checkpointing technique.

5.5.1 Considerations

We showed a new way of tracking modified portions of simulation objects states in speculative PDES, leveraging operating systems services, such as the `mprotect()` system-call, in order to avoid the multiple write-tracking problem of the instrumentation-based approach. We armed a signal handler to manage a subsequent illegal access, and in

order to reduce the costs associated to the state saving and the management of a single-page write-tracking, we devised a buddy-pages scheme to manage correlations between writes. We showed that the use of memory write correlation on buddy-pages reduces the overhead of incremental checkpointing by grouping memory updates with spatial locality, efficiently handling write-intensive workload.
However, there is still room for improvement in terms of performance, and in particular for what concerns the overhead given by the `mprotect()` system call. In fact, this service leads to a suboptimal solution for our tasks due to two aspects previously described, at the beginning of Chapter 5. To recall, some aspects that can still be targeted for performance improvement are related to the following facts:

- The `mprotect()`, used to intercept write operations on a memory area, needs to guarantee not only that the memory area is not writeable at a given time, but also that every other processing unit (working on the same address space) has a coherent view of the address space. This means that the operating system, at some point, will flush the TLB of all CPUs, at least for the range of addresses targeted by the protection operation. The TLB flushing is a heavy procedure due to the Inter-Processor-Interrupts (IPIs) management (sending and handling).

- At each memory protection requested and subsequent illegal write-access, a signal handler is executed. This leads to frequent user/kernel switches to detect the write access to a single page. Also,

write accesses are always identified uniquely, without the possibility of grouping them and managing them in a single write-interception phase.

In the next section, we will describe how we tackled these aspects, and reduced the overhead and intrusiveness of memory protection services when used to support the incremental checkpointing procedure in speculative PDES platforms.

5.6 Lightweight Operating System Service for Incremental Checkpointing

It is worth highlighting that improving the efficiency of PDES platforms on multi-core shared-memory machines regards not only devising advanced simulation algorithms, but also improving the underlying support provided by the operating system. In particular, in the previous section we described how exploiting Linux-based operating systems services can improve the performance of classical speculative PDES operations, such as the incremental checkpointing. However, we also showed how operating system services are not always tailored to the specific needs of simulation workloads, especially if they exhibit a fine-grained data sharing. In order to tackle the above-mentioned critical aspects of the `mprotect()`-based approach to support the incremental checkpointing, we designed and implemented a Linux Kernel Module (LKM) technology based on a fully new system call to enable the protection of one or multiple memory pages from write operations

without the need for interactions across CPUs – and so the threads they are running. In fact, it can be called by the thread in charge of a certain simulation object in order to build a local view of the memory that only disables its own write-access to the target pages. In this way, we avoid an excessive use of IPI and the handling of these interrupts at kernel level, drastically reducing the number of CPU cycles required for protecting memory for write activities by that thread on the pages. As a second aspect, we provided a system level service, still based on LKM, that allows acquiring information on what pages have been written, in a work-deferred mode. In fact, we avoid the need for running signal handlers when the illegal write operation occurs. Rather, we retrieve the whole pool of written memory pages when the checkpointing operation needs to be carried out also enabling the retrieval of multiple pages via a single user/kernel interaction.

As an additional note, our approach does not prevent that a simulation object is re-bound to a different thread while being within a checkpoint interval, since our architecture enabled the migration of the memory protection locally installed for a thread to another thread. This gives ride to a solution that is independent of any rebalancing policy of simulation objects to threads.

As mentioned, these services are embedded within an LKM, consequently they can be installed with no need for recompiling the kernel. To build our operating system service for incremental checkpointing, our architecture guarantees that any individual virtual page does not contain memory chunks that are used by more than a single simulation objects at a given time, as already discussed in Chapter 2. In fact, we recall that each simulation object has a disjoint set of pages mapped

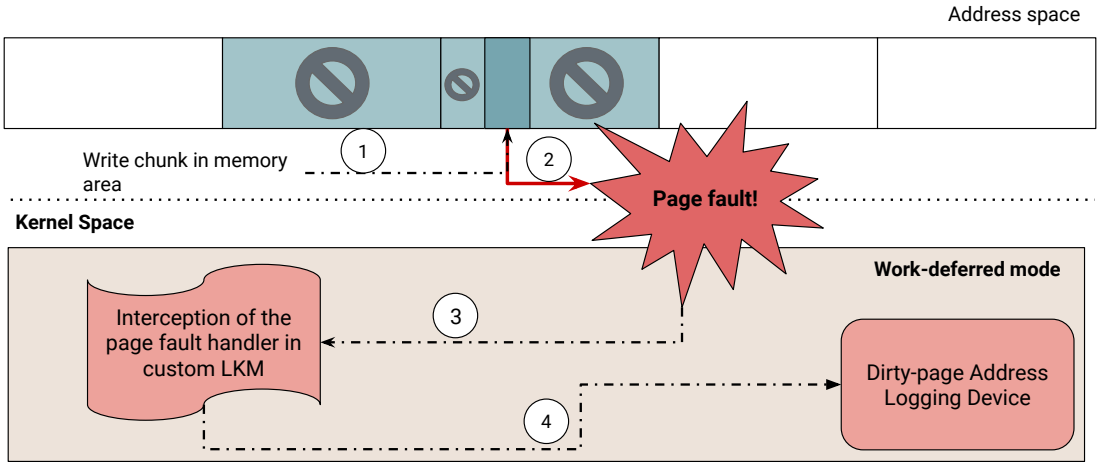

FIGURE 5.6.1: Write-tracking after Memory-protection via `track_memory()`

via the `mmap(..)` system-call. This allows us to develop an API to manage the state of the allocation system for each simulation object, since it is capable of identifying object-specific memory mapped areas. Starting from this baseline, in the following subsection, we will describe how the support for the incremental checkpointing is structured.

5.6.1 Write-tracking Mechanism via LKM

Each memory area, let's say the i-th one, used for allocating chunks for an object is associated with the virtual address interval $[START\_ADDRESS_i, END\_ADDRESS_i)$. All the logical pages with bytes falling in that interval are managed via the interception of the page-fault execution logic at the level of the Linux kernel (see Figure 5.6.1). The interception has been based on the exploitation of the kprobe subsystem offered by the Linux kernel.

At the same time, our custom system call for protection memory, `track_memory(..)`, works at the level of the `x86` page table entries used by Linux for managing these pages, and exploits bits that are unused by the `x86` processor, in particular bits 9-11 of the PTE table, to instantiate and manage a state machine that is exploited and updated by our page-fault interceptor (see Figures 5.6.2 and 5.6.3). The following two states are permitted while managing each page in that area:

- NORMAL, in this state the interceptor of the page-fault simply returns control to the page-fault handler to execute the default procedure for managing a page fault occurred due to the accesses to the object state by the event handler modules. In other words, this page fault is not caused by a write-access to a page marked by our system for its insertion in an incremental checkpoint of the simulation object state (although we will describe a corner case).
- WRITE-PROTECTED, in this state the interceptor of the page-fault handler logs the address of the write-accessed page into a backend data structure managed by a device driver, in order to keep track that it will need to be involved in an incremental checkpoint operation. In order to correctly generate a page-fault when a write-access is issued by the event handler software, while being in this state the real bit in the page table that enables writing the page (bit 1 in the PTE) is reset, and is set up again after the interception of the write access.

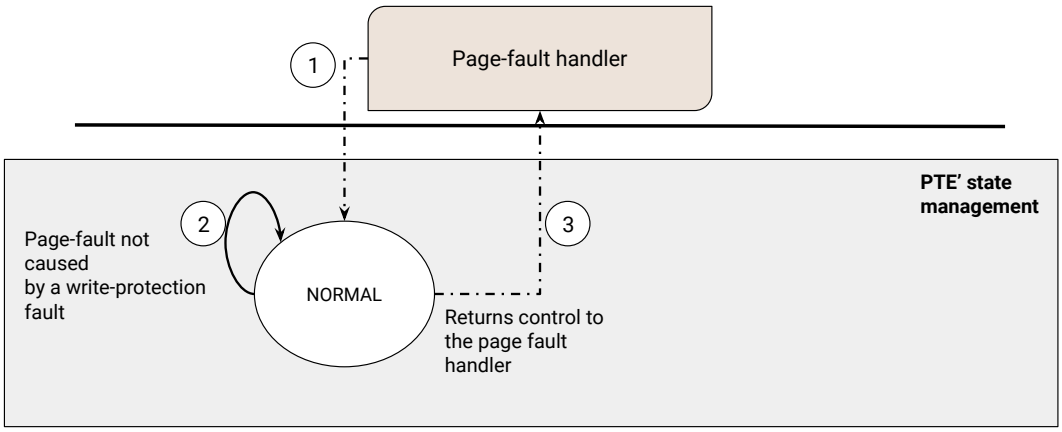

FIGURE 5.6.2: Page-fault Not Caused by Write-protection

When the page-fault interceptor finds a page table entry in the WRITE-PROTECTED state upon a write-access, beyond registering the address and resetting the write-bit, it also moves the page state to NORMAL. This way, future writes on already dirtied pages to be included in the checkpoint of the simulation object state will not be intercepted, thus eliminating the cost associated to multiple write-tracking of the same memory area, as for the approach based on `mprotect()` described earlier.

Additionally, it is important to note that unused bits by the `x86` processors in the page table entry, which we exploit to build and manage our state machine, can be used without incurring in the risk of conflict to the Linux kernel only when the page is really materialized in a RAM frame, so when the page table entry is valid. In fact, if the page table entry is not valid, e.g. it is still not materialized, Linux actually uses it, possibly overriding the aforementioned bits, for internal operations.

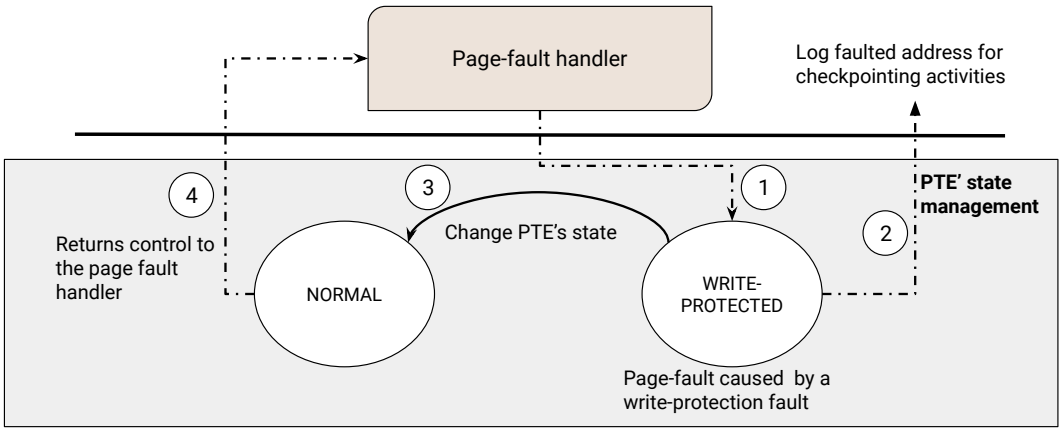

FIGURE 5.6.3: Page-fault Caused by Write-protection

This is the corner case we were referring to earlier, and it poses a problem related to the interception of the first write access to a page that is not yet materialized in RAM.

To cope with this problem, our page fault interceptor has been expanded with a logic that allows the identification of the write-access to a page that has the corresponding page table entry which is not valid. For this page, the address is also logged in order to correctly include the page in an incremental checkpoint operation that will be eventually requested.

When a new checkpoint interval needs to be started for a simulation object, all the already materialized pages used to keep chunks that belong to its state need to be transited to the WRITE-PROTECTED state. To reach this objective, the thread managing the object invokes our `track_memory(..)` system call. The parameters passed to it in order to correctly identify the entries of the page table to be updated are therefore $START\_ADDRESS_i$, and the number of pages of the

area, computed as

$$\lceil \frac{END\_ADDRESS_i - START\_ADDRESS_i}{4KB} \rceil \qquad (5.3)$$

(we recall that on x86 processors pages have size 4 KB).
If multiple mapped memory areas are used for keeping the chunks of an object, then this system call can be invoked multiple times, in order to change the page state for all the virtual pages that could be involved in write-access operations by the simulation object, which might be placed on the different mapped memory areas.
At the same time, we exploited free bits in the Virtual Memory Area (VMA) data structure of the Linux kernel – we recall that Linux keeps one VMA for each mapped memory area – in order to record that the whole area needs to be exploited in write protection manner for supporting incremental checkpointing. This information is used, as pointed out before, to correctly track write-accesses of not yet materialized pages, which are not passed to the WRITE-PROTECTED state by the system call, since the x86 unused bits in the page table might be actually already used by Linux.
Overall, when a thread manages a simulation object, it can start the write-interception phase to the pages hosting the memory chunks by simply calling the `track_memory(..)` system call. In its turn, this system call does not need any IPI to signal other CPUs that something is occurring in terms of page table modifications, since there is no need for other threads to perceive this change and so to synchronize. The cost for the management of IPIs is therefore completely

saved, compared to the previously described `mprotect()`-based approach, as it occurs when using common operating system services for tracking page-write accesses. Similarly, the cost for running the handler of the interrupt to the remote CPUs, which ultimately yields to managing the TLB for a possible flush, is completely saved.
At the same time, we note that the IPI technology exploited for page-protection management, along with the associated TLB flush, operates at a global level in the architecture. Hence, the larger the number of CPUs, the larger the expected intrusiveness caused by IPI-based solutions. This likely leads our alternative operating system services to better provide advantages with larger CPU counts in the machine, hence naturally scaling vs the underlying computing power offered by the machine.
The above described system call is coupled with another analogous system call, namely `untrack_memory(..)`, used to restore the possibility to write data to the pages of a specific memory area when a rollback of the simulation object occurs, so to recover from causality violations, and previously checkpointed information needs to be re-written in that area. We note that classical approaches based on `mprotect()` still need to re-open write-accesses in a rollback phase, thus again facing the same issues related to the IPI architecture, which are instead avoided in the solution discussed in this thesis.
The solution proposed in this thesis also offers the possibility to run the checkpoint interval of a simulation object, made by whatever number N of simulation events, across different threads. This can be particularly relevant in all the speculative PDES platforms, where the switch of the assignment of objects to threads can take place wherever along

wall-clock-time. In this scenario, different threads need to pass through the interception of write-accesses involving the state of the object while processing events. Hence, calling the `track_memory(..)` system call along a single thread A at the beginning of the checkpoint interval of the simulation object, which leads to updating the page table entries and the unique TLB of the CPU where thread A is running, does not guarantee that another thread B running on another CPU can actually exploit the new management rule of the pages containing the state of the simulation object. To solve this problem, our `track_memory(..)` system call offers also the possibility to simply flush the TLB on the processor it is invoked, and this is achieved by simply passing the value $NULL$ as its first parameter. Exploiting this behaviour, the thread B that becomes responsible for one or more simulation objects that were previously managed by another thread A can simply execute the system call passing $NULL$ as input. This way, the TLB of the local CPU will be flushed, enabling therefore the correct management of the write-access interception, based on the updated value of the page table entries. At the same time, we still avoid multiple interceptions of write operations on the same page along the checkpoint interval, since a page that is already in RAM and that has been transited to the NORMAL state after a write interception, will not generate any new page fault when thread B writes again on it, in fact the write-bit for that page has already been set.

