Exploiting Multi-core Systems for Parallel Network Simulation

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Dipl.-Inform.

Georg Johannes Kunz

aus Emmerich, Deutschland

Berichter:

Prof. Dr.-Ing. Klaus Wehrle
Prof., Phd. George Riley

Tag der mündlichen Prüfung: 01.03.2013

Abstract

Discrete event simulation constitutes a fundamental methodology in the design, development, and evaluation process of communication systems. Despite their abstract nature, simulation models often exhibit considerable computational complexity, resulting in extensive simulation runtimes. To counteract the runtime demand of complex simulation models, *parallel* discrete event simulation distributes the workload of a simulation model across multiple processing units. Traditionally, parallel discrete event simulation focused on investigating large scale system models utilizing distributed computing clusters.

In the last decade, however, two developments have fundamentally changed the established state-of-the-art in parallel discrete event simulation. First, multi-core systems have become the de facto standard hardware platform for desktop and server computers. In contrast to distributed computing clusters, multi-core systems provide *different hardware characteristics*, notably shared memory. Second, the focus of interest in the research community shifted from wired to wireless communication systems. Contrary to wired networks the simulated network entities are *tightly coupled* due to detailed modeling of physical layer and wireless channel effects, thereby hindering efficient parallelization. This thesis addresses the challenges resulting from these two developments by designing algorithms and tools to enable and support efficient parallel simulation of tightly coupled systems on multi-core systems. In particular, we make four distinct contributions:

Our first contribution is *parallel expanded event simulation*, a modeling paradigm extending discrete events with durations that span a period in simulated time. The resulting *expanded events* form the basis for a conservative synchronization scheme that considers overlapping expanded events eligible for parallel processing. We furthermore put these concepts into practice by implementing HORIZON, a parallel expanded event simulation framework specifically tailored to multi-core systems.

The durations carried by expanded events provide a deeper insight into event dependencies. Yet, they typically do represent the true dependencies among events. Hence, our second contribution, *probabilistic synchronization*, exploits the globally shared memory space of multi-core systems to observe the behavior of a simulation at runtime and learn accurate dependencies between events. Three different heuristics subsequently exploit the dependency information to guide speculative event execution.

While the previous two contributions focus on speeding up individual simulation runs, our third contribution exploits the massively parallel processing power of GPUs to reduce the runtime demand of entire parameter studies. To this end, we develop a *multi-level parallelism scheme* that bridges the gap between the fundamentally different processing paradigms underlying expanded event simulation and GPUs.

Finally, the performance of any parallelization scheme heavily depends on the structure of a given simulation model. Hence, we specify a *performance analysis methodology* that enables model developers to identify and eliminate performance bottlenecks in simulation models. In combination, our four contributions provide the means for efficient parallel simulation on multi-core systems.

Kurzfassung

Ereignisbasierte Simulation stellt ein grundlegendes Werkzeug im Entwicklungs- und Evaluationsprozess von Kommunikationssystemen dar. Trotz eines abstrakten Modellierungskonzepts weisen akkurate Simulationsmodelle oftmals eine erhebliche Berechnungskomplexität auf, welche wiederum zu beträchtlichen Ausführungszeiten der Simulation führt. *Parallele* ereignisbasierte Simulation wirkt dem Zeitbedarf komplexer Simulationsmodelle entgegenwirken, indem die Arbeitslast eines Simulationsmodells auf mehrere Berechnungseinheiten aufgeteilt wird. Der Fokus paralleler ereignisbasierter Simulation lag dabei traditionell auf der Untersuchung hochverteilter Systeme, z.B. Peer-to-Peer Netzwerke, mittels verteilter Rechnerverbünde.

Zwei Entwicklungen führten jedoch in den vergangenen zehn Jahren zu weitreichenden Veränderungen der Voraussetzungen, die den etablierten Parallelisierungstechniken zu Grunde liegen. Erstens sind Mehrkernprozessoren zum de-facto Standard für Server und Arbeitsplatzcomputer geworden. Im Gegensatz zu verteilten Rechnerverbünden weisen Mehrkernsysteme *andere Hardwareeigenschaften* auf, wie etwa durch Rechenkerne gemeinsam verwendbaren Speicher. Zweitens verlagerte sich der Interessensfokus in der Forschung von drahtgebundenen zu drahtlosen Kommunikationssystemen. Im Unterschied zu drahtgebundenen Netzen besteht aufgrund der detailgetreuen Modellierung des drahtlosen Kanals eine *enge Kopplung* zwischen den Komponenten drahtloser Netze. Diese Arbeit befasst sich mit den Herausforderungen, die aus diesen beiden Entwicklungen erwachsen. Konkret werden in dieser Arbeit vier Lösungsansätze erarbeitet und vorgestellt:

Beim ersten Lösungsansatz handelt es sich um "*parallel expanded event simulation*", ein Modellierungsparadigma welches zeitdiskrete Ereignisse derart erweitert, dass sie eine Periode in simulierter Zeit überspannen. Die daraus resultierenden *erweiterten Ereignisse* bilden die Basis für ein konservatives Parallelisierungsschema, welches eine effiziente Synchronisation eng gekoppelter Systeme ermöglicht.

Unter Ausnutzung der Eigenschaften von Mehrkernsystemen, insbesondere des gemeinsam verwendbaren Speichers, zielt ein *probabilistischer Synchronisationsalgorithmus* darauf ab akkurate Abhängigkeitsrelationen zwischen Ereignissen zur Simulationslaufzeit zu bestimmen. Diese Abhängigkeitsrelationen erlauben folglich eine spekulative parallele Ausführung nicht überlappender erweiterter Ereignisse.

Während die beiden vorigen Ansätze eine Laufzeitreduktion einzelner Simulationsläufe abstreben, befasst sich der dritte Lösungsbeitrag mit der Laufzeitreduktion von Parameterstudien, die mehrere einzelne Simulationsläufe umfassen. Hierzu nutzt ein *Mehrebenen-Parallelisierungsverfahren* die hochparallele Rechenleistung von Grafikprozessoren und überbrückt die grundlegenden architekturellen Unterschiede zwischen Grafik- und Standardprozessoren.

Zu guter Letzt umfasst der vierte Lösungsansatz ein *Leistungsanalyseverfahren*, welches Entwicklern von parallelen Simulationsmodellen ermöglicht Leistungsengpässe zu identifizieren. In ihrer Kombination legen diese vier Lösungsansätze die Grundlage für eine effiziente parallele ereignisorientierte Simulation von Kommunikationssystemen auf Mehrkernsystemen.

Acknowledgments

The genesis of this thesis was significantly influenced by students, colleagues, friends, and family.

I would like to express my sincere gratitude to my both my advisers, Klaus Wehrle and James Gross. Klaus paved the way for my PhD not just by teaching me how to create posters and slides, but also by ensuring that the center of my personal life could remain in Aachen. Moreover, his valuable feedback, advice, and ideas laid the foundations for this thesis. James openly welcomed me into his group, despite my evidently poor knowledge of physical layer modeling and statistical methods. His dedication and the fruitful discussions revolving around my research topic contributed considerably to this thesis. He never grew tired motivating me throughout periods of doubt and uncertainty. I would also like to thank George Riley for reviewing my thesis and for acting as second opponent.

I am very grateful for the opportunity to work with four highly talented students. Their tremendous efforts actively shaped the contributions of this thesis and the corresponding implementations. Remembering long and lively discussions, I would like to thank Mirko Stoffers for his influence on probabilistic synchronization and Horizon, Simon Tenbusch for his enthusiasm regarding optimization problems, Daniel Schemmel for his programming skills and CUDA knowledge, and Marc Peiter for his work on load balancing algorithms.

Over the years, my colleagues have been a constant source of valuable input and personal distraction. I owe Hanno Wirtz a special "thank you" for gracefully enduring my diversions into the non-research related corners of the Internet. Similarly, Raimondas Sasnauskas is not just a motivating role model of a successful researcher to me, but also a close friend. Many thanks to Oscar Puñal and Donald Parruca for taking good care of me at UMIC as well as allowing me to use and benefit from their simulation models. Moreover, I'd like to thank René Hummen for his friendship throughout our time as diploma and PhD students. Stefan Götz showed me how to approach and solve research problems in a structured manner and Tobias Heer was an example in discipline and dedication to me. I'd like to thank Olaf Landsiedel for recommending me as a PhD student to Klaus and for his interest in sharing and discussing new ideas. In addition, a big "thank you" goes to Dirk, Elias, Florian, Hamad, Henrik, Ismet, Janosch, Jó, Marco, Martin, Matteo, Nico, and all members of ComSys and MNP for the scientific adventures and for having a nice time with you.

Finally, my project "PhD" was only made possible by the unconditional support of my family. By believing in me, Rolf, Lili, and Julia provided the motivation for pursuing my PhD. Above all, I am very grateful to Simone for accepting my last minute decision to start a PhD in Aachen, for sacrificing many evenings of leisure time during paper and thesis writing phases, and for pushing me through all periods of doubts throughout my PhD.

Contents

1

Introduction

Discrete Event Simulation (DES) constitutes an essential methodology in the design, development, and evaluation process of communication systems. By employing an abstract model, i. e., a software-based implementation of a system under investigation, discrete event simulation provides a controllable and reproducible evaluation tool at low cost. Following the design principle that simulation models should be as accurate as necessary but as simple as possible [Box76], simulation models focus on the relevant characteristics of a system and abstract from the properties irrelevant to the goal of a particular evaluation study.

Despite this abstraction, simulation models often exhibit complex characteristics which are fundamental for correctly representing the systems being evaluated. For instance, wireless communication systems require detailed modeling of the physical effects influencing a wireless transmission, and simulated peer-to-peer networks need to comprise a large number of nodes to accurately capture the behavior of large scale networks. Modeling such characteristics in detail consequently results in complex simulation models and thus extensive simulation runtimes, which in turn hamper the development and evaluation process.

To counteract this effect, the research community developed parallel simulation techniques to harvest the processing power of multiple processing units [Fuj90a, Liu09, Nic96, Per06b]. These techniques traditionally focus on distributed simulations running on computing clusters composed of independent computers. However, such systems are expensive and not easily accessible to model developers and researchers. Parallel Discrete Event Simulation (PDES) is hence not widely adopted by networking researchers for day-to-day simulation, except for specialized applications developed and deployed by specialists [FPP+03, Per05, PPFR03].

Nevertheless, we observe two developments which fundamentally changed the established situation of parallel discrete event simulation over the last decade:

- The proliferation of multi-core computer systems, and
- the shift of focus from wired to wireless networks in the research community.