As a final consideration, a thread that has flushed its TLB when running on a specific CPU could be dynamically migrated by the operating system kernel to another CPU, just for load balancing. Hence, the reset of the TLB, for the correct exploitation of the new page table content,

might be made useless, in particular when the target CPU after the migration has just run another thread of the same process—in this case the page table pointer register `CR3` of `x86` processors is not necessarily rewritten, and the TLB could therefore yield stale data compared to the updates performed on the entries of the page table for write-protecting pages according to our solution. To cope with this problem, we have installed another `kprobe` to the context switch function of the Linux kernel, which simply flushes the local TLB when the current thread is switched. Overall, we can support incremental checkpointing with our services in speculative PDES platforms where pinning of threads to CPUs can be either used or not.
At the same time, all the aforementioned operations are executed only if the thread, which takes the CPU upon the context switch by the operating system, belongs to a process that is registered via an `ioctl(...)` call within a `/proc` entry we included in the virtual file system through our LKM. This check is carried out also for what concerns the access to the page tables and the VMA entries when running the interceptor of the page-fault handler. Hence, we completely avoid intrusiveness on threads that are not running the PDES engine (e.g., daemon-threads of the kernel).

5.6.2 Dirty-page Address Logging Device

Our support for incremental checkpointing also offers optimization in terms of passage of data, namely the addresses of the dirty pages, from kernel to user space. These optimizations tackle two aspects. Firstly, we enable the retrieval of information for each individual simulation

object. This facilitates the post-processing of the information at user space, since there is no need to parse data for determining which object is associated with the page addresses returned from the kernel. Secondly, we enable the user level software to retrieve a batch of addresses of dirty pages via a single system call invocation. As highlighted, this enables avoiding the costs spent for retrieving the same information via signal handler, which would be activated at each interception of the write operation, as we previously described for the `mprotect()`-based approach.

We note that both the configuration and the usage of the kernel level software device we designed for logging and delivering this information are extremely simple and do not require additional installation system calls. In fact, all the different actions to interact with the device are triggered by relying on the standard `ioctl(...)` system call. Still in relation to this aspect, the software device we designed corresponds to the one previously mentioned, which appears as a char-device in `/proc`, as we outlined, and it allows registering at kernel level the identifier of the process for which the incremental checkpointing support and the associated kernel level tasks need to be activated.

The exploitation of the `ioctl(...)` system call allows us to pass to the device different commands. One of these commands, named, `GET_DIRTY_PAGES`, for which our LKM offers the implementation of the kernel-side function, is used to retrieve the addresses of the pages that have been written along the current checkpoint interval. One additional parameter that is passed is the Object Identifier (OID) for which this information needs to be retrieved, while two more parameters are the address of the memory area where the information needs

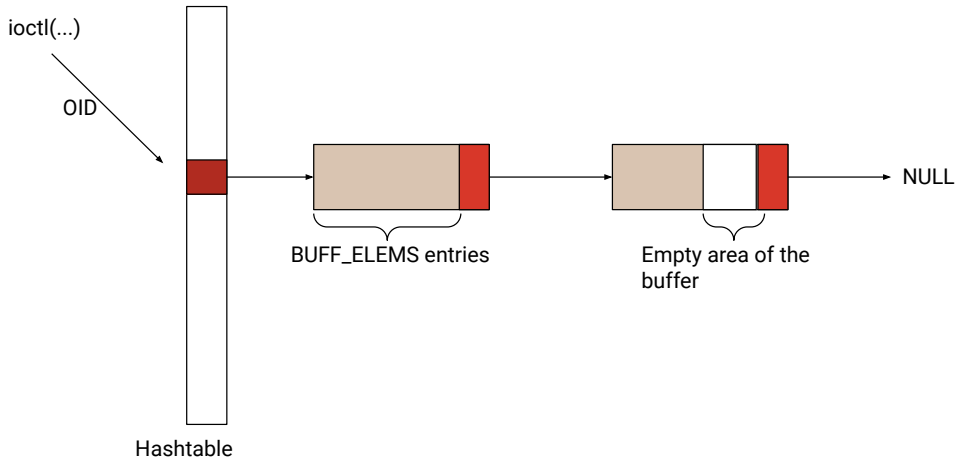

FIGURE 5.6.4: Logging Device Architecture

to be delivered, and the maximum number of addresses of dirty pages that we are currently retrieving. Hence, with this `ioctl(...)` request we can extract kernel data units, each one made of 64 bits, that is the size of a virtual address on x86-64 machines. The actual data structure used for the implementation of the device is shown in Figure 5.6.4. There is a hash table indexed via the OID, where each entry is associated to a linked list. Each element of the list is a buffer where multiple addresses of dirty pages can be recorded. In particular, the number of addresses kept in a single buffer element of the list is a configurable macro, named `BUFF_ELEMS`, which can be selected compiling our LKM on top of the Linux release where the PDES platform needs to be executed. We configured it by setting its value to 256, which appears reasonable since it allows maintaining information on 1MB (256 x 4 KB) of dirty memory via a single kernel buffer allocation operation. Also, each element in the list has metadata not only for pointing to the subsequent element, but also for the identification of the portion

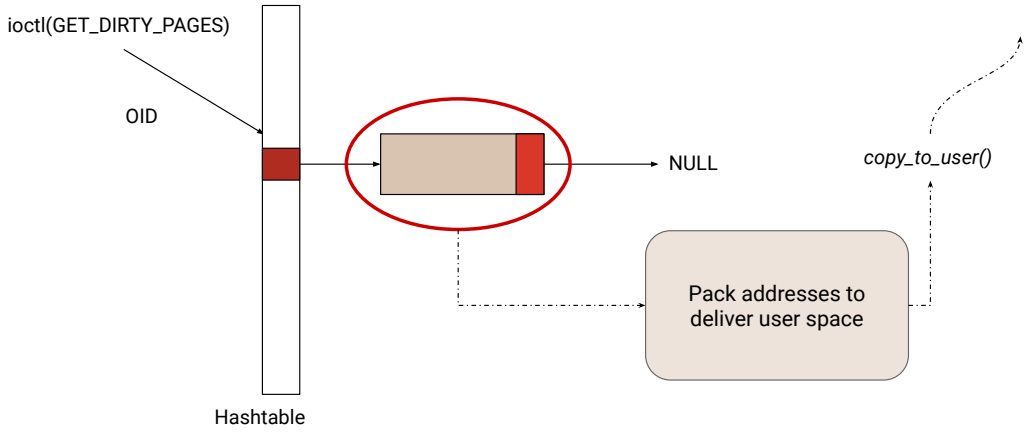

FIGURE 5.6.5: Logging Device Architecture

of buffer area still available for recording other addresses of pages that become dirty.

Additionally, the user-level memory allocator, which, as previously mentioned, uses `mmap(...)` to allocate memory pages, maps the chunks for a specific simulation object (associated with an OID value) to memory addresses selected based on a function of the OID. This approach avoids address conflicts. In particular, we used a fixed-mapping scheme, with the base address computed as a multiple of the OID value. Hence, when the interception of a page address is carried out, the OID of the corresponding simulation object can be immediately identified at kernel level, and so the corresponding hash table entry in order to insert the address of the page that has been dirtied.

As a last note, we implemented the passage of the kernel side logged page addresses via `ioctl(...)` rather than a new system call, since we need to exploit the kernel side `copy_to_user(...)` function for

delivering the information at user level (see Figure 5.6.5), which typically has a cost comparable to the one of other activities required for running `ioctl(...)` at the level of the virtual file system on Linux. Differently, the system calls `track_memory` and `untrack_memory` have been implemented outside the virtual file system to avoid its cost, given that with these system calls we only need to pass two parameters to the kernel through CPU registers, rather than moving data from user to kernel memory buffers or vice versa.

5.7 Experimental Evaluation

5.7.1 Test-bed Environment

We relied on a multi-processor machine equipped with two Intel Xeon Silver 4210R processors, each one hosting 10 cores and 20 hardware-threads, which we simply refer to as CPUs. The machine is equipped with 160GB of RAM, organized in two NUMA nodes, and has 8-way 640KB L1 caches, 16-way 20MB L2 caches and 11-way 27.5MB LLC. Also, we executed our tests with the support of an Ubuntu release[1] running on top of Linux kernel 6.2.

The performed tests focused on both the analysis of the reduction of the overhead with respect to the `mprotect()`-based approach, in terms of the already discussed drawbacks related to the IPI architecture to synchronize the address space across cores, and on the integration of the LKM services for supporting the checkpoint operation in a state-of-the-art speculative PDES platform.

[1] Ubuntu 22.04

5.8 Preliminary Experimental Evaluation

In a first set of tests, we compared the LKM-based solution to the one relying on the `mprotect()` system-call. To perform this comparison, we developed a benchmark application where a number of N threads iterate on the execution of the following set of activities: (1) they call the memory protection service on a mapped memory region consisting of $N \cdot P$ pages to write-protect the pages; (2) they write a single byte in each of the pages; (3) they intercept the address of each page that is dirtied by the write operations performed in point (2). The interception of the memory address of the dirtied page is done in the two compared memory management techniques, either using the `SIGSEGV` signal handler or through the LKM approach described in the previous subsections.

For this test, we scaled up the number of threads N running the benchmark, up to the value 40, corresponding to the maximum number of available CPUs on the target machine. Also, we exploited two different values for the number of pages $N \cdot P$ involved in the operation, being 1 and 512. The value 1 allows us to assess the different techniques when considering the minimum amount of memory actually managed by the operating system for each of the threads, while the value 512 allows us to scale up our analysis to the case when an entire 4-th level page table, namely the PTE, is used on the x86-64 processor running the experiments.

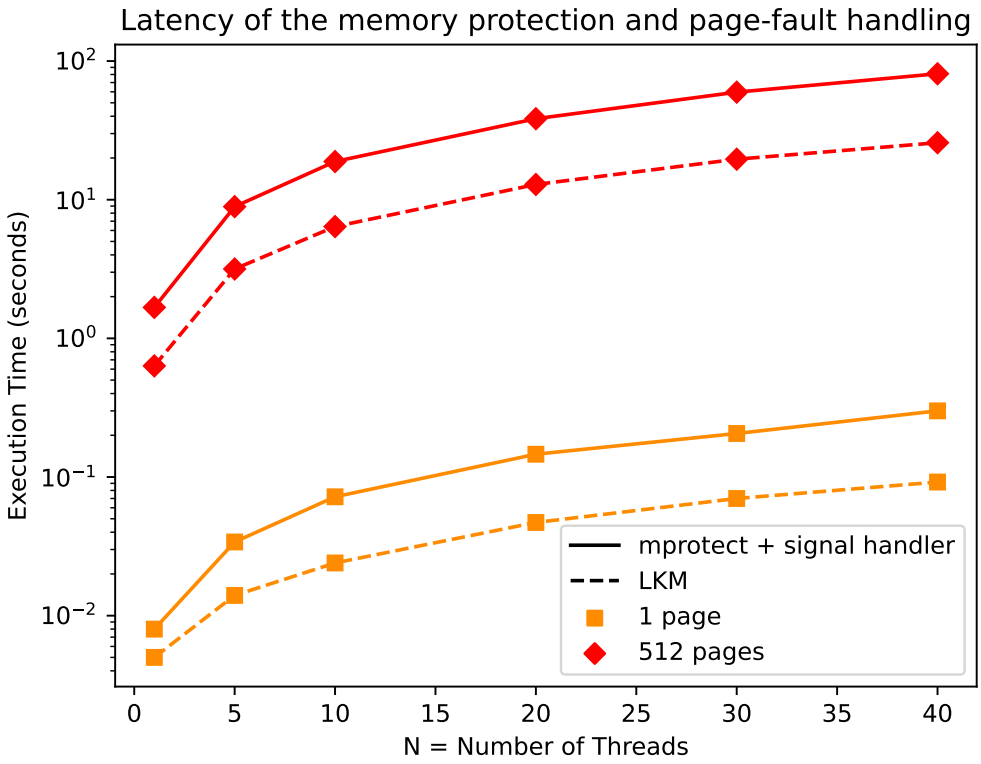

FIGURE 5.8.1: Latency of protection and page-fault handling with different memory sizes (log-scale on the y-axis)

5.8.1 Results

The results of the above-described test are shown in Figure 5.8.1, where we report the latency for running 1000 times, in an iterative manner, the aforementioned execution steps along all the involved threads. From the results, we observe how our new service (labelled as LKM and represented with a dotted line) enables a clear reduction of the latency. Also, the curve for a single page shows how the benefits provided by our solution scale well with the number of used threads. This is also true for 512 pages, but however this curves also include relevant costs

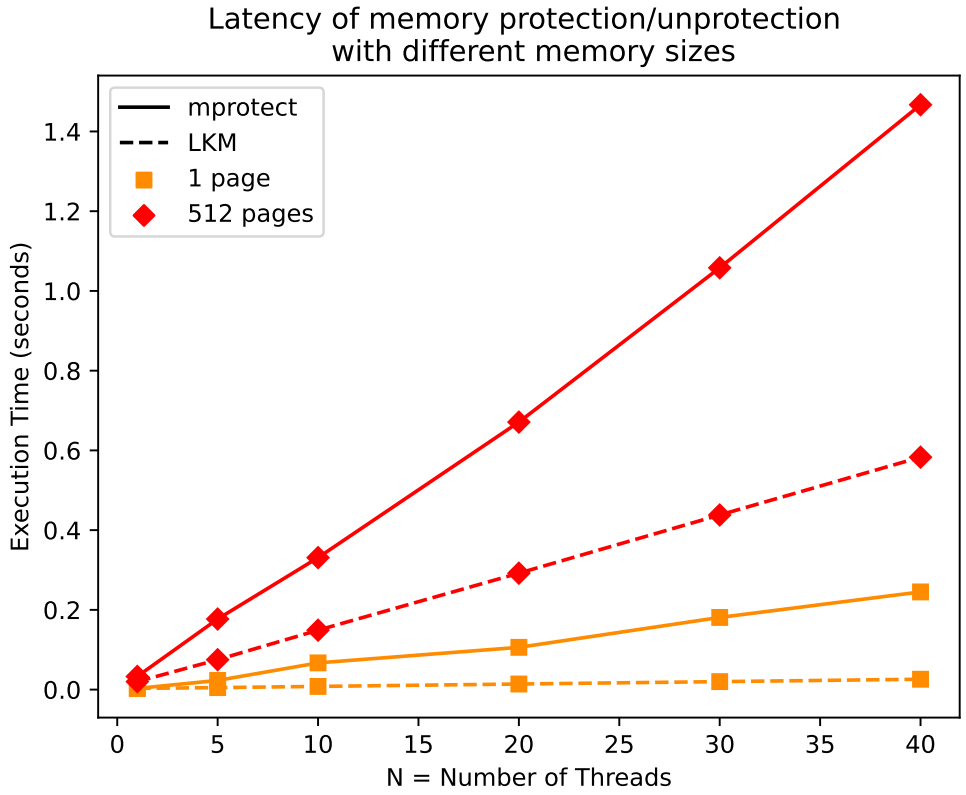

FIGURE 5.8.2: Latency of memory-write protection (and unprotection) via `mprotect(...)` vs our custom LKM syscalls with different memory sizes

related to the tracking of the addresses of the dirtied pages, which tend to become relevant when numerous pages are written at each iteration. To make the analysis more accurate, in particular for what concerns the advantages of our solution in terms of avoidance of the overhead of using IPI and related handlers, we also report in Figure 5.8.2 the latency for running 1000 times only step (1) of the previously listed ones plus the re-opening of the write-access for the pages, still in an iterative manner. In particular, according to these results, we can better observe the comparison of the costs that are simply required

by the different techniques for managing the TLB reset in the hardware architecture. We recall again that `track_memory(...)` and `untrack_memory(...)` only require flushing the TLB of the local CPU, while the classical `mprotect(...)` needs cross CPU interactions through the IPI support. By the data, we observe how the better scalability of our approach is directly perceivable, thanks to better reductions of the test execution time with larger number of threads.