Multi-core systems have become the de facto standard hardware platform for commodity desktop and server systems, providing simulation developers and researches with ubiquitously available parallel hardware. Specifically, multi-core systems contain one or more processors which in turn comprise multiple processing cores, each appearing as an individual processor to the operating system. However, while the number of cores in these systems increased constantly, the performance of each individual core remained stable [Sut05]. Hence, software aiming to take full advantage of multi-core systems must make use of parallelization techniques [SL05]. As a result, simulation developers and researchers inevitably have to transition from sequential to parallel simulations. However, due to the architectural differences between distributed computing clusters and multi-core systems, existing parallelization techniques do not efficiently utilize multi-core systems. Consequently, we identify the need for new parallelization techniques which are explicitly tailored to the properties of multi-core systems.

At the same time, the networking research community shifted its focus from wired networks to wireless systems. Specifically, the availability of cheap wireless transmission technologies such as IEEE 802.11 or IEEE 802.15.4 sparked extensive research in domains like Mobile Ad-hoc Networks (MANETs), Wireless Mesh Networks (WMNs), and Wireless Sensor Networks (WSNs) [AWK+11, BLKW08]. In contrast to simulation models of wired networks, accurately modeling wireless networks requires a detailed and hence computationally complex representation of the physical effects influencing wireless transmissions. Moreover, due to the broadcast nature of the wireless channel and the small distances between communicating network nodes, the entities in wireless systems are *tightly coupled*. This tight coupling in turn hinders parallelization of the corresponding simulation models and effectively limits simulation performance [JZT+04, LN02, MB98]. We thus observe the need for new parallelization paradigms that efficiently handle simulation models of tightly coupled systems.

Motivated by these two developments, the goal of this thesis is to develop novel and improve existing parallel simulation techniques that exploit the characteristics of multi-core systems in order to efficiently execute parallel simulation models of tightly coupled systems.

1.1 Problem Analysis

Before being able to design parallelization techniques for multi-core systems, we have to establish an understanding of the problem space. To this end, we first analyze the performance factors influencing a parallel discrete event simulation. We then identify the specific problems and research questions addressed in this thesis.

1.1.1 Performance Factors of Parallel Discrete Event Simulation

The performance of a parallel discrete event simulation is subject to three factors: i) the event synchronization algorithm, ii) the underlying hardware platform, and iii) the simulation model (see Figure 1.1).

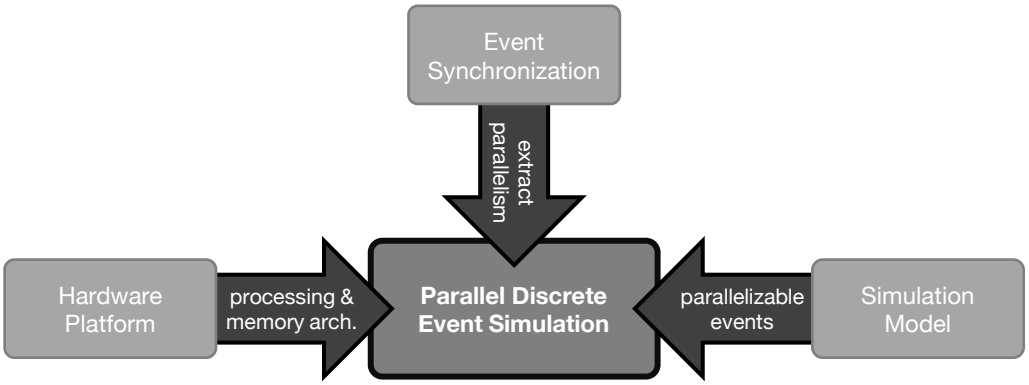


Figure 1.1 Three factors influence the performance of a parallel discrete event simulation: i) the characteristics of the underlying hardware platform, ii) the ability of the event synchronization algorithm to extract parallelizable events from a simulation model, and iii) the number of parallelizable events provided by a model.

Event Synchronization: In discrete event simulation, events represent changes of the state of a simulation model. The order in which these state changes occur during a simulation naturally influences the results of the simulation. Hence, parallel discrete event simulation employs synchronization algorithms that enforce a correct sequential event execution order among *dependent* events. Yet, to maximize the performance of a parallel discrete event simulation, synchronization algorithms try to execute a maximum number of *independent* events in parallel while minimizing the synchronization overhead.

Hardware Platform: A wide range of hardware platforms support parallel execution of simulations, including general purpose platforms such as distributed compute clusters and multi-core systems as well as specialized architectures like vector computers and Graphics Processing Units (GPUs). These platforms exhibit different *characteristics* in terms of memory structure, processing model, and communication overhead. Hence, the design and implementation of a parallel simulation framework has to be tailored to the characteristics of the underlying hardware to maximize performance and efficiency.

Simulation Model: A simulation model primarily has to correctly model the system under investigation. However, in order to *foster* efficient parallel execution, it also has to exhibit a *structure* that supports synchronization algorithms in extracting a maximum number of parallelizable events.

Designing an efficient and well performing parallel discrete event simulation requires consideration of all three performance factors. In the next section, we identify shortcomings of the current state-of-the-art with regard to these performance factors.

1.1.2 Problem Statement

Given the performance factors of parallel discrete event simulation, we now revisit the initially identified two developments that motivate this thesis: i) the proliferation of multi-core systems, and ii) the research focus on wireless networks. We derive three problem statements from these developments and put them into the context of the performance factors.

1.1.2.1 Lack of Event Dependency Information

The shift of focus from wired networks to wireless systems exposes the inability of discrete event simulation to express dependencies between events. Specifically, parallel discrete event simulation employs synchronization algorithms to ensure the correctness of the simulation results while extracting parallelizable events from a simulation model. Ideally, synchronization algorithms allow parallel processing of *all* events which do not depend on each other, i. e., which do not modify the same state of the simulation model. A general purpose simulation framework, however, has no knowledge of the semantics of an event and hence the state changes it performs. Instead, the only accessible information to derive dependencies between events is the *time of occurrence* of events. However, in wireless networks, nodes interact at small timescales due to the broadcast nature of the wireless channel and the small distances between communicating network nodes. As a result, the time of occurrence of events fails to accurately reflect event dependencies in such tightly coupled systems [JZT+04, LN02, MB98].

Still, events in a simulation model are often inherently dependent: In discrete event simulation, events happen instantaneously at a single point in time and have no means of representing time spans. Hence, to model processes that span a period of time, e. g., sending and receiving of packets, discrete event simulation utilizes *two* events: One representing the start of the process and another one indicating the completion of the process. These two events implicitly depend on each other since i) every "start"-event is followed by *exactly one* "completion"-event, and ii) the "completion"-event will *never precede* the "start"-event. The event synchronization algorithm, nevertheless, does not know about this relationship because it is missing from the simulation model and hence handles both events individually. It thereby ignores valuable dependency information, resulting in fewer parallelizable events. We thus conclude that discrete event simulation lacks the ability to express the dependency between two such events, thereby hampering efficient event synchronization.

1.1.2.2 Unsuitable Event Synchronization Algorithms

By employing techniques rooted in traditional distributed simulation, synchronization algorithms do not fully exploit the capabilities and processing power of multi-core systems. In distributed simulation, partitions of a simulation model are distributed across independent computers. Since all partitions of the simulation reside in isolated local memory at the compute nodes, events and synchronization information need to traverse a network interconnecting the nodes. As a result, the communication overhead dominates the total synchronization overhead.

On multi-core systems, however, the performance characteristics of the underlying hardware have changed significantly. A globally shared memory space provides synchronization algorithms with immediate access to the *entire* simulation model, hence enabling a deeper insight into the simulation model and its behavior. Furthermore, fast thread synchronization mechanisms allow for frequent synchronization with low overhead without the need for sending and receiving messages over a network. Moreover, recently emerging specialized multi-core hardware such as GPUs provide massively parallel processing power. However, GPUs implement a

fundamentally different processing model than classic CPUs. This fact needs to be explicitly considered by event synchronization algorithms.

Concluding, we claim that the synchronization algorithms deployed in today's parallel simulation frameworks are deeply rooted in traditional distributed simulation. As a result, they do not make full use of the hardware characteristics of multi-core systems, thereby failing to exploit the available processing power. Instead, multi-core systems provide the substrate for developing novel approaches to event synchronization.

1.1.2.3 Lack of Development Support

Parallel discrete event simulation is not widely employed today since the development of high-performance parallel simulations is complex and challenging: In addition to correctly modeling the system under investigation, model developers also have to structure the simulation model such that it provides a maximum of parallelizable events. This requires a solid understanding of parallel discrete event simulation and event synchronization. Yet, model developers are typically experts in their particular field of research, but not in parallel discrete event simulation. This lack of experience and knowledge prevents a wide adoption of parallel discrete event simulation.

However, multi-core systems constitute the current and future hardware architecture for desktop and server systems. Since these systems demand parallel programming to fully utilize their processing power, model developers and researchers are forced to employ parallel discrete event simulation. The resulting large yet inexperienced user base of parallel discrete event simulation hence requires dedicated tools to ease the development of parallelizable simulation models. In particular, model developers need performance analysis and prediction tools which provide a deep insight into the behavior of parallel simulations to allow for identifying and eliminating performance bottlenecks.

1.1.3 Research Questions

We condense the previous problem analysis in three distinct research questions. It is our goal to provide answers to these questions in the remainder of this thesis.

Question Q1 - How to improve parallel simulation of tightly coupled systems?
We explore modeling and synchronization schemes that explicitly take the properties of tightly coupled systems, such as wireless networks, into account in order to achieve efficient parallel discrete event simulation.

Question Q2 - How to efficiently exploit multi-core systems?
We investigate simulation techniques and parallelization schemes for efficiently exploiting multi-core systems and GPUs for parallel discrete event simulation.

Question Q3 - How to support developers of parallel simulations?
We study means which support model developers in developing efficiently parallelizable simulation models.

1.2 Contributions

We address the aforementioned questions by making four contributions in this thesis:

i) A modeling paradigm and a corresponding parallelization scheme [KLG+10, KLW09, KSGW11] tailored to efficiently handle tightly coupled systems.
ii) A probabilistic event synchronization scheme [KSGW12b, Sto11] which utilizes shared memory to learn dependencies between events at runtime.
iii) A multi-level parallelization scheme [KSGW12a, Sch11] exploiting the massively parallel hardware of general purpose GPUs on multi-core machines.
iv) A performance analysis methodology aiming to support the development process of parallel simulations in order to foster the adoption of parallel simulation among developers and researchers [KTGW11, Ten10].