5.9 Benchmark Application

We now discuss the integration of the proposed solution in the state-of-the-art PDES platform USE. The fine-grain resource sharing mechanism among worker threads employed by this platform offers a challenging testing environment for the solution discussed in this thesis. In fact, worker threads might execute simulation objects recently executed by some other thread, forcing to frequently flush the TLBs (up to once for each executed event) in order to align the memory management scheme to the information kept by the page table entries that have been updated by a different thread.
The integration of our new operating system services has been carried out in a fully transparent manner with respect to the layer that implements the simulation model. Hence, the usage of `track_memory(...)` and other services has been embedded exclusively at the level of the simulation engine layer offered by USE. At the same time, the interaction between simulation model specific software and the simulation engine has not been modified at all, and has been left based on the

following classical APIs: (1) the `ScheduleNewEvent(...)` service, callable by the application for injecting (i.e. scheduling) a new event to be processed by whichever simulation object in the model, and (2) the `ProcessEvent(...)` callback offered by the application layer in order to enable the simulation engine to pass control to the simulation model-specific code for processing the event that was previously scheduled destined to some simulation object.
Once again, we have used the Personal Communication System (PCS) model described in Chapter 2, considering cells equipped with 1000 channels each, where the workload of the calls leads to a probability of busy channel of the order of 25%, 50% and 100%. The total number of simulation objects, or LPs, has been set to 1024. Additionally, we varied the checkpoint interval, as the number of events occurring between each checkpoint taken, between two incremental checkpoints from 10 to 80 events. Hence, we explored this parameter value, moving it up to the double of the upper limit suggested by the literature in order to control memory usage thanks to fossil collection. All runs have been executed on 40 CPUs, and we have observed that the model is executed with a rollback frequency of less than 1%.

5.9.1 Results

Figure 5.9.1 reports the average throughput on the y-axis (the higher, the better) achieved by using our approach, labelled ISS-LKM and represented with a dotted line, compared to the `mprotect()`-based one, labelled ISS-mprotect and represented with a solid line, varying

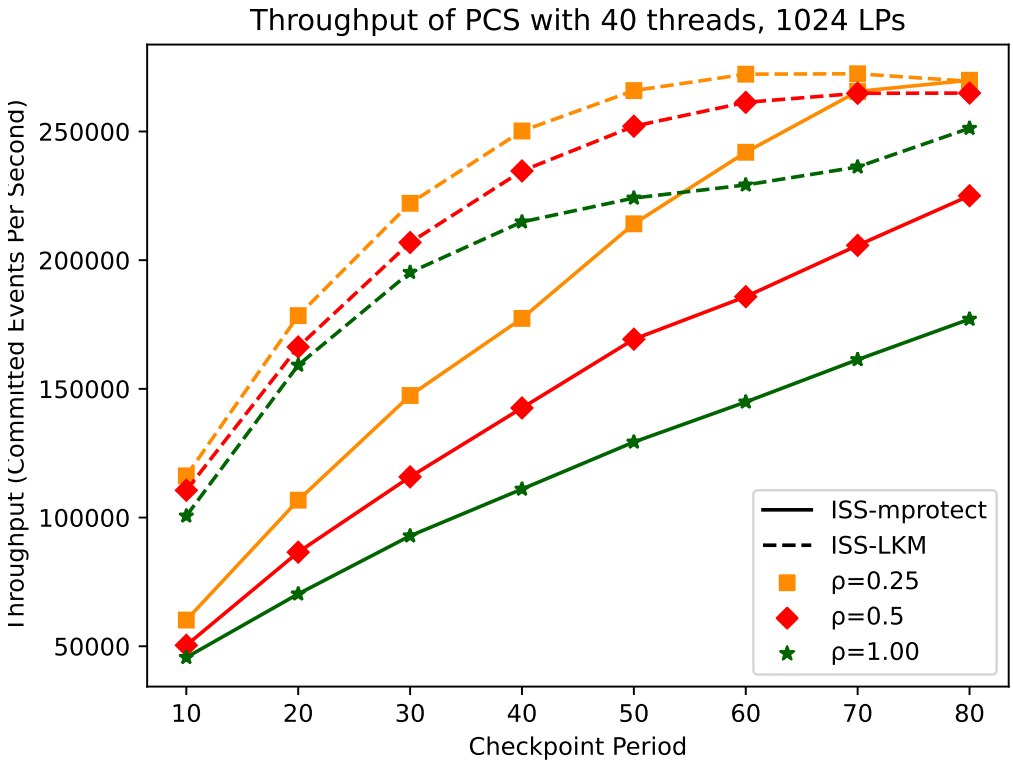

FIGURE 5.9.1: Throughput of Incremental State Saving in PCS via `mprotect(...)` vs LKM facilities on varying checkpoint periods and varying inter-arrival times

the checkpoint interval on the x-axis and with different PCS configurations in terms of utilization. When the checkpoint interval is lower than 50, our approach always provides improved performance with respect to ISS-mprotect, showing that it is able to reduce the overall management costs in a broad range of scenarios by providing from 1.38x speed-up (when $\rho = 0.25$ and checkpoint interval equal to 40) up to 2.27x speed-up (when $\rho = 1.00$ and checkpoint interval equal to 20).

With checkpoint period larger than 50, the benefits introduced by our

approach are slightly reduced with low utilization of cells. This is strictly related to the fact that the lower the utilization is, the lower the number of active channels and, consequently, the lower the wall-clock-time required for executing an event for a simulated cell. Given that the test-bed platform allows a simulation object to be executed by a different worker thread at each event, a lower event granularity increases the frequency of TLB flushes performed by individual CPUs in wall-clock time, which are required by our ISS-LKM. Conversely, since the ISS-mprotect keeps the TLB of each CPU synchronized via IPI, it does not need to pay the cost of a local TLB flush when a simulation object migrates from one worker thread to another one, which is a scenario that likely occurs in a share-everything platform like USE. This makes ISS-LKM being 3% slower than ISS-mprotect in the case of $\rho = 0.25$ and checkpoint interval equal to 80. However, as soon as the utilization factor increases, and thus event granularity, the benefits introduced by our solution emerge again by providing up to 1.6% speed-up (when $\rho = 1.00$ and checkpoint interval equal to 60). Also, the curves show a flatness, particularly for the case of finer grained events, indicating that they are representative in terms of analysis of the variation of performance vs the checkpoint interval up to the point where indirect performance adverse factors, e.g. linked to the memory usage for uncollected fossils, appear.

5.10 Final Remarks

At the beginning of this chapter, we described our approach to support the incremental checkpointing focusing on the exploitation of the memory protection services of operating systems, namely the `mprotect()` system call. We argued that instrumentation-based approaches, that represent the state-of-the-art for the write-tracking approach, hinder performance in case of write-intensive workloads and large state simulation models. This is also due to the multiple-write tracking problem, which occurs when the same memory area is tracked multiple times after being written to more than once, as might happen within a single checkpoint interval consisting of multiple events.
In this scenario, we have shown the effects of exploiting the `mprotect()` system-call to track write operations, and also how optimizing the partitioning of the memory used to host the state of a simulation object can give rise to an effective memory-protection based incremental checkpointing technique. This solution can effectively target models with large-state simulation objects and with events that are write-intensive in specific zones of the object state. We have presented a partitioning scheme that combines buddy-pages at different levels in order to enhance memory awareness of the overall approach, and have reported the results of an experimental assessment of its advantages in terms of memory usage reduction, compared to a classical full checkpointing technique, and reduced number of CPU cycles, compared to a classical incremental technique based on the instrumentation of memory accesses.

However, we have also noticed that the `mprotect()` has some drawbacks related to some kernel-level activities, that causes non-negligible overhead regardless of the optimized management at user level. One drawback is related to the synchronization of kernel level data structures, such as the TLBs, that must be updated, namely flushed, across all cores. Another drawback is related to the frequent user/kernel switch due to the signal handler execution.

As a consequence of the overhead caused by the interaction across cores due to the IPI architecture when updating the page table, we have developed a lightweight operating system service to support incremental checkpointing. The goal was to reduce the above-mentioned overhead for both the memory protection service, and so the across core interaction costs, but also for the management of the write interception of protected memory pages, exploiting the kernel-level interception of the page fault handler.

The new proposed solution, on the one hand, allows for higher scalability since it only acts locally on the Memory Management Unit (MMU) of the CPU that is running the thread in charge of processing events of a specific simulation object, for which incremental checkpointing needs to be setup. On the other hand, it also offers a kernel-side mechanism to register dirtied page addresses and support the incremental checkpointing in a work-deferred manner, and managing a batch of addresses instead of a single address per write-interception, also reducing the interactions between user and kernel level. We showed how the memory protection overhead, measured in terms of execution time, was reduced respect to the `mprotect()`-based solution, and also showed how lighweight the overall approach is in the context of speculative

PDES applications.

Chapter 6

Effective Access to the Committed Global State

In previous chapters, we have described some optimizations tackling a locality-based load-sharing scheme for processing events and supporting in a memory aware and lightweight way the incremental state saving in speculative PDES platforms. We have also described many other optimizations targeting several aspects of PDES systems that have been carried out through the years, still regarding causality violations recovery mechanisms, namely rollback, but also CPU scheduling, load balancing, optimism control, and the very important task in speculative PDES, that is the GVT computation. We have highlighted that, among all of these aspects, one aspect has been dealt with in a limited manner, that is related to the access to the committed portion of the execution trajectory. This is a relevant aspect, since it can be useful for predicate detection purposes, or for producing output data. In Chapter 2, we have discussed how critical is the GVT computation

and how it is fundamental to identify the committed global state of the simulation. In fact, in order to identify a committed portion of the state to inspect the execution trajectory, we have to guarantee that it is not subject to causality violations, and therefore to rollbacks. In order to do so, we must observe a state whose timestamp is past the GVT value.

To recall what has been mentioned in Chapter 3 (Section 3.3), since freezing the speculative processing of events in order to stop the progress and wait that their simulation time is outdated by the new GVT value is not feasible, because it might slow down the speed of simulation processing, solutions based on state-swapping have been proposed [21]. In this solution, the current state of the simulation object is temporarily swapped with a past committed one in order to inspect the simulation trajectory, and then swapped back. However, this scenario refers to the already discussed PDES architecture, in which each worker thread has its own set of simulation objects. This implies that when a thread T_i needs to swap the state of the object o_x that it manages, there will be no other thread capable of touching the same state concurrently. Modern speculative PDES platforms running on top of multi-core shared-memory machines, however, support the full sharing of workload among worker threads, allowing worker threads to process events destined to any simulation object in the system, creating a binding lasting no more than a single or very few events. In this scenario, the realignment to the committed state of an object o_x needs to avoid interference with respect to regular simulation operations carried out on the same object by another worker thread. Similar considerations apply also to simulation engines running on a cluster of shared-memory

machines, where fine-grain workload sharing among worker threads can be adopted within each node in the cluster.
Additionally, as already mentioned in Chapter 3, there are two core aspects that the literature does not take into account, which are our main focus towards the state-swapping to do output collection:

- A thread which is faster than others in swapping the states can then resume event processing of those objects before other threads, possibly leading to over-optimism in their execution. Hence, there is the need for optimizations in the distribution of the state-swapping activities among threads, and the creation of a well confined wall-clock-time window where all threads take care of these activities,
- There is a lack of support for making all threads simultaneously switch to the state swapping activities with close-to-zero delay after a selected wall-clock-time instant, which is useful for scenarios with interactive users who are willing to observe output data based on the committed state trajectory of the simulation at specific time instants. This is clearly interesting also for real-time analysis of the simulation output.

We focus on these two aspects in the scenario of speculative PDES running on top of multi-core shared-memory machines, supporting the full sharing of the workload and filling the gap with the state-of-the-art, but also introducing new techniques to support the access to the latest committed global state with minimal intrusiveness and with minimal latency. We provide the following support by:

- Introducing an operating system service exploiting the Inter Processor Interrupt (IPI) architecture, which allows to promptly notify all the worker threads that a switch to the state-swapping activities is needed, in order to deal with the committed state reconstruction. This service eliminates any delay that would be added by relying on traditional signal mechanisms, in fact using signals a thread can react with a delay related to the first subsequent access to a system call, if any, or an entire time quantum at the end of which an interrupt is generated [1],

- Supporting two different contexts for the execution of a thread: a normal context and a committed reconstruction context. The latter is used for making the thread work on the access to the committed global state of the simulation, and it is installed upon the asynchronous IPI arrival while saving at the same time the normal context, for future resume. Having two separate contexts enables a correct asynchronous interruption, and eventually the resume, of any event processing activity carried out by the thread in normal execution. Hence, the solution proposed in this thesis enables an almost instantaneous move of the computing power to the management of the committed global state of the simulation, while at the same time ensuring the correct resume of any processing activity,

- Introducing a mechanism that enables a balanced distribution of the state-swapping activities among all the worker threads.

[1] On multi-core machines running on Linux, the common setting for this delay can be on the order of 2.5 milliseconds

Hence, the proposed solution aims at providing an almost immediate switch back of the threads to their normal context, in order to reduce the likelihood of under-optimism on simulation objects that are still in the phase of committed state reconstruction, while some threads have already switched back to forward processing other simulation objects in optimistic manner.

As for the IPI support, we notice that the work in [127] presents a solution where the IPI technology is exploited on multi-core machines in order to early abort the execution of no longer consistent events in speculative PDES. In the solution proposed in this thesis, we exploit the IPI technology as well, but with a different objective and according to a different method. In particular, we exploit IPI for building a timeline where a broadcast of interrupts is destined to all the CPUs in order to enable all the worker threads to switch to committed state access and output production. Furthermore, differently from [127], in our architecture the generation of the IPI is demanded to an operating system daemon of the Linux kernel—the SoftIRQ daemon—with is managed via high-resolution-timers and enables real-time support. We also highlight that, even though we have already discussed (see Chapter 2) some possible drawbacks of the IPIs related to the coordination of TLBs across cores, in this case we do not encounter this issue. In fact, in the case of using memory protection services, IPIs are endemic and non-avoidable in the context of CPUs/TLBs coordination, due to the high frequency of new checkpoint interval restarts of the simulation objects. In our case, we leverage the IPI architecture just to notify the threads in a prompt manner to implement the

context-switch (namely, from normal context to committed reconstruction context), with the result of being less impactful on the fine-grain sharing workload scheme, since IPIs are less endemic than the previously described scenario. In fact, output production is less frequent than checkpoint interval restarts.
Our implementation is tailored for Linux operating systems and for x86-64 processors, and we embedded it in the speculative PDES system running on top of shared-memory multi-core machine named Ultimate-Share-Everything (USE) [54], even though the discussed concepts are general.

6.1 System Architecture

Our reference simulation platform architecture adheres to what proposed by the literature [21]. In particular, we consider the scenario where the application layer offers the simulation engine the following two callback functions:

1. `void ProcessEvent(unsigned int me, simtime_t now, int event_type, event_content_t *event_content, unsigned int event_size, void *state)`, which is used to execute an event occurring at the simulation object identified by `me` at a given point along simulation time. The last parameter of this function is a generic pointer `state` which enables the simulation object to reach its state in memory.

2. `bool onGVT(unsigned int me, void *state)`, which is used to pass to the application layer a committed state snapshot, pointed by `state`, of the simulation object whose index is `me`. This function can inspect the state snapshot to determine properties of interest for the modelling scenario and produce output related to the simulation state trajectory. It returns TRUE if, for this simulation object, the execution can be completed.

We note that the callback described in point 1 has been already mentioned in Chapter 5, even if not described in detail.
When all the simulation objects have replied with TRUE via the `onGVT()` callback, the simulation can end.
In order to perform the `onGVT()` callback correctly, the actions that need to be done by a worker thread are the following:

1. The current state of the simulation object `me` is saved.
2. The latest checkpoint of the object `me` with logical time preceding the last computed GVT is restored.
3. The `onGVT()` callback is invoked for this object.

When the `onGVT()` returns control to the simulation platform, the original state of the simulation object, which was saved, gets restored so that the simulation can proceed, if required.
Depending on the end-user choices, it is also possible that, after the restore in point (2) in the above list, the events of the simulation object with timestamp between the restored checkpoint time and the GVT,

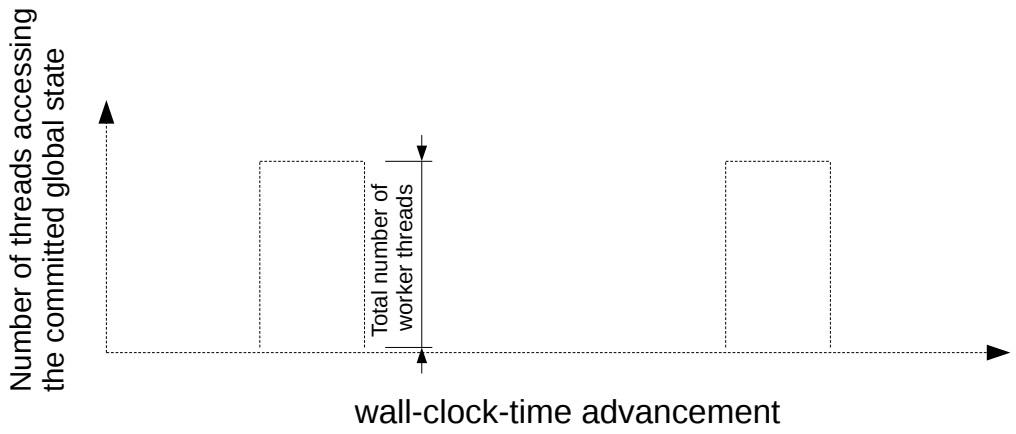

FIGURE 6.1.1: Target timeline of activities along wall-clock-time.

are reprocessed, as in the classical coasting-forward, to realign the simulation object state exactly to the reference GVT.

Since, as pointed out, we are in the scenario where all the worker threads can manage any simulation object at any time instant, we want to reverse all the computing power of the underlying machine (the processing units hosting the worker threads) to the management of the `onGVT()` callbacks at a given wall-clock-time instant, with minimal overhead and delay, and with balanced distribution of the activities across all the threads. In particular, we would like to get a timeline execution of the simulation similar to the one depicted in Figure 6.1.1, where at a given time instant all the threads switch to the execution of the operations related to the access to the committed global state, and then at a subsequent wall-clock-time instant they switch back to the forward execution mode of the simulation. This enables (i) low latency finalization of the access to the committed state snapshot, which beyond helping performance, is also useful in real-time scenarios for the

analysis of the simulation trajectory, and (ii) no risk of having threads executing objects in an overoptimistic manner while other threads are still performing the access to the committed global state, keeping at the same time their targets object(s) busy on this task. This can be helpful for not favouring the increase of rollbacks in the simulation run, thus still favouring performance and limiting the intrusiveness of the committed state access operations on synchronization. This scheme represents a first step towards a further optimization, which we will discuss later, regarding the memory impact and intrusiveness of the context-based approach adopted. We discuss the context-based approach in the next section.

6.2 Execution Contexts

As mentioned above, we envisaged a system architecture based on the possibility to switch the execution context of any thread T_i in a fully asynchronous manner. The asynchronous switch is the only opportunity we have for the thread to react promptly, and to fast –almost immediately– move to the reconstruction of the committed simulation object states related to the latest computed GVT. We denote this execution context as Committed State Reconstruction (CSR), while we denote as NOrmal (NO) the context of classical execution of the simulation main loop.
User Level Technology (ULT) cannot be effective in this scenario, since the thread T_i would require to synchronously reach the point of code where the ULT call passing control to the CSR context is present.

Furthermore, such ULT calls should be also integrated within the application logic in order for the thread to switch to the CSR in a prompt manner even when simply performing a normal forward processing of the simulation events. Clearly, this would reduce the transparency level of the solution with respect to the application code development process.

To tackle all of these limitations, our support for the fully asynchronous switch of the execution context has been based on operating systems facilities. In particular, we have developed an LKM which offers a new system call and a new interrupt handler, which can be used (i) to notify the existence of an additional context CSR for a worker thread, (ii) to switch to this context, and (iii) to return to the original NO context as soon as all the operations for accessing the committed global state have been completed.

More in detail, the system call we added has the signature `int setup_context(void *routine, void *stack)` and is used to notify to the operating system kernel the two core parameters required for setting up the CSR context associated with the thread invoking the system call, namely:

- The memory address of the function that the simulation platform code should use to start the operations in the CSR context.
- The stack area to be used for processing the activities in the CSR context.

```
struct pt_regs csr = {
  r15 = 0x0, ..., rax = 0x0; //default null values
  ip = routine; //rip setup for passing control to the CSR function
  sp = stack; //rsp setup for proper stack management
  ss = 0x2bU; //stack-segment setup for user space return
  cs = 0x33U; //code-segment setup for user space return
  flags = 0x200U; //flags setup for user space return
}
```

LISTING 6.1: Register setup for the CSR context.

We show in Listing 6.1 the actual set up of the CPU snapshot to be installed when the execution in this context needs to be started. The stack pointer register and the instruction pointer register, namely `rsp` and `rip` for x86-64 processors, and the other registers dealing with segmentation and execution mode are set either to common values or to the parameters passed to the system call we added. In the listing, our context is set up based on the Linux kernel `struct pt_regs` data structure, which is used just for recording a CPU state to be restored at some point in time.

The CSR context is associated at kernel level with the specific thread that called `setup_context()`. In particular, we envisage a scenario with worker threads of the simulation platform pinned to the different CPUs of the shared-memory machine. Hence, the data structure keeping the context information is actually placed on per-CPU memory zones. Also, the LKM offers a device driver for simply making the simulation application register itself for the usage of the context management facilities. So, a running thread pinned on a CPU can be immediately recognized as being part of the simulation platform.

In our implementation, the function, whose address corresponds to the value passed with the parameter `routine`, takes no parameter. Hence,

when starting the activities in the CSR context, all the general purpose CPU registers for its activation are simply set to the value zero. At the same time, there is no information at all in relation to floating point and vectorial registers of the x86-64 processor. The reason for this stands in the fact that upon activation of the CSR context, some fresh set up of these registers can be safely performed by the simulation engine or the application logic, for example if these registers are used by the simulation code.

Conversely, when entering CSR and leaving the NO context, floating point and vectorial registers potentially used in the NO context, e.g. used by the `ProcessEvent()` routine implementing the simulation model, need to be saved. To correctly do this operation, the system call `setup_context()` also reserves room for saving both the `struct pt_regs` registers, and the floating point and vectorial registers of the NO context, still on per-CPU memory.

When the CSR context is activated along a worker thread, the NO context is saved, and will not be resumed until the CSR context ends all its activities. To perform a very fast restore of the NO context, we exploited the same handler used for the passage from NO to CSR, this time invoked via a trap called by software. This trap should be invoked by any worker thread at the end of the activities related to the CSR context. Hence, the actual function taking control for the execution in the CSR context at the simulation platform level can be simply structured as

```
void csr_function(void){
    ...     //do the actions required for
            //swapping the simulation objects states
            //and calling the onGVT() function
    asm inline("int disp"::) ;//return control to the normal context
}
```

where `disp` represents the displacement at which the trap handler is installed on the Interrupt Descriptor Table (IDT) managed by the x86-64 processors. In the implementation, we have exploited a not currently used entry of the IDT, that is the spurious interrupt entry. Our kernel side implementation regarding the management of this trap simply restores the NO context of the thread, with no saving the current CSR context. In fact, when the CSR context will need to take control again, this will occur at the next time instant the access to the committed global state of the simulation needs to occur, and all the activities will need to resume simply re-installing the CPU snapshot shown in Listing 6.1.

In order to make all the worker threads currently running the simulation, switch to the CSR context, we exploited, as mentioned, the IPI architecture. In particular, in our LKM we embedded a kernel level function with signature `void switch_to_csr(void)`, which simply sends an inter-processor interrupt to all the processing units on the machine. This is done by exploiting the Linux kernel `send_IPI_all()` API, which enables sending an IPI associated with a specific displacement of the IDT to all the CPUs. When reacting to this interrupt, the handler linked at that entry makes the following actions:

- if the thread hit by the interrupt is the worker thread of the simulation platform that should run on the target CPU, then its context is switched to CSR;
- if the thread of the simulation platform that should run on the target CPU is not currently CPU-scheduled, e.g. because the CPU is currently dedicated to a kernel level daemon, a per-CPU flag is set.

The flag set in the second point is checked by a hook function we installed on the Linux scheduler. As soon as the target thread running the simulation is rescheduled on CPU, the hook checks the flag and if it set, then the switch to the CSR context is performed.
The call to `switch_to_csr()` is done periodically. In particular, we exploited the High-Resolution Timer (HRT) subsystem of Linux in order to post to the timer management subsystem the request for issuing the `switch_to_csr()` after a specific timeout. This does not require any action at the simulation platform level, since the actions related to HRT are already executed by the Linux kernel-level softIRQ daemon. In Figure 6.2.1, we schematize the state machine used for managing the two contexts CSR and NO on a worker thread in our implementation. As a last point, the function `switch_to_csr()` also exploits a bitmap, with one bit for each CPU, indicating whether there is currently a CPU that requires the worker thread to return to the NO context after switching to the CSR context. If the bitmap has at least one bit still set to indicate that a thread did not come back to the NO context, the sending of IPI is skipped, since the last activated access to the committed global state is still being processed.

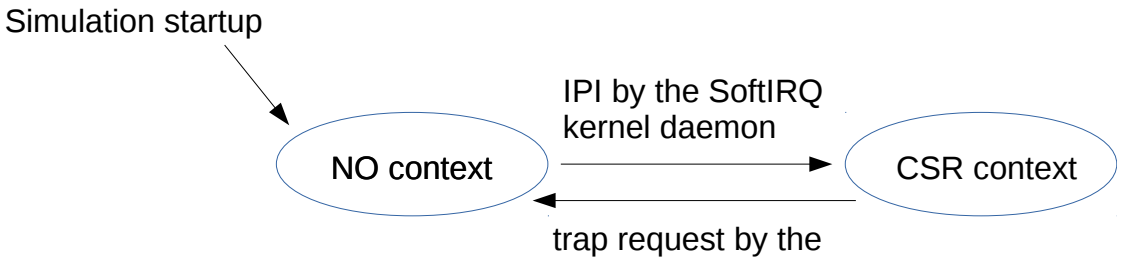

FIGURE 6.2.1: State diagram for the management of contexts

6.3 Memory Safety of Simulation Object States

We discussed in the first chapters how shared-memory computing has become relevant over the years, and how we want to optimize speculative PDES on multi-core shared-memory machines. In this kind of computing systems, the logical processors, namely hyper-threads, share the same address space and can fully share the workload. In the context of PDES, this translates the full sharing of the workload in terms of events to be processed, destined to any simulation object in the system. In our target PDES platform, namely USE, each worker thread simply locks a simulation object for processing its next event currently standing into the pool. The selection of the object to be locked is based on checking into the pool which events with minimal timestamp still need to be processed. Also, each thread attempts to lock a simulation object using a try-lock mechanism, based on the execution of an atomic Compare-and-swap (CAS) machine instruction, which does not lead to real block/wait phases for the thread. Clearly, once the thread has successfully locked an object, it will eventually start processing its next event, thus accessing the object in read/write mode.