In the following, we outline the concept of each contribution in more detail.

Contribution C1 - Parallel Expanded Event Simulation

Targeting the lack of dependency information in discrete event simulations, we propose a modeling paradigm that explicitly adds additional timing information to simulation models in order to express dependencies among events. Specifically, the *expanded event simulation* modeling paradigm assigns *durations* in simulated virtual time to events [KLG+10, KLW09]. The resulting *expanded events* represent physical processes that span a period of time. Based on expanded events, we furthermore specify *parallel expanded event simulation*, a centrally coordinated parallel event execution scheme. This scheme utilizes event durations to identify independent events for parallel execution. We moreover prove that our parallelization scheme ensures the correctness of a parallel simulation run.

In addition to this conceptual work, we implement the parallel expanded event simulation principle in a state-of-the-art simulation framework. The resulting extended framework, named Horizon, employs a centralized event synchronization architecture that dynamically distributes events eligible for parallel processing to a pool of worker threads. Since performance is key in parallelization, Horizon builds on event handling and thread synchronization algorithms that are specifically tailored to multi-core systems [KSGW11]. Our evaluation illustrates that Horizon achieves significant performance improvements over traditional distributed parallelization.

Contribution C2 - Probabilistic Event Synchronization

We design a *probabilistic synchronization* [KSGW12b, Sto11] scheme that exploits shared memory on multi-core systems to collect extensive knowledge of event dependencies at runtime. Three heuristics utilize this dependency information to dynamically decide if a given event should be *speculatively* executed in parallel to other events or not.

This scheme combines the advantages of the two fundamental synchronization paradigms underlying parallel discrete event simulation: Conservative and optimistic synchronization. By switching between conservative and optimistic synchronization on a per event basis, the synchronization scheme is able to extract a larger degree of parallelism from tightly coupled system models than

conservative synchronization while limiting the overhead of overly optimistic synchronization. Our evaluation confirms that probabilistic event synchronization outperforms both traditional synchronization schemes as well as previous work on probabilistic synchronization.

Contribution C3 - Multi-level Parallelization on GPUs

We utilize the highly parallel processing power of GPUs to cost-efficiently reduce the runtime demand of parameter studies. Parameter studies comprise multiple independent simulation runs of the same model using different parameterization to find an optimal configuration of the system under investigation. To efficiently combine GPUs with parallel discrete event simulation despite fundamentally different programming and execution models, we employ a *multi-level parallelization* scheme [KSGW12a, Sch11]. The scheme utilizes i) *external parallelism* between the individual simulations of a parameter study to integrate the processing model of GPUs, and ii) *internal parallelism* among independent events within each simulation to hide the latency between CPU- and GPU-memory. By means of a prototype implementation, we demonstrate significant performance improvements and cost savings over traditional CPU-based parameter studies.

Contribution C4 - Performance Prediction and Analysis

We present a *performance analysis methodology* that enables model developers to identify performance bottlenecks in parallel expanded event simulations [KTGW11]. Given an event execution trace of an expanded event simulation, the methodology determines an optimal event-to-CPU mapping that minimizes the simulation runtime under consideration of event dependencies and the utilization of a given number of CPUs. By analyzing the resulting event mapping, developers can pinpoint events that prevent parallel execution and cause low CPU utilization.

Since this mapping problem is NP-complete [Che90, LK78], we model the parallel expanded event simulation paradigm as a Mixed Integer Linear Program (MILP). As a result, we leave the problem of calculating an optimal event schedule to the efficient heuristics and algorithms of modern MILP solvers. Moreover, to further mitigate the complexity problem, we develop performance optimizations and relaxations of the MILP. These optimizations reduce the scheduling problem on average from exponential to polynomial complexity, thereby making it applicable in practice. We demonstrate that the methodology accurately identifies optimal runtime bounds and performance bottlenecks of parallel expanded event simulations.

These contributions were partially developed in cooperation with students in the context of their Bachelor's and Master's theses [Pei12, Sch11, Sto11, Ten10]. Hence, I'd like to explicitly thank these students for their input and their contributions to this thesis.

1.2.1 Interplay of Contributions and Research Questions

The four contributions of this thesis provide answers to the previously identified three research questions. However, there is no simple one-to-one mapping of contributions

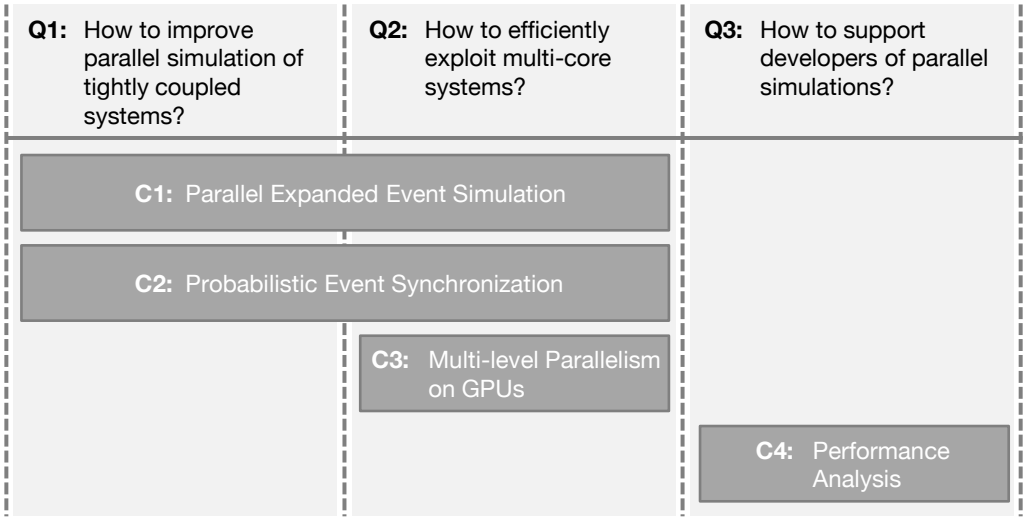


Figure 1.2 Overview showing the interplay of contributions and research questions. Both, parallel expanded event simulation (C1) and Probabilistic synchronization (C2) address multi-core systems as well as efficient simulation of tightly coupled systems. Multi-level parallelization (C3) in turn focuses solely on multi-core systems while our performance analysis methodology (C4) answers the question of development support.

to research questions. Instead, our contributions address the research questions from different perspectives. Figure 1.2 illustrates the interplay of contributions and research questions in this thesis.

Parallel expanded event simulation (C1) addresses both questions Q1 and Q2. While the centralized architecture exploits multi-core computers, expanded events convey dependency information about events in order to handle tightly coupled systems. The probabilistic synchronization scheme (C2) similarly tackles questions Q1 and Q2, yet from a different perspective. It uses shared memory to obtain detailed knowledge of event dependencies which in turn guides speculative event execution to mitigate the performance limitations of tightly coupled systems. Furthermore, multi-level parallelization (C3) utilizes the massively parallel processing power of GPUs which constitute a special-purpose multi-core architecture. The scheme makes heavy use of shared CPU- and GPU-memory, thus contributing an answer to question Q2. Finally, the performance analysis methodology (C4) supports simulation developers in understanding and optimizing the performance of parallel expanded event simulations, thereby giving an answer to question Q3. Throughout this thesis, we compare our contributions to related efforts and illustrate how our approaches deviate from the established state-of-the-art in dedicated related work sections.

1.2.2 Relationship of Contributions

Our four contributions are not isolated but exhibit strong relationships as illustrated in Figure 1.3. Parallel expanded event simulation (C1) constitutes our central contribution. It lays the foundation for our performance analysis methodology (C4) which is designed to accurately reflect the behavior of parallel expanded event simulations. Hence, these two contributions are closely linked.

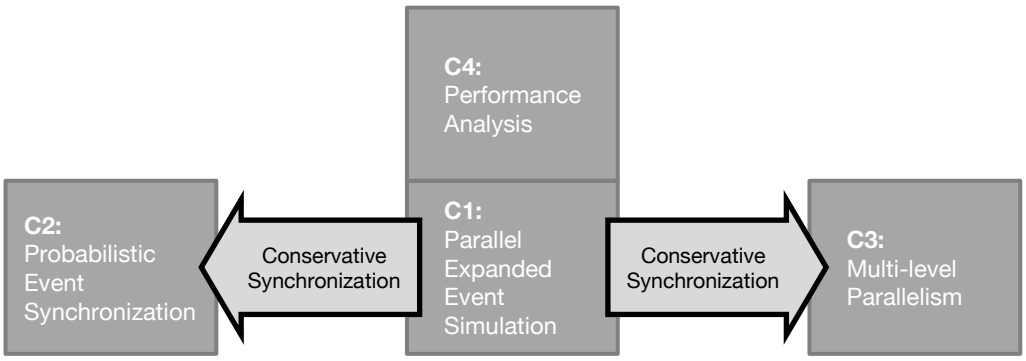


Figure 1.3 Relationship of our four contributions. Parallel expanded event simulation (C1) constitutes the core contribution of this thesis, laying the foundation for the performance analysis methodology (C4). Probabilistic synchronization (C2) and multi-level parallelization (C3) are orthogonal to parallel expanded event simulation and hence independent of expanded events. Yet both contributions can utilize parallel expanded event simulation as conservative synchronization scheme.

Furthermore, parallel expanded event simulation seamlessly integrates with probabilistic event synchronization (C2) and multi-level parallelization (C3): In probabilistic event synchronization, parallel expanded event simulation can serve as conservative synchronization scheme. Similarly, multi-level parallelization can make use of parallel expanded event simulation to conservatively identify independent events according to its internal parallelization scheme.

1.3 Outline

The structure of this thesis is as follows. In Chapter 2, we lay the foundation of the thesis by introducing the established techniques and the current state-of-the-art in parallel discrete event simulation. Chapter 3 introduces the concept of parallel expanded event simulation. In particular, we define a centralized parallelization scheme, prove its correctness, and evaluate its performance. Subsequently, Chapter 4 presents our probabilistic synchronization scheme. Moreover, we show in Chapter 5 how to efficiently incorporate GPUs in parallel discrete event simulation. In addition to those efforts, we address the challenge of creating efficient parallel simulation models in Chapter 6 by designing and evaluating our performance analysis methodology. Finally, Chapter 7 concludes the thesis by summarizing our contributions and discussing future work.

2

Parallel Discrete Event Simulation

Parallel Discrete Event Simulation (PDES) is the subject of extensive research for more than two decades. Consequently, the literature on PDES comprises a rich set of theoretical and practical results [Fuj90a, Kun10, Liu09, Nic96, Per06b]. This chapter introduces the relevant basics that lay the foundation for this thesis. To this end, we first introduce the fundamental concepts of Discrete Event Simulation (DES) and motivate the need for parallelization. Based on this motivation, we then discuss the key challenges of parallelization, classify basic parallelization techniques, and define relevant terms. Finally, we give a brief overview of state-of-the-art parallel simulation frameworks.

2.1 Discrete Event Simulation

The purpose of Discrete Event Simulation (DES) is to model the behavior and the properties of a particular system under investigation, for instance a communication network. Specifically, DES aims at enabling a controllable and repeatable evaluation process at low cost and with short development cycles. It provides the primary means for evaluating complex systems before prototyping and deployment. DES achieves these goals by abstracting from the system under investigation and instead relying on a simplified, purely software based implementation, denoted a *simulation model*.

2.1.1 Nomenclature

This thesis clearly distinguishes between the *simulation framework* and the *simulation model*. The simulation framework provides basic functionality for implementing and executing simulation models. Simulation models in turn define the behavior of a system under investigation. To this end, simulation models maintain a state and

represent state changes by means of *events*. In a communication system, for instance, an event might model that a simulated network node sends a packet and increases its sent-packet-counter state variable.

To add a notion of temporal relation to state changes, a discrete event simulation maintains a global virtual time, denoted *simulated time*, that represents the time within a simulation model. Events change the state of the simulation model at *discrete* points in this simulated time. To this end, every event carries a timestamp, specifying its time of occurrence in simulated time, and an *event handler function*. The event handler function is code of the simulation model that modifies the state of the simulation and potentially creates new events with a timestamp equal or larger than the current simulated time. For example, a send-packet-event might create a new event representing the next sending operation at a future point in simulated time. During a simulation, the *Future Event Set (FES)* holds all events available for execution sorted in increasing timestamp order.

Moreover, we define the terms *event type* and *event instance* as follows: The type of an event is given by the event handler function and possibly additional metadata. An event instance, in turn, is an incarnation of a unique event type. Drawing a comparison with the terminology of object oriented programming, an event type corresponds to a class definition while an event instance is a particular object created from a class definition. In the remainder of this thesis, we use the term *event* as synonym for event instance unless explicitly stated otherwise.

Formally, we define the following terms used throughout the remainder of this thesis:

- $E = \{e_1, \ldots, e_n\}$ is the finite set of all events occurring in a simulation,
- $F \subseteq E$ is the subset of events in the FES,
- $T \subseteq \mathbb{R}^{\geq 0}$ denotes the virtual simulated time, and
- $t : E \to T$ is a function that assigns a timestamp to each event $e \in E$.

Note that F changes over time as events are created and processed. Hence, we can consider F to be a function of time, mapping a given point in simulated time to a subset of E, i. e., $F : T \to \mathcal{P}(E)$. However, we refrain from using this notation in this thesis for the sake of simplicity and to avoid redundancies: Where needed, we explicitly state relevant events in F as $e \in F$ and denote their timestamps as $t(e)$, thereby indicating the simulated time.

2.1.2 Execution Model

The execution model of a discrete event simulation is as follows. An *event scheduler* drives a *simulation run* by continuously i) dequeueing the first event from the FES, ii) setting the simulated time to the timestamp of the event, and iii) executing the event handler function associated with the event.

During execution of the event handler function, the simulated time remains constant, i. e., events happen instantaneously and last zero units in simulated time. Since the FES is sorted with respect to increasing timestamps, simulated time hence advances i) monotonically and ii) in discrete steps. The first property guarantees a *causally correct* execution of the simulation in which the cause (events with an earlier timestamp) always precedes its effect (events with a later timestamp). The

second property decouples simulated time from the continuous notion of real time, called *simulation time*. As a result, simulated time may advance both faster as well as slower than simulation time, depending on the complexity of the model and the processing power of the simulation platform. A simulation run finally ends when either the FES is empty or a predefined termination condition is met.

2.1.3 Modeling Principle

The primary modeling principle underlying simulation is abstraction. In line with the famous quote of George Box that "essentially, all models are wrong, but some are useful" [BD87], the goal of a model developer is to design a model of a system under investigation that is as accurate as necessary but as simple as possible. This means in particular, that a simulation model is designed specifically towards the goals of an evaluation study while leaving out details and aspects not relevant to the study.

For instance, in models of wired networks, the physical effects of a transmission are generally ignored since the error rates on a wire are negligibly. In contrast, modeling the physical effects of the wireless channel in wireless networks is imperative for correctness. A further example is headers in simulated network packets. Typically, simulation model abstract from real world technicalities such as bit fields and byte orders when modeling packet headers. Instead, header fields might just be member variables of a C++ packet class. If, however, the simulation contains real code or interacts with real system, utilizing bit-based headers might be necessary.

Along the same line of reasoning, the system parameters as well as the results and statistics generated by a simulation model depend on the properties being studied. Hence, a simulation model should only provide and vary those parameters which are necessary to collect the statistics of interest with respect to the goals of an evaluation study. In conclusion, the goal of abstraction is to limiting the model complexity in order to obtain understandable, maintainable, and verifiable simulation models.

2.1.4 Comparison with other Evaluation Methodologies

Simulation plays an integral role in the design and evaluation process of communication systems. Nevertheless, its primary principle – abstraction – is both boon and bane as it allows for fast development as well as abstraction beyond correctness. Hence, simulation constitutes only a single step in a thorough evaluation process [KLW10]. In the following, we briefly introduce three evaluation methodologies complementing simulation and put all methodologies in context.

Analytical Modeling: Based purely on mathematical models, analytical modeling typically exhibits a higher level of abstraction than simulation. Hence, an analytical model represents a specific property of a system by means of a formula, for instance the throughput in TCP [MSMO97, PFTK98]. In contrast, simulation models are usually abstract implementations of a specific protocol that is conceptually similar to a real protocol implementation. As a result, an analytical system model provides a clean, fast, and often provable evaluation

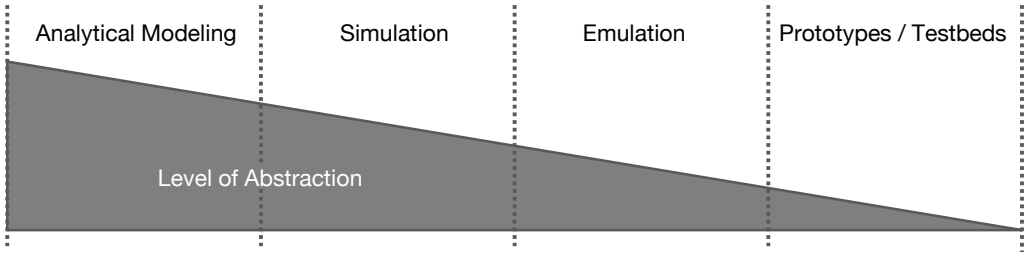


Figure 2.1 Comparison of the four major evaluation methodologies available for the development of communication systems. The level of abstraction from real-world systems is the highest for analytical models and steady decreases over simulation and emulation towards prototypes and testbeds.

methodology, assuming the problem formulation is efficiently solvable. However, devising an accurate analytical model demands extensive mathematical knowledge and typically relies on strong assumptions. Moreover, representing the involved behavior of distributed systems in a purely mathematical fashion often results in complex and computationally intractable models [GG10].

Emulation: In the context of communication systems, emulation refers to a simulation based evaluation process that involves production code as used in real systems [JM05], or even incorporates real systems in the simulation process [Fal99]. Please note that in other domains, emulation refers to highly accurate simulation models, for instance of hardware and/or software systems [Bel05]. Due to the inclusion of real system components in the network simulation, emulation exhibits a significantly lower level of abstraction than simulation, yielding evaluation results closer to reality. At the same time, however, real components increase the complexity of the evaluation setup and limit scalability in comparison to simulation. Moreover, the simulation has to handle headers of network packets on a bit level, or at least provide functionality to convert real packets to and from an abstract representation used by the simulation.

Prototypes and Testbeds: Testbeds built from prototypes or real systems constitute the means for the final evaluation process of a system before deployment in a production environment. Since prototypes comprise the (almost) finalized hard- and/or software, they provide a precise insight into the system behavior, performance, and implementation bugs. Nevertheless, to aid evaluation, analysis, and debugging, testbeds might employ unobtrusive means of abstraction such as a wired side-channel in a wireless sensor network testbed [ASW05]. The most prominent limitations of testbeds, however, are the cost of purchasing and maintaining a testbed, thereby restricting scalability, as well as a lack of repeatability due to uncontrollable side-effects, thus impeding evaluation and debugging [SPBP06].

Figure 2.1 compares discrete event simulation with the evaluation methods discussed above in terms of the level of abstraction from real world systems. We observe that analytical models exhibit the highest level of abstraction which steadily decreases for simulation and emulation towards prototypes and testbeds.

Unfortunately, the evaluation methods are typically incompatible in terms of programming interfaces. Hence, transitioning between different methods often requires re-implementing the system under investigation. In order to eliminate the need for re-implementations, abstraction layers provide a common programming interface across different evaluation and deployment platforms. As part of our overall work, we contributed to the development of an abstraction layer [LKGW09] targeting protocol development and experimentation [KLGW09].

Due to the high level of abstraction, simulation aims at the initial phase of the development process, allowing for quickly estimating the performance of a system. In practice, however, simulation models exhibit considerable computational complexity, thereby hampering the ability for rapid performance estimation. This thesis investigates parallelization techniques to retain this fundamental property of simulation.

2.2 Goals and Challenges of Parallelization

In this section, we first motivate the need for parallel discrete event simulation in greater detail. We then discuss different means of parallelization before formalizing the central challenges of parallel discrete event simulation.

2.2.1 The Need for Parallel Discrete Event Simulation

Simulation models must accurately reflect the properties of a given system to allow for correctly investigating its properties. In general, the complexity of the system under investigation carries over to the respective simulation model. As a result, accurate models of complex systems, such as communication systems, exhibit non-trivial computational resource requirements in terms of computing power and memory demands. These resource requirements in turn hamper thorough evaluation studies, e. g., due to extensive runtimes, and hence hinder efficient development and evaluation of new systems.