In the proposed solution, we do not force a barrier among threads when the committed global state needs to be reconstructed through the state-swapping operation. Hence, it is possible that when the thread T_i switches to the CSR context, where the state-swapping operation is executed, another thread T_j might be still processing some event in NO context, destined to a target simulation object o_x. In fact, the effects of the IPI that triggers the switching of the worker threads to the CSR context can be non-immediate, recall the discussed scenario where the target thread is currently not scheduled on CPU, and will therefore suspend its current activity switching to CSR later along wall-clock-time. In this case, the state of o_x cannot be swapped until we know that thread T_j has not stopped working on that object for normal forward execution in the NO context of the simulation, otherwise we might incur problems when the swapping operation tries to reuse the same memory areas currently used for the state of the simulation object. We refer to this problem as *memory safety*.
At the same time, when the thread switch to the CSR context, we need a mechanism to enable understanding that the object o_x can now be considered for the state-swapping operation.
To address all these problems, we introduced a non-anonymous simulation object locking mechanism. In the proposed solution, a lock associated with a simulation object, whose structure is shown in Figure 6.3.1, is a 64-bit memory location where:

- the most significant bit indicates is the simulation object is currently locked, 0 meaning it is free, 1 meaning that it is locked;

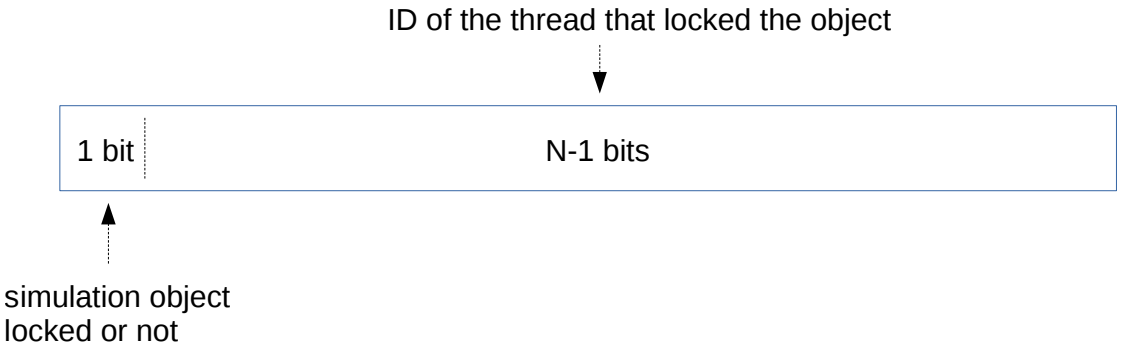

FIGURE 6.3.1: N-bit structure of the per-simulation object lock—N is set to 64 in our x86-64 oriented implementation.

- the remaining 63 bits keep the identifier of the worker thread, if any, that locked the object.

Algorithm 6.3.1 Simulation Object Locking Algorithm

```
1: function TRY_LOCK(lock_t * lock)
2:     lock_t oldValue ← 0
3:     lock_t newValue ← MAKEWORD(1, myTid())
4:     if CAS(oldValue, newValue, lock) then
5:         return TRUE
6:     end if
7:     return FALSE
8: end function
```

The actions done by a thread for trying to lock a simulation object are described in the pseudocode shown in Algorithm 6.3.1. We use the macro MakeWord(val1, val2) to build a 64-bits mask, the most significant of which is set to val1, and the remaining one are set to val2. We use a classical approach where an atomic CAS instruction attempts to put the simulation object as busy, and attempts to write into the lock the identifier of the thread who is performing the locking operation. The condition for the successful execution of this CAS instruction is that the whole set of bits in the lock location are equal

to zero, the 64-th lock hold bit needs therefore to be zero.
Exploiting the non-anonymous locking mechanism, we can detect what thread was in charge of processing events for a given object in the NO context. This will enable that thread to process the object after switching to the CSR context, while preventing any other thread from taking care of the same object.

Algorithm 6.3.2 Usage of the `potential_locked_object` Variable Kept in Thread-local-storage.

```
1: when locking an object with identifier X in the NO context:
2: potential_locked_object ← X
3: if try_lock(&locks[X]) then
4:     process the actual task
5:     unlock(&locks[X])
6: end if
7: potential_locked_object ← NO_OBJECT
```

When attempting to lock a simulation object in the NO context, the thread first writes the identifier of the target object into a thread-local-storage (TLS) variable that we refer to as `potential_locked_object` (see Algorithm 6.3.2). Hence, if the thread successfully acquires the lock, in constant time we can exploit the variable `potential_locked_object` as an index to determine the lock to query on to find the ID of the thread. At the same time, the `potential_locked_object` variable is reset to the `NO_OBJECT` value right after the release of the lock on the object in the NO context. As we will explain in the next section, this will be exploited in our algorithm to distribute the state-swapping activities of the simulation objects among the threads, which we recall being another crucial aspect in shared-memory machines.

6.4 State-swapping Activities Distribution

As stated at the beginning of this chapter, distributing the state-swapping activities among the worker threads has not been covered by the literature, since no approaches targeting systems with full sharing of the workload have been proposed. In particular, the crucial aspect is related to the fact that there might be faster threads in swapping the states, that once switched back to the NO context, start forward-processing events, leading to over-optimism in their execution. In order to solve this problem, we have to guarantee the property that for each simulation object, the swapping activity of its state is executed exactly once, unless we can accept that no output is provided for that object or that duplication of the output can occur.

As shown in the pseudocode in Algorithm 6.4.1, we exploit an integer `object_id` shared among all the threads. Assuming, with no loss of generality, that the simulation object identifiers are integer values starting from zero, `object_id` is initialized to $N - 1$, where N is the total number of simulation objects.

This integer variable is manipulated atomically by all the worker threads, which execute Algorithm 6.4.1 concurrently once they have switched to the CSR context, in particular using the Fetch-and-Add (FAD) machine instruction. FAD returns the original value of the variable while atomically adding, or removing, some units. A thread executes the FAD(`&object_id, -1`) to read the simulation object identifier currently set into the `object_id` variable and to atomically update it, eliminating one unit, in order for other threads to read different object

identifiers in their attempts.
Once a simulation object identifier is acquired by a thread running in the CSR context, the thread attempts to lock the object. In the positive case, the swapping operation targeting that simulation object's state is executed; then the `onGVT()` callback is invoked. Successively, the current object state is restored, and the lock is released. All the activities in the CSR context go ahead until all the simulation object identifiers have been acquired by the threads. However, a corner case exists, since a simulation object identifier X can be acquired by a thread T_i when managing `object_id`, while a lock for normal processing activities on that same simulation object is acquired by a different simulation thread T_j. In this scenario, T_i fails in the lock acquisition, thus not processing the state-swapping activities for that simulation object X. At the same time, the thread T_j did not acquire the object identifier X via the variable `object_id`, since it was delivered to T_i. Hence, T_j does not process the state-swapping of object X as well.
To avoid that object X will not provide its output data when the committed global state needs to be inspected, in the successive part of Algorithm 6.4.1 each thread T_i simply searches if there is one lock that it currently manages and that it has been acquired in NO context, this is done by exploiting the `potential_locked_object` per-thread variable we discussed in the previous section. In this case, the corresponding simulation object state swap can be safely processed by the thread T_i, which will not release the lock, since it will resume processing this object upon switching back to the NO context.
Another important aspect is related to the reliance on a `bitmap` variable, which indicates if a simulation object has been processed in the

Algorithm 6.4.1 Algorithm executed in the CSR context—N is the number of simulation objects, with identifiers in $[0, N-1]$.

SimulationPlatformData:

```
simtime_t reference_GVT = last_computed_GVT()
```

Inputs:

```
int object_id = N-1
bitmap_t bitmap = 0x0
int num_threads = total number of worker threads
```

PerThreadData:

```
int target_object
int potential_locked_object
```

REDO:

```
1: target_object ← FAD(&object_id, -1)
2: if target_object ≥ 0 then
3:     if try_lock(&locks[target_object]) then
4:         if not is_set(&bitmap, target_object) then
5:             swap_state(target_object, reference_GVT)
6:             onGVT(target_object, state_pointers[target_object])
7:             restore_state(target_object)
8:         end if
9:         unlock(&locks[target_object])
10:    end if
11:    goto REDO
12: end if
13: target_object ← potential_locked_object
14: if ( target_object ≠ NO_OBJECT and
15:        locks[target_object] = MakeWord(1,myTid()) ) then
16:    swap_state(target_object, reference_GVT)
17:    onGVT(target_object, state_pointers[target_object])
18:    restore_state(target_object)
19:    set_bit(&bitmap, target_object)
20: end if
21: if FAD(&num_threads, -1) = 1 then
22:    object_id ← N − 1
23:    bitmap ← 0x0
24:    num_threads ← total number of worker threads
25: end if
26: asm inline("int disp"::)          ▷ return control to the normal context
```

CSR context by some thread. In particular, even though the selection of an object identifier through the FAD executed on `object_id` guarantees that different threads always take different object identifiers, it is possible that a thread T_i takes the identifier of an object that is the potential locked one for another thread T_j. If the lock has been actually taken by T_j, this thread can process this object and then return to the NO context, eventually releasing the lock. Making T_j set the bit associated with this object in the `bitmap` variable after having processed it, enables avoiding that T_i will reprocess this same object while executing in the CSR context. The set_bit() API we used to set bits into the `bitmap` variable is assumed to work atomically, for example, using atomic machine instructions.

Any thread T_i simply returns to the NO context via the opposite trap. However, before returning, a re-initialization of the `object_id` and `bitmap` variables needs to be carried out, in order to have their values correctly set for the next access to the CSR context. An additional counter, still managed using FAD, is used to detect the number of threads which already finished their processing activities in the current CSR execution, and to make the last of these threads re-initialize the shared variables, including the counter itself.

The algorithm also uses a reference to the GVT value, exploited in the state-swapping operation of the objects, which, with no loss of generality, is assumed to be published into an appropriate simulation platform variable.

With this algorithm, no two different threads will process the state-swapping activities for an individual simulation object. At the same

time, no simulation object will remain unprocessed since, if lock acquisition in the CSR context fails on a thread, another thread, i.e. the one which locked the object in NO context, will process the state swap on it. Therefore, we guarantee the exactly-once semantic previously declared of each simulation object, at any fully asynchronous switch to the CSR context.
Also, the workload of all the state-swapping operations is fully distributed among all the worker threads, each of which attempts a new state swap after just finishing its last executed one. Therefore, there is no batch of work acquired by a thread in advance, thus reducing the risk that some heavy batch (involving simulation objects for which the cost for the state-swapping operation is larger) can retain some thread in the CSR context for a too long time, while other threads already returned to the NO context. At the same time, the return to the NO context has no barrier, which might potentially induce costs, especially in the scenario of large number of worker threads operating on top of the shared-memory platform.

6.5 Experimental Evaluation

6.5.1 Test-bed Environment

We have already mentioned in Chapter 2 how memory is allocated and managed in the USE platform. As the unique platform-specific facility we had to implement, there is the management of the memory segment that keeps the current state of the simulation object. To recall, USE

relies on DyMeLoR as memory allocator for the states of the simulation objects, and as checkpoint/restore support. Hence, when processing an event at a simulation object, it is possible that a memory allocation/release API (e.g. the malloc() function) is intercepted and processed through DyMeLoR. However, given the IPI-based support for the proposed solution, it is possible that the execution within DyMeLoR gets interrupted. In order to avoid inconsistencies in DyMeLoR metadata, when swapping the state of the simulation object whose event processing has been interrupted, e.g. while just executing DyMeLoR, we perform a flat memory copy of the entire memory segment hosting both its state and the DyMeLoR metadata. This information is then restored upon finishing the access activities to the committed state. Additionally, for all the simulation objects whose event execution is not interrupted, we directly rely on DyMeLoR API in order to checkpoint and restore the current state upon the swapping operation, and to restore the checkpointed state preceding the GVT.

6.6 Benchmark Application

As a benchmark application, we exploited the well-known PCS model, configured such that each cell manages 1000 channels, the average duration of any call is set up to up to 2 minutes, and the call arrival rate has been set to achieve about 40% channel utilization. Handoffs of communicating devices can take place, depending on a mobility model that enables the switch across different cells according to an average delay of 5 minutes. All the stochastic values are sampled by

relying on exponential distributions.
As underlying hardware platform, we used a machine equipped with an Intel i7-12700K with 12 CPU cores (20 Hardware Threads) and 64GB of DDR5 RAM. The installed operating system is Ubuntu 22.04 embedding version 5.19 of the Linux kernel. When running on this machine, the PCS model we used in the experiments shows an average wall-clock-time delay for processing an individual event of the order of 120 microseconds.

6.6.1 Compared Solutions and Metrics

We compared our new, fully asynchronous solution against a synchronous one. The latter relies on the exploitation of a flag, set periodically by a dedicated thread that acts as a metronome, and which is checked synchronously by the threads in order to determine whether the access to the committed global state has been required. If the flag is set, the threads will call the module for the reconstruction and access to the committed state by simply performing a function call. Also, in the synchronous solution, the distribution of the simulation objects to the worker threads is based on a static pre-partitioning, which is what happens when relying on classical solutions in the literature. We still use simulation object locking in this configuration in order to avoid inconsistent access, caused by thread concurrency on the same simulation object, that might anyhow happen because of the intrinsic operations of the USE environment.
For what concerns the metrics used in the comparison, we consider:

- The distance along wall-clock-time for the passage of threads from the normal execution phase to the committed state reconstruction phase. This metric, which we refer to as NO-to-CSR delay (NC-D) enables determining how simultaneous is the actual passage of all the threads to the requested execution task that manages the committed state.
- The delay for processing the committed state access task. This metric, which we denote Committed-State-Access Delay (CSA-D), enables us to determine how effective the used mechanism appears for the timely production of the output data, this is relevant for any scenario where real-time output production is a target.

Finally, we also report data related to the simulation dynamics, in terms of the efficiency of the speculative run. We recall that the efficiency of speculative simulation is the ratio between the total number of committed events and the total number of processed events committed plus rolled back. It provides indications on the incidence of rollback. This helps assess whether the employed mechanism can impact synchronization in some negative manner.
As a last note, for the synchronous solution, which we use as a reference to assess our innovative proposal, the flag indicating to the threads that a consistent state access is requested for the output data production, as mentioned earlier is periodically set by an additional thread, and we pinned that thread to a specific CPU of the underlying machine. This has been done in order to achieve an execution dynamic where the cadence of the access to the committed global state

is regulated via a mechanism that has some similarity with respect to the IPI-send achieved via the SoftIRQ daemon of the Linux kernel, which we exploit in our asynchronous solution. Additionally, all the metrics mentioned above deal which the relative timing of the worker thread activities, hence being independent with respect to the mechanism that is exploited for starting the processing of the committed state access activity. Furthermore, the used machine has a maximum number of 20 hardware threads, while we decided to use no more than 16 worker threads. This has been done just to avoid interference with respect to system level tasks carried out by the operating system. However, this also allowed us to exploit the additional thread used in the synchronous case with no interference on the activities of the worker threads carrying out the simulation execution.