Abstraction is the primary means in simulation to limit model complexity. An abstract model focuses only on the relevant aspects of a system and ignores the irrelevant ones [GG10]. However, as abstraction is a trade-off between accuracy and complexity, the level of abstraction is naturally limited if a high degree of accuracy is required [JZTB06]. If a (computationally) complex system property is important for capturing the behavior of the system, the corresponding simulation model cannot entirely abstract from this property but instead has to model this property appropriately. For instance, investigating wireless communication systems relies on detailed radio propagation, interference, and signal coding models. Those operations exhibit a significant computational complexity – often exceeding the computational complexity of all other events in a simulation model by orders of magnitude as illustrated in Figure 2.2. In addition, large scale communication systems involving thousands or even millions of entities, such as the Internet or peer-to-peer networks, show characteristics, such as delay, latency, and churn, that cannot be observed in networks of smaller size. Hence, representative simulation models have to comprise a large number of network nodes to accurately reflect the behavior of the entire

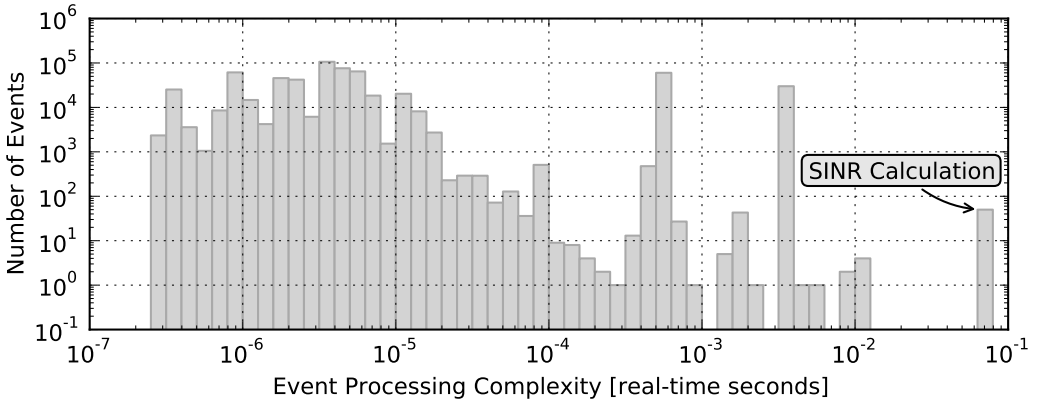


Figure 2.2 Histogram showing the computational complexity and number of occurrences of events in a simulation model of a wireless cellular communication network. The complexity of computing the Signal-to-Interference-plus-Noise Ratio (SINR) for all nodes exceeds the complexity of the other events by orders of magnitude. The model is used to investigate resource allocation algorithms in LTE networks.

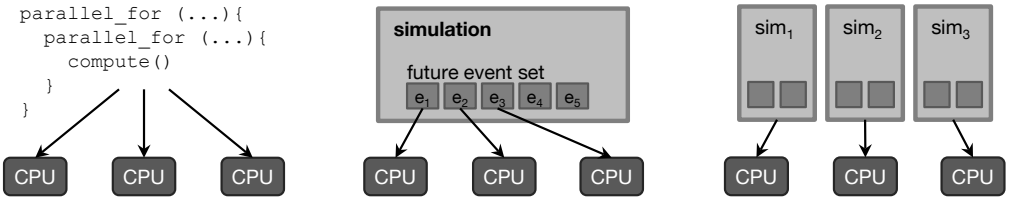
system [CNO99, SW05, LGW06]. Since each simulated network node occupies memory and generates events, such large scale models require considerable memory and computational resources [FPP+03, WGLW12].

The direct effects of those extensive resource requirements on the evaluation process are two-fold: i) extensive simulation runtimes due to computation complexity, thereby hindering an efficient design space exploration, and ii) limited size of the simulation model due to memory constraints, thereby limiting researchers to potentially unrealistic scenarios.

Parallelization constitutes a possible solution for both limitations. First, distributing large simulation models over multiple computing nodes, i. e., individual computers in a cluster, allows for utilizing the combined memory of all machines. Thus, the simulated network can scale to a realistic size. Second, utilizing multiple processing units, e. g., Central Processing Units (CPUs), allows for processing complex events in parallel, thus reducing the overall runtime of the simulation. The importance of parallelization is further amplified by the fact that in recent years the speed of an individual CPU core remained relatively constant. Hence, when faced with a prohibitively complex computation problem it is no viable solution anymore to simply wait until next generation of CPUs are powerful enough to handle the problem. Instead, the number of cores in a CPU increases, forcing developers to apply parallel programming techniques [GK06, SMD+10].

2.2.2 Approaches to Parallelization

We distinguish three orthogonal approaches to parallelizing a discrete event simulation (see Figure 2.3). Since the workload of a simulation is structured in events, parallelization can target i) the code within the event handlers, denoted as intra-event parallelization, or ii) the execution of multiple events in parallel, referred to as inter-event parallelization. Furthermore, iii) inter-simulation parallelization denotes the execution of entire simulation runs in parallel, each using a different parameter set or random seed.



(a) Intra-event parallelization distributes the workload of a single event to CPUs.

(b) Inter-event parallelization distributes individual events to CPUs.

(c) Inter-simulation parallelization distributes entire simulations to CPUs.

Figure 2.3 Comparison of three approaches to parallelization: Intra-event, inter-event, and inter-simulation parallelization.

Intra-event Parallelization: The event handlers of a given simulation can generally contain computationally complex code. In this case, parallelizing the code within such event handlers is a viable approach (see Figure 2.3(a)). Specifically, this approach reduces the overall runtime of a simulation by shortening the processing time of complex events through parallelization, yet events are still executed sequentially.

A variety of threading libraries such as Intel Threading Building Blocks [Rei07], OpenMP [DM98], or Cilk [BJK+95] provide a rich set of features for parallelizing complex event handlers. For instance, these libraries allow for conveniently parallelizing computations involving loops over arrays of data, as often the case when computing the characteristics of the wireless channel. In case loop iterations are independent, the workload can easily be distributed to multiple worker threads, coordinated by the threading library.

Inter-event Parallelization: Instead of executing one parallelized event at a time, inter-event parallelization aims at processing multiple events of the same simulation in parallel (see Figure 2.3(b)). This approach builds upon the observation that events in a simulation model do not necessarily interfere or depend on each other. For example, handling a network packet in the local protocol stack at two separate nodes is inherently independent. Thus, the fundamental idea of inter-event parallelism is to automatically identify such independent events which are suitable for parallel execution.

Inter-simulation Parallelization: In contrast to inter- and intra-event parallelization, inter-simulation parallelization is a nearly trivial approach to parallelization. It relies on the fact that individual simulation runs with different parameters and seeds are inherently independent, thus allowing for parallel execution (see Figure 2.3(c)). Since thorough evaluation studies typically investigate a given parameter space, this parallelization scheme is in fact widely used – particularly on computing clusters.

The efficiency of intra-event parallelism heavily depends on the workload characteristics of the given simulation model. For example, in simulation models of large scale peer-to-peer networks, the complexity of each event is negligible, but the multitude of events creates a large computational load. Thus, intra-event parallelism is not suitable for such models, hence lacking general applicability. Moreover, if only a

subset of events in a simulation model possess reasonable complexity, the overall reduction in runtime is limited by the fraction of runtime these events contribute to the total simulation runtime. This observation follows directly from Amdahl's Law [Amd67].

Most importantly, intra-event parallelism requires solid knowledge and experience to (re-)structure a given event handler, i. e., algorithm, such that it supports efficient parallel execution. Additionally, every event handler needs to be modified manually by the model developer. Instead, in inter-event parallelization the burden is on the simulation framework to identify parallelizable events while the modeler can focus on the semantics of the model. Consequently, inter-event parallelization is model agnostic.

Finally, inter-simulation parallelization exhibits two drawbacks: First, it does not reduce the runtime of individual simulation runs, thereby hindering rapid prototyping. Second, to fully utilize a multi-core computer, we need to run one simulation instance per CPU. However, the memory requirements of large scale simulation models might prevent running a sufficiently large number of instances on a multi-core computer.

In conclusion, intra-event parallelization lacks general applicability while inter-simulation parallelization does not improve the runtime of individual simulation runs. Because of these limitations, inter-event parallelization constitutes the primary approach to parallel discrete event simulation. As a result, we focus entirely on inter-event parallelism in the remainder of this thesis.

2.2.3 Challenges of Parallel Event Execution

Recall from Section 2.1.2 that a sequential discrete event simulation is causally correct if all events execute in monotonically increasing timestamp order. It is crucial that a parallel simulation fulfills this correctness requirement as well to retain its applicability as a deterministic evaluation tool. At the same time, parallel discrete event simulation aims for increasing simulation performance by executing events in parallel. Hence, the key challenge of parallel discrete event simulation is maintaining the correctness of the simulation while maximizing simulation performance.

To illustrate that this challenge is indeed non-trivial, consider a naïve approach to parallelization. The simple approach iteratively assigns all events $e \in F$ in the FES to the available processing units. We consider a simple example shown in Figure 2.4 to illustrate that this approach however cannot guarantee causal correctness, i. e., a monotonically increasing simulated time:

Assume a simulation model which is split into multiple components c_1 and c_2, in accordance with common software engineering practice. Let furthermore each component maintain an own local virtual time t_{c_1} and t_{c_2}. Assume moreover two events $e_1, e_2 \in F \subseteq E$ with $t(e_1) < t(e_2)$. According to our simple parallelization approach, we execute e_1 and e_2 in parallel on two separate CPUs. Thus, the local time of c_i advances to $t(e_i), i \in \{1, 2\}$ according to the general execution model of discrete event simulation. In general, the event handler of e_1 at c_1 can create a new event e_3 with $t(e_1) < t(e_3) < t(e_2)$, scheduled for execution at component c_2. However,

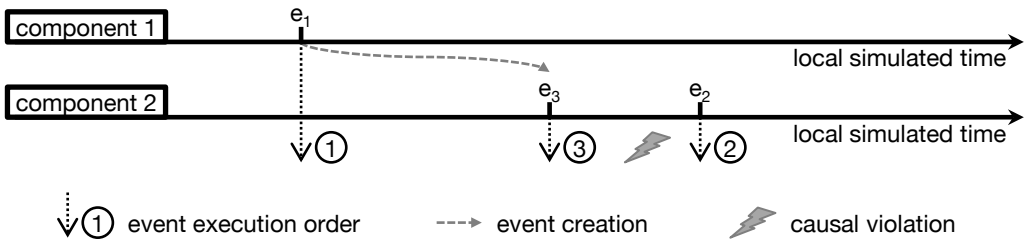


Figure 2.4 Naïve, i. e., unsynchronized, parallel event execution cannot guarantee causal correctness: Processing e_2 before e_3, which is created by e_1 and hence arrives later than e_2 in the FES, results in a causal violation because $t(e_3) < t(e_2)$.

executing e_2 at c_2 already set the local time of c_2 to $t(e_2)$ which is later than $t(e_3)$. Consequently, executing e_3 sets the local time *back* to $t(e_3)$, thereby resulting in a *causal violation*, i. e., out-of-order event execution.

We conclude from this example that parallelization needs to synchronize and coordinate parallel event execution to achieve causally correct simulation execution. To solve this challenge, parallel discrete event simulation employs specific modeling concepts and synchronization algorithms, which we discuss in the next section.

2.3 Concepts of Parallel Discrete Event Simulation

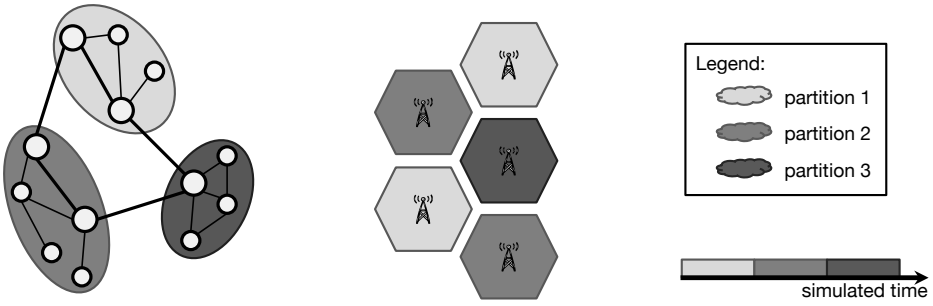
This section introduces the fundamental concepts underlying parallel discrete event simulation. Specifically, we first introduce schemes for partitioning a simulation model, followed by a formal definition of causal correctness. Finally, we briefly discuss two classes of synchronization schemes and distinguish three different event execution environments.

2.3.1 Workload Partitioning

A parallel simulation needs to partition the workload of a simulation run to make use of multiple processing units. In the following, we briefly sketch three different partitioning schemes.

Space-parallel Partitioning: The de facto standard partitioning scheme in PDES is *space-parallel partitioning* [Fuj90a]. This scheme splits a simulation model into groups of simulated entities which in turn form a partition. In case of network simulations, these entities correspond to network nodes and their subcomponents such as protocol stacks, network cards, routing tables, etc. Since simulation models typically exhibit a modular structure corresponding to those entities, this scheme is generally applicable.

In a special case of space-parallel partitioning, a simulation model is distributed across different simulation frameworks, called *federates*. The purpose of this approach is to couple specialized simulators to take advantage of their particular features and simulation models. For instance, in order to evaluate the hardware design of a network router, a network simulation can simulate the



(a) Space-parallel partitioned model of a wired network. Clusters of local nodes map to partitions.

(b) Channel-parallel partitioned model of a cellular network. Non-neighboring cells map to partitions.

(c) Time-parallel partitioned simulation run. Different periods of the simulated time map to partitions.

Figure 2.5 Overview of space-, channel-, and time-parallel partitioning. In all three examples, the workload is distributed over three partitions.

surrounding network and its traffic while a hardware simulator models the internals of the router. However, despite a different terminology (federates instead of partitions), the parallelization challenges and corresponding solutions are equivalent. Hence, we do not explicitly consider federated simulation in the remainder of this thesis.

A key goal of the partitioning process is to achieve an equal distribution of workload over all partitions while limiting inter-partition communication. Thus, depending on the properties of a given simulation model, assigning entities to partitions follows different strategies. For instance, large wired networks, such as the Internet, lend themselves to partitioning along backbone links as shown in Figure 2.5(a). Assuming that the majority of network traffic is between local nodes, this strategy maps clusters of local nodes to partitions in order to minimize cross-partition communication. In contrast, small and local wireless ad-hoc networks do not posses such backbone links. Instead, partitioning such networks often bases on the simulated geographic location of a network node.

Channel-parallel Partitioning: A partitioning scheme specific to network simulation is *channel-parallel partitioning* [LA96, LA97]. The scheme leverages the fact that wireless communication on non-interfering channels is independent. It partitions a simulated network according to non-interfering channels and forms clusters of network nodes communicating on the same channel. This scheme finds application primarily in simulations of cellular communication systems such as GSM, UMTS, or LTE in which neighboring cells use non-interfering channels (see Figure 2.5(b)). Nevertheless, it is not well suited when neighboring channels overlap such as in IEEE 802.11g/n or if the whole network transmits on the same channel, for instance in Code Division Multiple Access (CDMA) systems.

Time-parallel Partitioning: An even more specific partitioning scheme is *time-parallel partitioning* [LL91]. This scheme subdivides the time-domain of a simulation into distinct periods $p_0, \ldots, p_n$ and assigns each period to a separate

processing unit for parallel execution. Hence, each partition comprises the whole state of the simulation, but at different times of a simulation run as illustrated in figure 2.5(c). Since, however, the initial state of period p_i depends on the resulting state of period p_{i-1}, the simulation framework initializes a new simulation run by assigning randomly generated or user-provided initial states to each period. Subsequently, the simulation framework executes all periods in parallel and checks if the resulting state of p_{i-1} matches (approximately [Kie05]) the initial state of p_i. If the states do not match, the simulation iteratively repeats simulating period p_i with the final state of period p_{i-1} until both states converge.

This scheme is only applicable to simulations with a small state space, for instance queueing systems, to allow for fast convergence of initial and final states. The state of a network simulation is in general too large, thus rendering the scheme infeasible for time-parallel partitioning.

In summary, space-parallel partitioning is the conceptually most intuitive approach and the only generally applicable one. Thus, in the remainder of this thesis, the term "partitioning" refers to space-parallel partitioning if not explicitly stated otherwise.

2.3.2 Causal Correctness

The example of a naïve yet incorrect parallelization scheme presented in Section 2.2.3 illustrates the fundamental challenge in PDES: Enabling parallel event execution while maintaining a valid ordering of events. In the following, we formally define a distributed event handling scheme and a formal criterion for guaranteeing a causally correct event ordering.

2.3.2.1 Logical Processes

A partitioning scheme merely determines how the workload of a simulation is clustered and distributed across multiple processing units but it does not define a scheme for distributed event handling and the interaction among partitions. Hence, we introduce the notion of a *Logical Process (LP)*. An LP resembles a sequential discrete event simulation which is restricted to a subset of a larger simulation model. Specifically, each LP maintains a partition of a simulation model, a local FES, and a local virtual clock. By applying the sequential event handling scheme presented in Section 2.1, each LP forms an independent subsimulation.

However, LPs cannot simply execute in parallel as isolated simulations since each LP comprises only a fraction, i. e., a single partition, of a simulation model. Instead, activities in a simulation model spanning multiple partitions, consequently involve multiple LPs. Examples include a network packet traversing a space-parallel partitioned network, or a simulation entity reading from and writing to a data storage entity at a remote LP, e. g., a routing table. To handle inter-LP communication and data access, LPs exclusively communicate by exchanging timestamped events, thereby explicitly prohibiting direct data access or remote procedure calls. The reason for this restriction is that only timestamped events traverse the event scheduler

and thus enable a synchronization algorithm (see Section 2.3.3) to coordinate correct parallel event execution.

2.3.2.2 Causal Correctness

Based on the definition of logical processes, Fujimoto defines the Local Causality Constraint as a criterion for the order of event execution at each LP:

Definition 1 (Local Causality Constraint)
"A [parallel] discrete event simulation, consisting of logical processes that interact exclusively by exchanging timestamped [events], obeys the local causality constraint if and only if each logical process executes events in non-decreasing timestamp order." [Fuj90a]

Building on the notion of the local causality constraint, a parallel discrete event simulation is causally correct if it fulfills the following criterion:

Definition 2 (Causal Correctness)
A parallel discrete event simulation is causally correct if and only if it obeys the local causality constraint.

2.3.3 Synchronization Schemes

In order to fulfill the local causality constraint, parallel discrete event simulation employs synchronization algorithms. These algorithms generally fall into one of two categories: Conservative or optimistic synchronization. Conservative synchronization strives to strictly avoid causal violations during a simulation run. In contrast, optimistic synchronization allows causal violations to occur, but provides means for detecting and correcting causal violations.

The following sections briefly introduce both classes, compare their properties, and discuss combined and alternative synchronization schemes. However, over the last two decades, the research community has proposed numerous extensions, additions, and improvements of these basic algorithms. For the sake of brevity, we deliberately discuss only the fundamental techniques and algorithms in this background chapter to give a basic understanding of the general idea. We address relevant related efforts in depth in dedicated related work sections later throughout this thesis.

2.3.3.1 Conservative Synchronization

Conservative synchronization aims at strictly avoiding causal violations during simulation runtime. Thus, conservative synchronization only allows executing events which are *independent* and hence safe for parallel execution. More formally, given two events e_1 and e_2, conservative synchronization needs to determine whether or not parallel event execution will result in a causal violation. Hence, conservative synchronization aims at achieving two goals:

i) Determine whether or not events are independent before executing them, and

ii) maximize the set of independent events for maximizing parallel performance.

A general purpose discrete event simulation framework has no insight into the semantics of a simulation model, i. e., no knowledge of the causal relationship of events. As a result, the only property available a-priory execution is the timestamps of the events in the FES. Hence, conservative synchronization needs to take the synchronization decision solely based on *time information*. The most fundamental time information in this regard is the lookahead:

Definition 3 (Lookahead)
The lookahead in a conservatively synchronized parallel discrete event simulation, consisting of logical processes $lp_0, \ldots, lp_m$, is the lower bound on the difference in simulated time between the execution of an event e at lp_i and the arrival of an event e' created by e at lp_j, $i \neq j$.

We conclude from this definition that all events within the lookahead from one LP to another LP are independent. To see why, consider the following example. Assume a simulated communication network consisting of two nodes n_1, n_2 connected via a single link. Each node maps to a separate LP and the simulated link has a hypothetical propagation delay of 1 s. Then, the lookahead equals 1 s since any packet sent by n_1 at simulated time t takes 1 s to reach n_2. Therefore, any event e at n_2 with $t \leq t(e) \leq t + 1\,\text{s}$ is safe for parallel processing as no event with a timestamp smaller than $t + 1\,\text{s}$ can arrive at n_2 from n_1.