6.7 Results

The PCS model has been run in two different configurations for what concerns the number of used worker threads, namely 8 and 16. In Figure 6.7.1, we report the distribution of NC-D for both the synchronous and the asynchronous mechanisms. From the results, it appears evident that the asynchronous solution we are presenting definitely allows a more timely switch of threads to the procedure that accesses the committed state and supports the output production. In particular, the distribution of NC-D has a peak around 10/30 microseconds for the asynchronous solution, while it shows no peak for the synchronous case and a time span up to the much bigger value of 300 microseconds. This

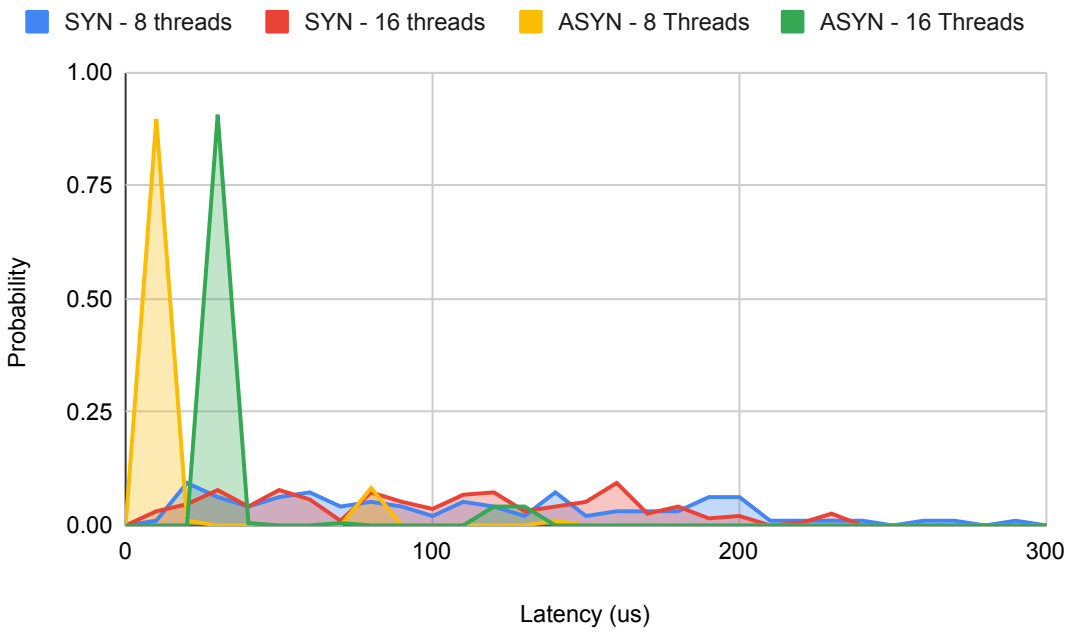

FIGURE 6.7.1: PCS model - NC-D distribution

result shows how our IPI based approach for passing to the committed state access really supports a very rapid variation of the execution flow of all the worker threads involved in the simulation execution.
In Figure 6.7.2, we show the distribution of CSA-D. These data confirm the effectiveness of our asynchronous approach, in particular of the algorithm proposed in Section 6.4 for the distribution of state swap activities among the worker threads. In more details, when running with the asynchronous support, the completion of the activities related to the access to the committed state takes place with a delay bounded by 1 millisecond, while we observe much higher delays, spanning up to 10 milliseconds, for the synchronous mechanism. As noted, this result is highly relevant when considering real-time orientation in the usage of the simulation system.

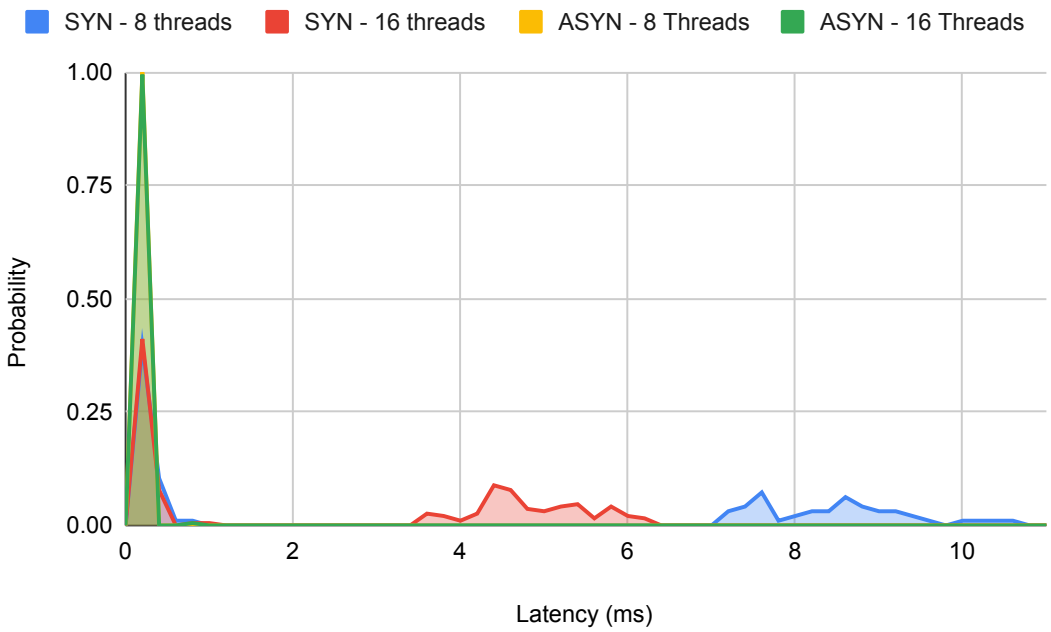

FIGURE 6.7.2: PCS model - CSA-D distribution

So, we have already showed how we actually achieved the goals promised at the beginning of this chapter, but what remains to analyse is the actual intrusiveness of our approach. In fact, the reduced intrusiveness of our asynchronous mechanisms in terms of synchronization and rollback occurrence is supported by the results in Figure 6.7.3. In particular, the variations of the efficiency when the mechanism is active are essentially negligible, independently of the number of used worker threads, and are anyhow non-negative. This is a clear indication of the capability of the mechanism to locate itself in a favourable manner along wall- clock-time, for what concerns the activities of all the threads.

Finally, to provide the reader with additional data related to the dynamics we may expect when employing our mechanism, we report in

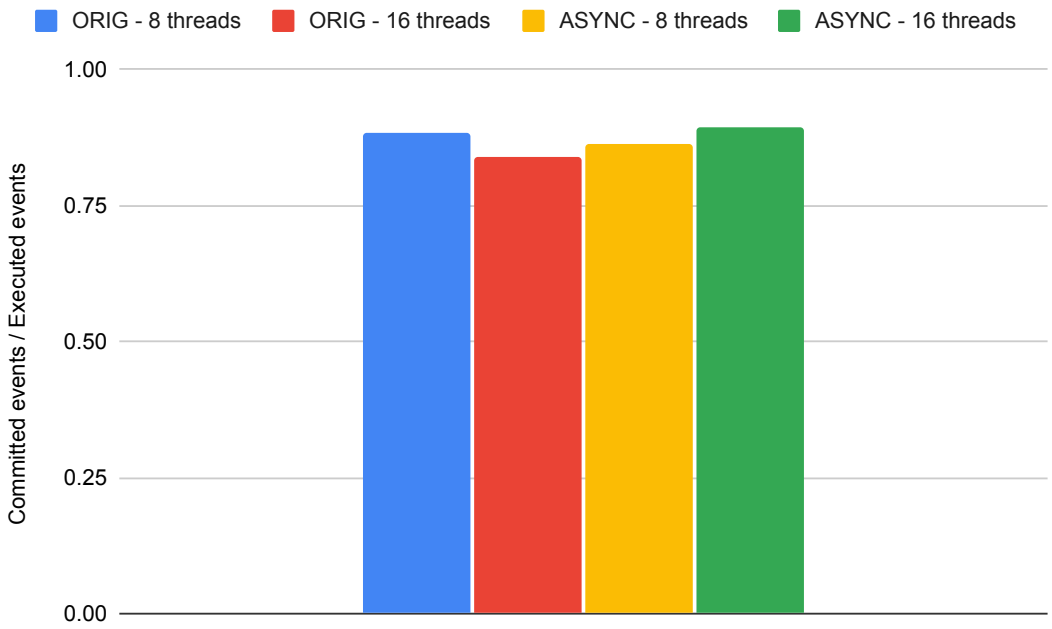

FIGURE 6.7.3: Efficiency w.r.t. original USE

Figure 6.7.4 the relative speed-up that we observed when running the simulations with our committed state access mechanism and when excluding it from USE. For this experiment, we configured the access to the committed state to occur each 5 seconds along wall-clock-time. By the data we observe an essentially negligible overhead of our solution, which can therefore add a feature in the simulation system, the production of output data related to the committed trajectory of the speculative simulation execution, at reduced cost. Clearly, the actual cost can also be determined by the final user choices, since the period after which the output needs to be provided can be selected at simulation startup. Anyhow, our data show what we may expect for a relatively low, although reasonable, setup of this period of time.

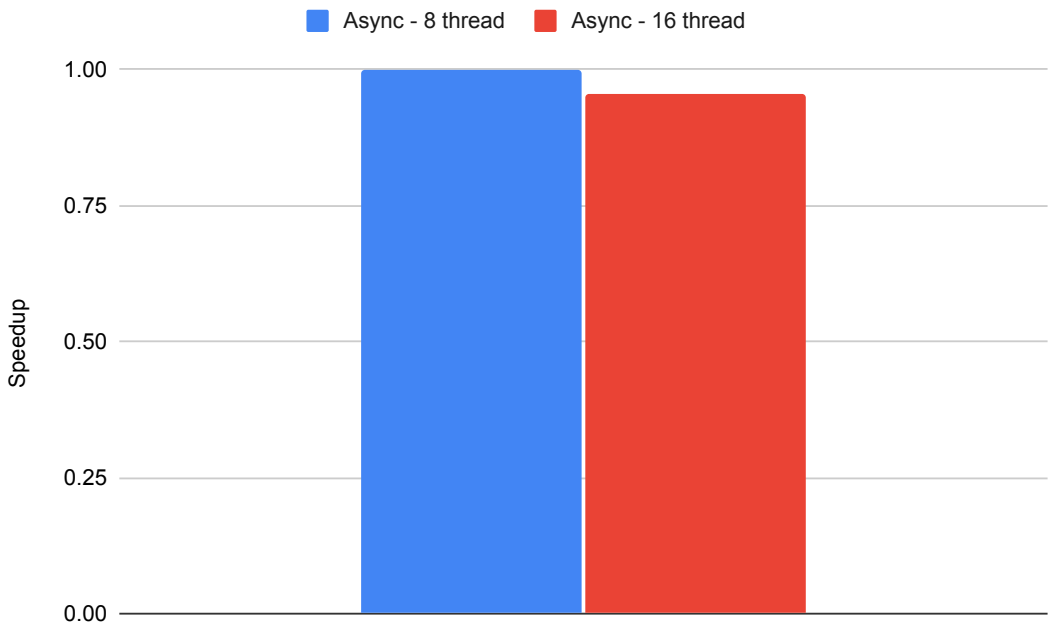

FIGURE 6.7.4: Relative speedup w.r.t. original USE

6.8 Final Remarks

We provided a support for enabling the on-line access to the committed state trajectory in speculative simulations and producing output data that are causally consistent with respect to the executed model. We filled the gap present in the literature for what concerned the state-swapping operation, which we recall was targeted in the scenario of a traditional PDES architecture, that presented a long-term binding between simulation objects and threads, failing to exploit the capabilities given by a shared-memory machine. In fact, the proposed solution is suited for last generation parallel simulation platforms devised for multi-processor/multi-core machines, where the simulation workload is shared among worker threads at very fine grain.

The solution proposed in this thesis, based on innovative operating system services and the IPI technology, shows how it is possible to switch all the worker threads to access the committed state of the simulation simultaneously, thus making them synergic in this activity. We also provided an algorithm for the distribution of the activities among worker threads, which makes them terminate the access to the committed state of the different simulation objects rapidly and in a balanced manner. This point is particularly relevant when considering the evolved fully-shared PDES paradigm, while was unnecessary in earlier approaches due to the simulation objects partitioning among threads. This also enables avoiding negative impacts on execution synchronization (e.g., rollback occurrence) that would be caused in scenarios where only a subset of threads process simulation objects speculatively in forward mode while other threads are still working on committed state access for other simulation objects.

This proposal appears orthogonal to, and combinable with, other literature solutions for the effectiveness of speculative simulation platforms.

Chapter 7

Conclusions

In this thesis, we addressed innovative solutions for speculative Parallel Discrete Event Simulation (PDES) on multi-core shared-memory machines. We motivated the need for these solutions discussing how the hardware has evolved over the years, in particular for what concerns the transition from single-processor systems to multi-processor systems and then multi-core systems. We highlighted the importance of shared-memory machines in the context of the well-known memory wall problem, as well as the challenges imposed by NUMA architectures.

We argued that improving speculative PDES platforms on this kind of machines required a shift on how the workload is distributed in these platforms, and also on novel memory management strategies to fully exploit the capabilities of the underlying hardware. In order to further motivate our statement, we discussed the evolution of PDES systems over the years, from cluster-based approaches, in which the simulation model was distributed across several nodes communicating via message

passing, to an evolved architecture due to the establishment of shared-memory machines. We discussed classical PDES architectures, consisting of a long-term binding distribution, which does not fully exploit the shared-memory machine's capabilities (see Chapter 2), moving then to a finer-grained workload distribution scheme, presenting a short-term binding scheme. While the latter provides higher scalability and better load balancing, we argue that it introduces several new challenges to tackle, such as memory locality/NUMA unawareness, overhead of operating systems' services and critical-path operations intrusiveness. We focused on aspects related to memory management, in order to improve overall memory utilization and reduce the overhead and the intrusiveness when managing shared resources.

This thesis revealed that locality-aware load-sharing strategies, leveraging memory locality principles, are critical for improving overall memory utilization and therefore performance (i.e. throughput). It also showed how NUMA-aware designs help in reducing the costs of cache misses. Therefore, we devised a locality-aware load-sharing scheme in order to favour both spatial and temporal locality when processing events, leveraging a multi-level queue to avoid frequently accessing the shared event pool. We also devised a dynamic window-based scheme to manage the batch processing of events, to reduce the likelihood of increasing the rollback probability, and a simulation objects migration mechanism across NUMA nodes for workload balancing purposes (see Chapter 4).