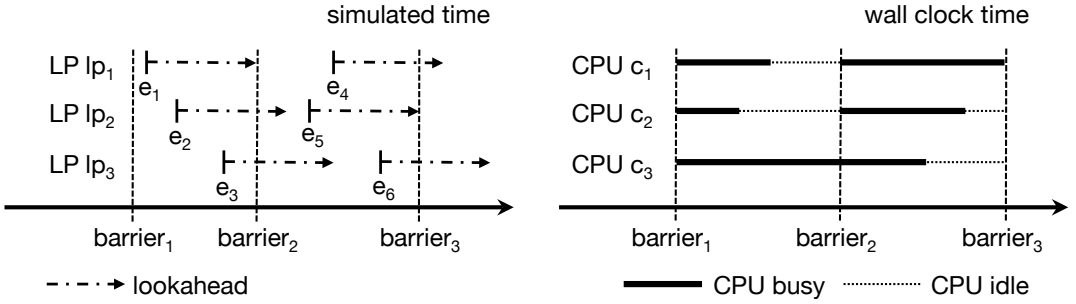
We furthermore define the event density as follows:

Definition 4 (Event Density)
The event density denotes the number of events per unit of simulated time.

In combination with the event density, the lookahead is a highly performance critical factor in conservative synchronization. A large lookahead with regard to the event density covers a large number of independent events. Consequently, with respect to the second goal of conservative synchronization stated above, it is imperative to extract the largest possible lookahead from a simulation model. Potential sources of lookahead are the propagation delay on a network link, the service time in a queueing system, or the processing delay of a hardware component. These examples indicate that the lookahead highly depends on the system under investigation. Revisiting the network example from above, we observe that intercontinental Internet links exhibit a latency of several milliseconds. In contrast, the propagation delay of wireless links in a WiFi network is in the range of merely nanoseconds. Moreover, the lookahead is often reduced further due to interference on the wireless broadcast medium. As a result, the lookahead constitutes the performance limiting factor in conservative synchronization [BT02, LN02, MB98, MB99].

Finally, to determine the set of independent events, each conservatively synchronized LP computes and maintains two time values:

Earliest Output Time (EOT): The EOT is the lowest bound on the *absolute* simulation time at which an event sent from one LP can affect a neighboring LP. Thus the EOT is given by the current local time of the LP plus the lookahead to its neighbors.



(a) The EIT among all LPs determines a barrier in simulated time. All events between two barriers are eligible for parallel processing.

(b) All LPs need to arrive at a barrier before a new round can start. Hence, per round all but one CPU exhibit an idle phase.

Figure 2.6 Illustration of synchronous, i. e., barrier-based, synchronization algorithms in simulated and simulation time.

Earliest Input Time (EIT): The EIT denotes the lowest bound on the *absolute* simulation time at which an event from a neighboring LP can take effect at the local LP. Hence, the EIT is the lowest EOTs of all neighboring LPs. In the literature [Fuj90a, Per06b], the EIT is also referred to as *Lower Bound on incoming Time Stamps (LBTS)*.

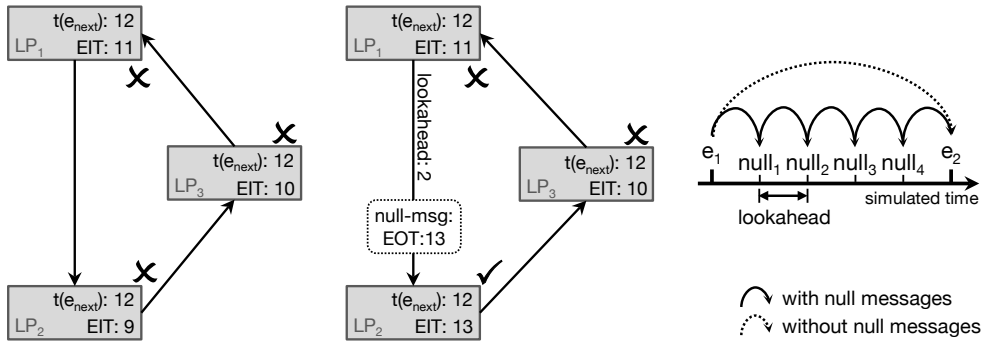
Based on these values, all events $e \in F$ of an LP with $t(e) \leq \text{EIT}$ are safe for parallel processing: Since no events with a timestamp lower than EIT can arrive at an LP, no causal violations can occur by executing all events up to EIT. However, if the EIT is smaller than the timestamp of all events in the local FES, an LP blocks until the EIT increases.

Conservative synchronization algorithms are further subdivided into two fundamental categories: Synchronous and asynchronous algorithms. The following sections present the general concepts as well as one selected algorithm from each category.

Synchronous Algorithms

Synchronous algorithms advance the simulation in rounds alternating between global synchronization among all LPs and parallel event execution. At the beginning of each synchronization round, all LPs communicate the timestamp of their next local event plus the lookahead to all other LPs. Using this information, all LPs compute a minimal global EIT, i. e., barrier. Subsequently, all LPs process events up to this barrier and re-synchronize (see Figure 2.6(a)).

The primary advantage of this class of algorithms is their simplicity and their ability to deal with zero lookaheads between LPs, since the global synchronization process considers the timestamp of actual events in the local FESs. However, global synchronization can become a performance bottleneck and limit scalability. For instance, since all LPs need to arrive at a barrier before a new round can start, early arriving LPs are blocked for a potentially long time. This in turn results in idling CPUs and a sub-optimal resource utilization (see Figure 2.6(b)). Moreover, due to the communication overhead of global synchronization, these algorithms typically find application in multi-threaded simulation using shared memory synchronization primitives [PVM09, Seg09]. Nevertheless, modern distributed communication libraries



(a) The NMA deadlocks due to a cyclic dependency: All LPs wait for a message from their neighbors.

(b) An EOT update via a null-message increases the EIT of LP_2 beyond the timestamp of its next event.

(c) Time creeping problem: Small lookaheads cause excessive numbers of null-messages between events.

Figure 2.7 The Null Message Algorithm uses null-messages to break deadlocks (Figures (a) and (b)). However, small lookahead can cause large numbers of null-messages, thereby causing the time creeping problem (Figure (c)).

such as the Message Passing Interface (MPI) provide global collection operations, e. g., `MPI_ALLgather`, which foster the use of synchronous algorithms in distributed simulation [BBC+12, PR11].

Asynchronous Algorithms

Asynchronous algorithms alleviate the drawbacks of global synchronization by letting all LPs continuously and independently distribute EOTs to the neighboring LPs. Using these EOT-updates from their neighbors, LPs continuously advance their own local EIT. To reduce the communication overhead of the update messages, LPs can piggyback updates on top of ordinary events sent to neighboring LPs. However, a fundamental limitation of this approach is deadlocks. Figure 2.7(a) illustrates a simple example in which three LPs block due to a cyclic dependency. All three LPs wait for a message from a neighboring LP which would allow for increasing the EIT. However, due to the cyclic waiting condition, no messages will ever be sent and the simulation deadlocks.

The *Null Message Algorithm (NMA)* [CM79] solves this issue by means of additional synchronization messages. Every time an LP blocks or updates its local EOT, it proactively sends the current respectively the new EOT to all neighboring LPs to enable them recomputing their EITs. Since this update message does not carry any meaning in terms of the simulation model, it is hence called null-message. In the example in Figure 2.7(b), sending null-messages upon blocking allows all LPs to iteratively advance their local EIT until actual simulation events are eligible for processing. Nevertheless, a critical restriction of the Null Message Algorithm is its requirement for non-zero lookaheads. Otherwise LPs would indefinitely send null-messages with identical EOTs, hence preventing the receiving LPs to update their EIT resulting in a livelock.

A further limitation of the NMA is the *Time Creeping Problem* that arises in conjunction with small lookaheads and low event densities [Fuj99b]. The root of the

problem is that null-messages iteratively advance EITs, but the amount of simulated time by which the EIT can be advanced depends on the lookahead. In case the lookahead is much smaller than the time intervals between events, an excessive number of null-messages is needed to advance the simulation time to the next event as shown in Figure 2.7(c). For instance, assume an LP with a current local time of 10 and a local event e with $t(e) = 100$. Assuming furthermore a lookahead to its neighboring LP of 1, this LP needs to receive 90 null messages, in the worst case, until the EIT reaches 100, making the event eligible for processing. This limitation is a consequence of the asynchronous nature of the NMA and not present in synchronous algorithms due to global synchronization.

Despite these limitations, the NMA is widely applied in distributed parallel simulation. Since it is one of the earliest proposed algorithms, many different flavors of the NMA exist today [BT02, Fuj90a, Nic96, Per06b]. These flavors differ in the details, e. g., in the exact methods of computing EITs or the laziness of disseminating EOTs, yet the basic concept corresponds to the one presented here.

2.3.3.2 Optimistic Synchronization

Conservative synchronization often unnecessarily blocks parallel execution of independent events if it cannot yet determine their independence, mainly due to small lookaheads (cf. time creeping problem). To solve this problem, *optimistic synchronization* generally assumes that two events are independent and do not interfere. It thus executes all events *speculatively* as soon as they are available in the FES.

However, since speculative execution inevitably results in causal violations when two events do in fact interfere, optimistic synchronization provides means for detecting and correcting causal violations. Following the definition of the local causality constraint, optimistic synchronization detects a causal violation if an LP lp_i receives an event e with $t(e) < t(lp_i)$, i. e., a timestamp smaller than its current local time. In order to correct a causal violation, optimistic synchronization algorithms employ either state saving or reverse computation techniques, as discussed in the following.

State Saving Algorithms

State saving algorithms, most notably the *Time Warp Algorithm* [Jef85], continuously save the state of all LPs at simulation runtime in the form of checkpoints. Upon detecting a causal violation at LP lp_i, the algorithms *roll back* the state of lp_i to a correct state by restoring a previously stored checkpoint, typically the one directly preceding the timestamp of the conflicting event.

However, the invalid state of the simulation model is not restricted to lp_i, but any LP that received a message from lp_i since the last checkpoint may be in a potentially incorrect state. In order to restore their states as well, lp_i sends for each message sent since the respective checkpoint one so called anti-message to its neighbors. If a message and its corresponding anti-message coincide in the FES of an LP, i. e., the original message was not processed yet, both messages cancel each other out. If, however, the original message was already processed when receiving the anti-message, the respective LP performs a rollback as well. Eventually, this recursive rollback scheme eliminates all invalid messages and states from the simulation model.