Regarding classical state recovery schemes for speculative PDES, we

showed the importance of reducing the costs related to memory management for the incremental checkpointing facility, exploiting operating systems' protection services (namely `mprotect()`) and leveraging Linux Kernel Modules (LKM) techniques. First, we focused on reducing the costs associated to the `mprotect()` system-call when used to track memory updates by devising a buddy-page scheme to coalesce contiguous pages. Then, we focused on the reduction of intrusiveness and overhead of the above-mentioned memory protection services, by developing new operating systems services to support the incremental checkpointing in a lightweight manner, leveraging LKM techniques, in order to avoid paying the costs of the kernel level activities pursued by the `mprotect()` system-call. We also enabled a work-deferred way of supporting the incremental checkpointing through a device driver, accordingly queried to retrieve the information regarding memory pages registered by the kernel facilities. We showed how the proposed solution that exploits the `mprotect()` system-call increases the throughput, compared to instrumentation-based approaches and to single page-based approaches (namely, without the buddy-page scheme). We then showed the reduced intrusiveness, as well as the improved performance, of the solution that leverages LKM techniques, compared to the one exploiting the `mprotect()` system-call (see Chapter 5).

Still focusing on the fine-grain sharing of resources, we tackled critical-path operations such as the committed global state identification, in order to inspect simulation trajectory, with close-to-zero delay and with reduced intrusiveness. We highlight that avoiding interference among threads when accessing simulation objects states is a challenging aspect

in this kind of PDES systems. We devised an effective scheme to do state-swapping with close-to-zero delay, leveraging the IPI architecture of Linux-based operating systems, and a workload balancing algorithm to fairly distribute the activities across all threads. We showed the effectiveness of the proposed approach in terms of reduced delay of the overall operation in Chapter 6, comparing the proposed approach to a classic synchronous one.

The advancements presented in this thesis emphasize the need for hardware-aware designs and memory aware strategies in PDES platforms. As the hardware continues to evolve, future research must be aware of it and be up-to-date, ensuring that PDES systems continue to be a viable and effective tool for modelling complex systems. The insights gained from this thesis not only advance the state-of-the-art in PDES systems, filling the gap in the literature regarding fine-grain sharing of resources in such systems, but also make significant contribution to the fields of HPC and parallel simulation.

Bibliography

[1] Abar, S., Theodoropoulos, G. K., Lemarinier, P., and O'Hare, G. M. Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review 24* (2017), 13–33.

[2] Abrams, M., and Richardson, D. S. Implementing a global termination condition and collecting output measures in parallel simulation.

[3] Agbaria, A., and Plank, J. S. Design, implementation, and performance of checkpointing in netsolve. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000* (2000), IEEE, pp. 49–54.

[4] Antonacci, F., Pellegrini, A., and Quaglia, F. Consistent and efficient output-streams management in optimistic simulation platforms. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2013), pp. 315–326.

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

[6] Ayani, R., and Rajaei, H. Parallel simulation based on conservative time windows: a performance study. *Concurrency: Practice and Experience 6*, 2 (1994), 119–142.

[7] Babaoglu, O., and Marzullo, K. Consistent global states of distributed systems: Fundamental concepts and mechanisms. *Distributed Systems 53* (1993).

[8] Banerjee, W. Challenges and applications of emerging nonvolatile memory devices. *Electronics 9*, 6 (2020), 1029.

[9] Boroumand, A., Ghose, S., Kim, Y., Ausavarungnirun, R., Shiu, E., Thakur, R., Kim, D., Kuusela, A., Knies, A., Ranganathan, P., et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018), pp. 316–331.

[10] Büsing-Meneses, V., Montañola-Sales, C., Casanovas-Garcia, J., and Pellegrini, A. Analysis and optimization of a demographic simulator for parallel environments. In *2015 Winter Simulation Conference (WSC)* (2015), IEEE, pp. 3218–3219.

[11] Carnà, S., Ferracci, S., De Santis, E., Pellegrini, A., and Quaglia, F. Hardware-assisted incremental checkpointing in speculative parallel discrete event simulation. In *2019 Winter Simulation Conference (WSC)* (2019), IEEE, pp. 2759–2770.

[12] Carothers, C. D., Bauer, D., and Pearce, S. Ross: A high-performance, low-memory, modular time warp system. *Journal of parallel and distributed computing 62*, 11 (2002), 1648–1669.

[13] Carothers, C. D., and Fujimoto, R. M. Efficient execution of time warp programs on heterogeneous, now platforms. *IEEE Transactions on Parallel and Distributed Systems 11*, 3 (2000), 299–317.

[14] Carothers, C. D., Perumalla, K. S., and Fujimoto, R. M. The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture. In *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 2* (1999), pp. 1624–1633.

[15] Carothers, C. D., Perumalla, K. S., and Fujimoto, R. M. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 9*, 3 (1999), 224–253.

[16] Chandy, K. M., and Misra, J. Distributed simulation: A case study in design and verification of distributed programs.

IEEE Transactions on software engineering, 5 (1979), 440–452.

[17] Chen, L.-l., Lu, Y.-s., Yao, Y.-p., Peng, S.-l., et al. A well-balanced time warp system on multi-core environments. In *2011 IEEE Workshop on Principles of Advanced and Distributed Simulation* (2011), IEEE, pp. 1–9.

[18] Chetlur, M., and Wilsey, P. A. Working set based scheduling in time warp simulations. In *40th Annual Simulation Symposium (ANSS'07)* (2007), IEEE, pp. 221–230.

[19] Cingolani, D., Pellegrini, A., and Quaglia, F. Transparently mixing undo logs and software reversibility for state recovery in optimistic pdes. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 27*, 2 (2017), 1–26.

[20] Cox, D. C. Personal communications-a viewpoint. *IEEE Communications Magazine 28*, 11 (1990), 8–12.

[21] Cucuzzo, D., D'Alessio, S., Quaglia, F., and Romano, P. A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. In *11th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'07)* (2007), IEEE, pp. 227–234.

[22] D'Angelo, G., and Ferretti, S. Adaptive parallel and distributed simulation of complex networks. *Journal of Parallel and Distributed Computing 163* (2022), 30–44.

[23] Dantzig, G. B. Discrete-variable extremum problems. *Operations research 5*, 2 (1957), 266–288.

[24] Davari, B., Dennard, R. H., and Shahidi, G. G. Cmos scaling for high performance and low power-the next ten years. *Proceedings of the IEEE 83*, 4 (1995), 595–606.

[25] Deelman, E., and Szymanski, B. K. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems. In *Proceedings of the twelfth workshop on Parallel and distributed simulation* (1998), pp. 46–53.

[26] Dennard, R. Scaling limits of silicon vlsi technology. In *The Physics and Fabrication of Microstructures and Microdevices: Proceedings of the Winter School Les Houches, France, March 25–April 5, 1986*. Springer, 1986, pp. 352–369.

[27] Denning, P. J. The locality principle. *Commun. ACM 48*, 7 (July 2005), 19–24.

[28] Dickens, P. M., Nicol, D. M., Reynolds Jr, P. F., and Duva, J. M. The impact of adding aggressiveness to a non-aggressive windowing protocol. In *Proceedings of the 25th conference on Winter simulation* (1993), pp. 731–739.

[29] Dickens, P. M., Nicol, D. M., Reynolds Jr, P. F., and Duva, J. M. Analysis of bounded time warp and comparison with yawns. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 6*, 4 (1996), 297–320.

[30] Dickman, T., Gupta, S., and Wilsey, P. A. Event pool structures for pdes on many-core beowulf clusters. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2013), pp. 103–114.

[31] Dolev, S., Hendler, D., and Suissa, A. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing* (2008), pp. 125–134.

[32] Eggers, S. J., and Katz, R. H. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th annual international symposium on Computer architecture* (1989), pp. 2–15.

[33] Eker, A., Williams, B., Chiu, K., and Ponomarev, D. Demand-driven pdes: Exploiting locality in simulation models. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2020), pp. 39–48.

[34] Feldkamp, N., Bergmann, S., and Strassburger, S. Online analysis of simulation data with stream-based data mining. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2017), pp. 241–248.

[35] Ferscha, A., and Tripathi, S. K. *Parallel and distributed simulation of discrete event systems*. Citeseer, 1998.

[36] Fleischmann, J., and Wilsey, P. A. Comparative analysis of periodic state saving techniques in time warp simulators. In *Proceedings of the ninth workshop on Parallel and distributed simulation* (1995), pp. 50–58.

[37] Fu, Y. *Architectural Support for Large-scale Shared Memory Systems*. PhD thesis, Princeton University, 2017.

[38] Fujimoto, R. Time warp on a shared memory multiprocessor. In *ICPP (3)* (1989), pp. 242–249.

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

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

[41] Fujimoto, R. M. Parallel and distributed simulation systems. In *Proceeding of the 2001 Winter Simulation Conference (Cat. No. 01CH37304)* (2001), vol. 1, IEEE, pp. 147–157.

[42] Fujimoto, R. M., and Hybinette, M. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 7*, 4 (1997), 425–446.

[43] Fujimoto, R. M., and Panesar, K. S. Buffer management in shared-memory time warp systems. In *Proceedings of the ninth workshop on Parallel and distributed simulation* (1995), pp. 149–156.

[44] Fujimoto, R. M., Tsai, J.-J., and Gopalakrishnan, G. C. Design and evaluation of the rollback chip: Special purpose hardware for time warp. *IEEE Transactions on Computers 41*, 01 (1992), 68–82.

[45] Gomes, Z. X. F., Unger, B., and Cleary, J. A fast asynchronous gvt algorithm for shared memory multiprocessor architectures. *ACM SIGSIM Simulation Digest 25*, 1 (1995), 203–208.

[46] Gupta, S., and Wilsey, P. A. Lock-free pending event set management in time warp. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2014), pp. 15–26.

[47] Hackenberg, D., Molka, D., and Nagel, W. E. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on microarchitecture* (2009), pp. 413–422.

[48] Hassidim, A. Cache replacement policies for multicore processors. In *ICS* (2010), pp. 501–509.

[49] Hay, J., and Wilsey, P. A. Experiments with hardware-based transactional memory in parallel simulation. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2015), pp. 75–86.

[50] He, Y., Leiserson, C. E., and Leiserson, W. M. The cilkview scalability analyzer. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures* (2010), pp. 145–156.

[51] Hennessy, J. L., and Patterson, D. A. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[52] Heo, J., Yi, S., Cho, Y., Hong, J., and Shin, S. Y. Space-efficient page-level incremental checkpointing. In *Proceedings of the 2005 ACM symposium on Applied computing* (2005), pp. 1558–1562.

[53] Ianni, M., Marotta, R., Cingolani, D., Pellegrini, A., and Quaglia, F. Optimizing simulation on shared-memory platforms: the smart cities case. In *2018 Winter Simulation Conference (WSC)* (2018), IEEE, pp. 1969–1980.

[54] Ianni, M., Marotta, R., Cingolani, D., Pellegrini, A., and Quaglia, F. The ultimate share-everything pdes system. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2018), pp. 73–84.

[55] Ianni, M., Marotta, R., Pellegrini, A., and Quaglia, F. A non-blocking global virtual time algorithm with logarithmic number of memory operations. In *2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (2017), IEEE, pp. 1–8.

[56] Ianni, M., Marotta, R., Pellegrini, A., and Quaglia, F. Towards a fully non-blocking share-everything pdes platform. In *2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (2017), IEEE, pp. 1–8.

[57] Jefferson, D. R. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS) 7*, 3 (1985), 404–425.

[58] Kandukuri, S., and Boyd, S. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE transactions on wireless communications 1*, 1 (2002), 46–55.

[59] Klaiber, A. C., and Levy, H. M. A comparison of message passing and shared memory architectures for data parallel programs. *ACM SIGARCH Computer Architecture News 22*, 2 (1994), 94–105.

[60] Klinkenberg, J., Kozhokanova, A., Terboven, C., Foyer, C., Goglin, B., and Jeannot, E. H2m: exploiting heterogeneous shared memory architectures. *Future Generation Computer Systems 148* (2023), 39–55.

[61] Koo, G., Oh, Y., Ro, W. W., and Annavaram, M. Access pattern-aware cache management for improving data utilization in gpu. In *Proceedings of the 44th annual international symposium on computer architecture* (2017), pp. 307–319.

[62] Kung, H.-T., and Robinson, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS) 6*, 2 (1981), 213–226.

[63] Lawall, J. L., and Muller, G. Efficient incremental checkpointing of java programs. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000* (2000), IEEE, pp. 61–70.

[64] Leiserson, C. E., Thompson, N. C., Emer, J. S., Kuszmaul, B. C., Lampson, B. W., Sanchez, D., and Schardl, T. B. There's plenty of room at the top: What will drive computer performance after moore's law? *Science 368*, 6495 (2020), eaam9744.

[65] Li, K., Naughton, J. F., and Plank, J. S. Low-latency, concurrent checkpointing for parallel programs. *IEEE transactions on Parallel and Distributed Systems 5*, 8 (1994), 874–879.

[66] Lindén, J., and Jonsson, B. A skiplist-based concurrent priority queue with minimal memory contention. In *Principles of Distributed Systems: 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings 17* (2013), Springer, pp. 206–220.

[67] Liu, C., Jia, J., Zhang, Q., and Zhao, L. Lightweight websim rendering framework based on cloud-baking. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2017), pp. 221–229.

[68] Mahmood, I., Haris, M., and Sarjoughian, H. Analyzing emergency evacuation strategies for mass gatherings using crowd simulation and analysis framework: Hajj scenario. In *Proceedings of the 2017 acm sigsim conference on principles of advanced discrete simulation* (2017), pp. 231–240.

[69] Maldonado, W., Marlier, P., Felber, P., Suissa, A., Hendler, D., Fedorova, A., Lawall, J. L., and Muller, G. Scheduling support for transactional memory contention management. *ACM Sigplan Notices 45*, 5 (2010), 79–90.

[70] Maqbool, F., Malik, A. W., Raza Naqvi, S. M., Ahmed, N., D'Angelo, G., and Mahmood, I. Seecssim: A toolkit for parallel and distributed simulations for mobile devices. *Journal of Simulation 15*, 3 (2021), 235–260.

[71] Marathe, M. High performance simulations to support real-time covid19 response. In *Proceedings of the 2020 ACM SIGSIM conference on principles of advanced discrete simulation* (2020), pp. 157–157.

[72] Marotta, R., Ianni, M., Pellegrini, A., and Quaglia, F. A lock-free o (1) event pool and its application to share-everything pdes platforms. In *2016 IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (2016), IEEE, pp. 53–60.

[73] Marotta, R., Ianni, M., Pellegrini, A., and Quaglia, F. 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 (2017), pp. 15–26.

[74] Marotta, R., Montesano, F., Pellegrini, A., and Quaglia, F. Incremental checkpointing of large state simulation models with write-intensive events via memory update correlation on buddy pages. In *2023 IEEE/ACM 27th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (2023), IEEE, pp. 40–47.

[75] Marotta, R., Montesano, F., and Quaglia, F. Effective access to the committed global state in speculative parallel discrete event simulation on multi-core machines. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2023), pp. 107–117.

[76] McKee, S. A. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers* (2004), p. 162.

[77] Mei, X., Zhao, K., Liu, C., and Chu, X. Benchmarking the memory hierarchy of modern gpus. In *Network and Parallel Computing: 11th IFIP WG 10.3 International Conference, NPC 2014, Ilan, Taiwan, September 18-20, 2014. Proceedings 11* (2014), Springer, pp. 144–156.

[78] Mellor-Crummey, J., Whalley, D., and Kennedy, K. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 13th international conference on Supercomputing* (1999), pp. 425–433.

[79] Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS) 9*, 1 (1991), 21–65.

[80] Misra, J. Distributed discrete-event simulation. *ACM Computing Surveys (CSUR) 18*, 1 (1986), 39–65.

[81] Montañola-Sales, C., Gilabert-Navarro, J. F., Casanovas-Garcia, J., Prats, C., López, D., Valls, J., Cardona, P. J., and Vilaplana, C. Modeling tuberculosis in barcelona. a solution to speed-up agent-based simulations. In *2015 Winter Simulation Conference (WSC)* (2015), IEEE, pp. 1295–1306.

[82] Montañola-Sales, C., Onggo, B. S., Casanovas-Garcia, J., Cela-Espín, J. M., and Kaplan-Marcusán, A. Approaching parallel computing to simulating population dynamics in demography. *Parallel computing 59* (2016), 151–170.

[83] Montesano, F. Full-stack revision of memory and data management in pdes on multi-core machines. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing* (2024), pp. 417–420.

[84] Montesano, F. Towards the optimization of memory and data management in speculative pdes on multi-core machines. In *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2024), pp. 63–64.

[85] Montesano, F., Marotta, R., and Quaglia, F. Lightweight operating system services for incremental checkpointing in speculative discrete event simulation on linux platforms. In *2024 IEEE 30th International Conference on Parallel and Distributed Systems (ICPADS)* (2024), IEEE, pp. 745–752.

[86] Montesano, F., Marotta, R., and Quaglia, F. Spatial/temporal locality-based load-sharing in speculative discrete event simulation on multi-core machines. *ACM Trans. Model. Comput. Simul. 35*, 1 (Nov. 2024).

[87] Moore, G. Moore's law. *Electronics Magazine 38*, 8 (1965), 114.

[88] Moore, G. E. Cramming more components onto integrated circuits. *Proceedings of the IEEE 86*, 1 (1998), 82–85.

[89] Nicolae, B., and Cappello, F. Ai-ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing* (2013), pp. 155–166.

[90] Palaniswamy, A. C., and Wilsey, P. A. Adaptive bounded time windows in an optimistically synchronized simulator. In *[1993] Proceedings Third Great Lakes Symposium on VLSI-Design Automation of High Performance VLSI Systems* (1993), IEEE, pp. 114–118.

[91] Palaniswamy, A. C., and Wilsey, P. A. An analytical comparison of periodic checkpointing and incremental state saving. *ACM SIGSIM Simulation Digest 23*, 1 (1993), 127–134.

[92] Palaniswamy, A. C., and Wilsey, P. A. Scheduling time warp processes using adaptive control techniques. In *Proceedings of Winter Simulation Conference* (1994), IEEE, pp. 731–738.

[93] Papamarcos, M. S., and Patel, J. H. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual international symposium on Computer architecture* (1984), pp. 348–354.

[94] Pellegrini, A., and Quaglia, F. The rome optimistic simulator: a tutorial. In *European Conference on Parallel Processing* (2013), Springer, pp. 501–512.

[95] Pellegrini, A., and Quaglia, F. Wait-free global virtual time computation in shared memory timewarp systems. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing* (2014), IEEE, pp. 9–16.

[96] Pellegrini, A., and Quaglia, F. Numa time warp. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2015), pp. 59–70.

[97] Pellegrini, A., and Quaglia, F. Cross-state events: A new approach to parallel discrete event simulation and its speculative runtime support. *Journal of parallel and distributed computing 132* (2019), 48–68.

[98] Pellegrini, A., Vitali, R., and Quaglia, F. Didymelor: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation* (2009), IEEE, pp. 45–53.

[99] Pellegrini, A., Vitali, R., and Quaglia, F. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems 26*, 6 (2014), 1560–1569.

[100] Perumalla, K. S., Park, A. J., and Tipparaju, V. Discrete event execution with one-sided and two-sided gvt algorithms on 216,000 processor cores. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 24*, 3 (2014), 1–25.

[101] Plank, J. S., Beck, M., Kingsley, G., and Li, K. *Libckpt: Transparent checkpointing under unix.* Computer Science Department, 1994.

[102] Plank, J. S., Li, K., and Puening, M. A. Diskless checkpointing. *IEEE Transactions on parallel and Distributed Systems 9*, 10 (1998), 972–986.

[103] Preiss, B. R., Loucks, W. M., and Macintyre, I. D. Effects of the checkpoint interval on time and space in time warp. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 4*, 3 (1994), 223–253.

[104] Quaglia, F. Event history based sparse state saving in time warp. *ACM SIGSIM Simulation Digest 28*, 1 (1998), 72–79.

[105] Quaglia, F. Fast-software-checkpointing in optimistic simulation: Embedding state saving into the event routine instructions. In *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation, PADS '99, Atlanta, GA, USA, May 1-4, 1999* (1999), R. M. Fujimoto and S. J. Turner, Eds., IEEE Computer Society, pp. 118–125.

[106] Quaglia, F. A cost model for selecting checkpoint positions in time warp parallel simulation. *IEEE Transactions on Parallel and Distributed Systems 12*, 4 (2001), 346–362.

[107] Quaglia, F. A low-overhead constant-time lowest-timestamp-first cpu scheduler for high-performance optimistic simulation platforms. *Simulation Modelling Practice and Theory 53* (2015), 103–122.

[108] Quaglia, F., and Cortellessa, V. On the processor scheduling problem in time warp synchronization. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 12*, 3 (2002), 143–175.

[109] Quaglia, F., and Santoro, A. Nonblocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed systems 14*, 6 (2003), 593–610.

[110] Rab, M., Marotta, R., Ianni, M., Pellegrini, A., and Quaglia, F. Numa-aware non-blocking calendar queue.

In *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)* (2020), IEEE, pp. 1–9.

[111] Rao, D. M., and Higiro, J. D. Managing pending events in sequential and parallel simulations using three-tier heap and two-tier ladder queue. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 29*, 2 (2019), 1–28.

[112] Reed, D. A., and Dongarra, J. Exascale computing and big data. *Communications of the ACM 58*, 7 (2015), 56–68.

[113] Riley, G. F., Fujimoto, R. M., and Ammar, M. H. A generic framework for parallelization of network simulations. In *MASCOTS'99. Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (1999), IEEE, pp. 128–135.

[114] Romano, P., Palmieri, R., Quaglia, F., Carvalho, N., and Rodrigues, L. E. T. On speculative replication of transactional systems. *J. Comput. Syst. Sci. 80*, 1 (2014), 257–276.

[115] Rönngren, R., and Ayani, R. Adaptive checkpointing in time warp. *ACM SIGSIM Simulation Digest 24*, 1 (1994), 110–117.

[116] Rönngren, R., Liljenstam, M., Ayani, R., and Montagnat, J. Transparent incremental state saving in time warp

parallel discrete event simulation. *ACM SIGSIM Simulation Digest 26*, 1 (1996), 70–77.

[117] Ronngren, R., Liljenstram, M., Ayani, R., and Montagnat, J. A comparative study of state saving mechanisms for time warp synchronized parallel discrete event simulation. In *Proceedings of the 29th Annual Simulation Symposium* (1996), IEEE, pp. 5–14.

[118] Ross, C. J., Carothers, C. D., Mubarak, M., Ross, R. B., Li, J. K., and Ma, K.-L. Leveraging shared memory in the ross time warp simulator for complex network simulations. In *2018 Winter Simulation Conference (WSC)* (2018), IEEE, pp. 3837–3848.

[119] Ross, C. J., Wolfe, N., Plagge, M., Carothers, C. D., Mubarak, M., and Ross, R. B. Using scientific visualization techniques to visualize parallel network simulations. In *Proceedings of the 2019 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2019), pp. 197–200.

[120] Santoro, A., and Quaglia, F. Transparent optimistic synchronization in the high-level architecture via time-management conversion. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 22*, 4 (2012), 1–26.

[121] Schaller, R. R. Moore's law: past, present and future. *IEEE spectrum 34*, 6 (1997), 52–59.

[122] Schordan, M., Oppelstrup, T., Jefferson, D., and Barnes, P. D. Generation of reversible c++ code for optimistic parallel discrete event simulation. *New Generation Computing 36* (2018), 257–280.

[123] Seelam, S., Fong, L., Tantawi, A., Lewars, J., Divirgilio, J., and Gildea, K. Extreme scale computing: Modeling the impact of system noise in multicore clustered systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (2010), IEEE, pp. 1–12.

[124] Serena, L., D'Angelo, G., and Ferretti, S. Security analysis of distributed ledgers and blockchains through agent-based simulation. *Simulation Modelling Practice and Theory 114* (2022), 102413.

[125] Shun, J. *Shared-memory parallelism can be simple, fast, and scalable*. Morgan & Claypool, 2017.

[126] Siguenza-Torres, A., Cai, W., and Knoll, A. Towards a performance-aware partitioning algorithm for cloud-based microscopic vehicle traffic simulations. In *Proceedings of the 2023 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2023), pp. 44–45.

[127] Silvestri, E., Milia, C., Marotta, R., Pellegrini, A., and Quaglia, F. Exploiting inter-processor-interrupts for virtual-time coordination in speculative parallel discrete event

simulation. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2020), pp. 49–59.

[128] Silvestri, E., Pellegrini, A., Di Sanzo, P., and Quaglia, F. Effective runtime management of tasks and priorities in gnu openmp applications. *IEEE Transactions on Computers 71*, 10 (2021), 2632–2645.

[129] Sköld, S., and Rönngren, R. Event sensitive state saving in time warp parallel discrete event simulations. In *Proceedings of the 28th conference on Winter simulation* (1996), pp. 653–660.

[130] Smith, J. E. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)* (1998), pp. 202–215.

[131] Soliman, H. M., and Elmaghraby, A. S. An analytical model for hybrid checkpointing in time warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems 9*, 10 (1998), 947–951.

[132] Srinivasan, S., and Reynolds Jr, P. F. Non-interfering gvt computation via asynchronous global reductions. In *Proceedings of the 25th conference on Winter simulation* (1993), pp. 740–749.

[133] Srinivasan, S., and Reynolds Jr, P. F. Elastic time. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 2 (1998), 103–139.

[134] Suryanarayanan, V., and Theodoropoulos, G. Synchronised range queries in distributed simulations of multiagent systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 23*, 4 (2013), 1–25.

[135] Sutter, H., et al. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal 30*, 3 (2005), 202–210.

[136] Swenson, B. P., and Riley, G. F. A new approach to zero-copy message passing with reversible memory allocation in multi-core architectures. In *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation* (2012), IEEE, pp. 44–52.

[137] Tang, W. T., Goh, R. S. M., and Thng, I. L.-J. Ladder queue: An o (1) priority queue structure for large-scale discrete event simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 15*, 3 (2005), 175–204.

[138] Taylor, M. B. A landscape of the new dark silicon design regime. *IEEE Micro 33*, 5 (2013), 8–19.

[139] Toccaceli, R., and Quaglia, F. Dymelor: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout. In *2008 22nd Workshop on Principles of Advanced and Distributed Simulation* (2008), IEEE, pp. 163–172.

[140] Veenstra, J. E., and Fowler, R. J. A performance evaluation of optimal hybrid cache coherency protocols. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems* (1992), pp. 149–160.

[141] Vitali, R., Pellegrini, A., and Cerasuolo, G. Cache-aware memory manager for optimistic simulations. In *Fifth International Conference on Simulation Tools and Techniques* (2012).

[142] Vitali, R., Pellegrini, A., and Quaglia, F. Load sharing for optimistic parallel simulations on multi core machines. *ACM SIGMETRICS Performance Evaluation Review 40*, 3 (2012), 2–11.

[143] Vitali, R., Pellegrini, A., and Quaglia, F. Towards symmetric multi-threaded optimistic simulation kernels. In *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation* (2012), IEEE, pp. 211–220.

[144] Vogt, D., Miraglia, A., Portokalidis, G., Bos, H., Tanenbaum, A., and Giuffrida, C. Speculative memory checkpointing. In *Proceedings of the 16th Annual Middleware Conference* (2015), pp. 197–209.

[145] Wang, B., Yu, W., Sun, X.-H., and Wang, X. Dacache: Memory divergence-aware gpu cache management. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (2015), pp. 89–98.

[146] Wang, J., Abu-Ghazaleh, N., and Ponomarev, D. Air: Application-level interference resilience for pdes on multi-core systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 25*, 3 (2015), 1–25.

[147] Wang, J., Jagtap, D., Abu-Ghazaleh, N., and Ponomarev, D. Parallel discrete event simulation for multi-core systems: Analysis and optimization. *IEEE Transactions on Parallel and Distributed Systems 25*, 6 (2013), 1574–1584.

[148] Wenjie, T., Yiping, Y., Tianlin, L., Xiao, S., and Feng, Z. An adaptive persistence and work-stealing combined algorithm for load balancing on parallel discrete event simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 30*, 2 (2020), 1–26.

[149] West, D., and Panesar, K. Automatic incremental state saving. In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation* (1996), pp. 78–85.

[150] Wong, W. A., and Baer, J.-L. Modified lru policies for improving second-level cache behavior. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)* (2000), IEEE, pp. 49–60.

[151] Young, C. H., and Wilsey, P. A. Optimistic fossil collection for time warp simulation. In *Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences* (1996), vol. 1, IEEE, pp. 364–372.

[152] Zehe, D., Viswanathan, V., Cai, W., and Knoll, A. Online data extraction for large-scale agent-based simulations. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2016), pp. 69–78.

[153] Zhang, B., Zhong, J., and Cai, W. A data-driven approach for pedestrian intention prediction in large public places. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (2022), pp. 33–36.