Reverse Computation Algorithms

An inherent problem of the state saving approach is extensive memory consumption. *Reverse computation* [CPF99, TPF+05] mitigates the memory consumption problem by trading computational load for memory resources. Specifically, for every invalid instruction performed since the timestamp of the conflicting event, reverse computation executes an inverse instruction undoing the incorrect state changes. Put simply, this synchronization scheme reverts the effects of an erroneous addition by performing a corresponding subtraction. Modelers can either manually provide inverse event handlers for each event occurring in the simulation or employ compiler-based techniques to generate corresponding inverse events.

However, reverse computation still needs to maintain additional state information for two reasons. First, in order to correctly reverse an event handler, the scheme needs to follow the exact inverse control flow path of the previous erroneous execution. Hence, for each instruction influencing the control flow, e. g., conditional branches or loops, a control-flow-state holds information allowing for constructing the reverse control flow path, e. g., the branches taken or the number of loop iterations. Second, not all operations in a computer are accurately reversible. For instance, floating point operations introduce rounding errors and operations such as modulo eliminate information about the original operands. To handle such destructive instructions, reverse computation resorts to state saving. Thus, reverse computation achieves an actual reduction in the memory consumption if the total size of the control flow and the destructive instructions state is smaller than the entire data state of a simulation model.

2.3.3.3 Comparison of Conservative and Optimistic Synchronization

Although conservative synchronization algorithms achieve distributed coordination, their actual implementation is far less complex than that of optimistic algorithms. If no safe events are available, conservative synchronization simply suspends execution of further events and waits for safe events to become available again. Hence, resource utilization, particularly memory consumption, is lower, thus benefiting the scalability of the simulation model. However, the key performance factor of conservative synchronization is the lookahead which in turn strongly depends on the particular system under investigation. As a result, conservative synchronization is too pessimistic when confronted with a small lookahead, thus hindering parallel execution of actually independent events.

In contrast, optimistic synchronization is largely independent of the lookahead available in a given simulation model. This makes optimistic synchronization particularly suitable for simulation models with small lookaheads, such as models of wireless networks. Moreover, as long as no causal violations occur at simulation runtime, optimistic synchronization does not impose additional communication overhead since it does not make use of synchronization messages. However, in order to avoid frequent rollbacks, optimistic synchronization requires uniform progression of all LPs through simulated time. Large differences in simulated time at the LPs increase the risk of receiving a message with a timestamp preceding the local time of the receiving LP. Furthermore, the implementation of optimistic synchronization algorithms is highly

	Conservative Synchronization	Optimistic Synchronization
Key advantages	simplicity	independent of lookahead
Key drawbacks	too strict	high memory usage, too aggressive
Implementation complexity	simple	complex
Performance factors	lookahead	uniform progress of LPs,

Table 2.1 Comparison of conservative and optimistic synchronization in terms of key advantages & drawbacks, implementation complexity, and performance factors.

complex due to the need for efficient memory management, checkpointing, or compiler supported reverse computation. In particular, extensive memory utilization severely limits the scalability of simulation models.

Table 2.1 summarizes and compares the properties of conservative and optimistic synchronization in terms of implementation complexity, performance factors, and key advantages as well as drawbacks.

To mitigate the drawbacks of purely conservative and purely optimistic algorithms, the research community proposed combinations of both schemes [JB94, NL02, Per05, RAT93]. For the sake of brevity we do not detail on these approaches here, but defer the discussion of related efforts on combined synchronization to the dedicated related work sections later throughout this thesis.

2.3.4 Parallel Event Execution Environments

Until now, PDES merely denotes the general concept of processing multiple events in parallel on different processing units. This concept, however, does not specify the runtime environment, i. e., the hardware and software platform, used for actually executing the simulation. In the context of this thesis it is important to differentiate between three parallel execution environments: i) distributed simulation, ii) multi-threaded simulation, and iii) distributed multi-threaded simulation.

Distributed Simulation: A distributed simulation typically runs on a computing cluster consisting of multiple individual compute nodes. Each compute node contributes computing resources, e. g., processing power, memory, and storage space, to the cluster. Communication between compute nodes utilizes a communication network, for instance based on commodity IEEE 802.3 Ethernet [iee] or specialized, low-latency interconnects such as Myrinet [BCF+95] or InfiniBand [inf].

Each compute node executes one or more partitions (or federates) of a distributed simulation as an individual process. As compute nodes communicate solely across the interconnect, no globally shared memory space is available to the processes. Hence, synchronization between partitions relies entirely on message passing.

Multi-threaded Simulation: A multi-threaded simulation executes on a multi-processor computer which provides all available computing resources. In contrast to distributed simulation, a multi-threaded simulation runs in a single process comprising multiple worker threads. These worker threads share a common global memory space which enables them to access the entire state of the simulation model and share the workload, i. e., event processing. Moreover, as threads share a common memory space, multi-threaded simulation utilizes in-memory synchronization primitives such as mutexes, barriers, and spinlocks.

Distributed Multi-threaded Simulation: Due to the proliferation of multi-processor computers, compute nodes in a cluster provide multiple processors. As a result, distributed multi-threaded simulation follows a hierarchical approach by distributing multiple partitions of a simulation model to a compute node which in turn utilizes its processors to execute local partitions in parallel.

The goal of this thesis is to develop efficient parallel simulation techniques which are specifically tailored to the characteristics of multi-core systems. Since multi-core systems provide fast and scalable thread synchronization mechanisms utilizing a globally shared memory space, multi-threaded simulation constitutes a natural foundation for our work. Furthermore, we aim at making desktop and server multi-core systems available to model developers and researchers as cheap and simple-to-use basis for parallel simulations. Considering the cost and complexity of a distributed simulation setup, the remainder of this thesis hence focuses solely on multi-threaded simulation.

2.4 Parallel Discrete Event Simulation Frameworks

As a result of the extensive research in the domain of parallel simulation, a wide range of simulation frameworks support different flavors of parallel simulation. This section briefly reviews the state-of-the-art in parallel simulation frameworks and their key properties. We complement this overview with in depth discussions of particular related efforts later throughout this thesis in dedicated related work sections.

2.4.1 Overview

In the following, we present a brief overview of related efforts targeting parallel discrete event simulation. This overview differentiates between simulation frameworks created by the research community and commercial frameworks.

2.4.1.1 Simulation Frameworks based in the Research Community

Parallel discrete event simulation is in the focus of the research community for over two decades. Consequently, the combined efforts of the community resulted in a multitude of excellent parallel simulation frameworks.

OMNeT++: The OMNeT++ framework [Var01] is an open source, general purpose discrete event simulation environment. In terms of parallel simulation, it supports distributed simulation synchronized by means of the Null Message Algorithm [SVE03]. Despite not featuring multi-threaded simulation, multiple processes of a distributed simulation can operate on a single multi-core machine, thereby utilizing all CPUs.

In order to be suitable for parallel simulation, simulation models are restricted to static topologies with non-zero link delays to accommodate the Null Message Algorithm and the space-parallel partitioning scheme. OMNeT++ enables transparent cross-partition communication by means of proxy gates and placeholder modules. Each partition knows about the entire topology of a given simulation model, however, modules not mapped to a particular partition are replaced by placeholder modules which send incoming events towards the real modules.

Parallel/Distributed ns (pdns): Building on top of the widely used ns-2 network simulator [BEF+00], pdns [RFA99] extends ns-2 with parallel simulation capabilities. Specifically, Parallel/Distributed ns (pdns) connects multiple instances of ns-2, called federates, to form a distributed simulation. Moreover, it utilizes the RTIKIT [FH98] runtime which enables communication and conservative synchronization between federates. Like OMNeT++, pdns exploits multi-core machines by placing multiple federates on one machine despite not supporting multi-threaded execution.

Each federate handles one partition of a space-parallel partitioned simulation model. In contrast to OMNeT++, each federate only knows about the network nodes mapped to its partition in order to minimize the memory footprint of large scale simulation models. Hence, to allow for defining and creating logical connections between simulated nodes, pdns extends the model description language with unique (IP address, port)-identifiers which can be used across partition boundaries.

GTNetS: Based on the experiences gained during the development of pdns, scalability is a primary concern in the architectural design of GTNetS, the Georgia Tech Network Simulator [Ril03]. In terms of parallelization, it builds upon a similar architecture as pdns. Thus, GTNetS utilizes remote links for connecting nodes across a space-parallel partitioned and conservatively synchronized simulation model. In contrast to ns-2 however, GTNetS replaces the RTIKIT with MPI which allows for distributed simulation on computing clusters and multi-core machines. More importantly, it eliminates the dual Tcl/C++ programming language approach of ns-2 and solely focuses on C++, resulting in a significant reduction in memory consumption. The distributed simulation architecture of GTNetS eventually became a fundamental building block of ns-3.

ns-3: Like its predecessor, ns-3 [HRFR06] supports conservatively synchronized distributed simulation of space-parallel partitioned simulation models. In contrast to ns-2, inter-partition communication and synchronization rely on MPI instead of RTIKIT. ns-3 achieves conservative synchronization by determining a global Lower Bound on incoming Time Stamps (LBTS) across all links

between all partitions in a round based fashion. As a result, partitioning a simulation model is restricted to so called point-to-point links which provide a propagation delay used as lookahead.

In addition to distributed simulation, the ns-3 project conducted efforts towards multi-threaded simulation [Seg09]. Despite applying the same synchronization and partitioning techniques as in distributed parallelization, this effort however was discontinued due to the runtime and maintenance overhead involved with a thread-safe simulation core. Instead, ns-3 recommends using the shared memory communication capabilities of MPI for utilizing the CPUs of multi-core computers by means of distributed parallelization.

SSF (SSFNet / DaSSF): Despite its name, the Scalable Simulation Framework (SSF) [CON02] is in itself not a simulation framework, but rather a specification of a simulation API targeting efficient parallel simulation. SSFNet and DaSSF constitute concrete implementations of SSF for the Java and C++ programming language, respectively. SSFNet applies solely a multi-threaded parallel execution scheme, whereas DaSSF supports both multi-threaded as well as distributed simulation over MPI.

PARSEC / GloMoSim: PARSEC (PARallel Simulation Environment for Complex Systems) [BTC+98] is a programming language and environment specifically designed for parallel simulation – in contrast to the aforementioned simulators which regard parallelization as an additional feature. It hence supports distributed as well as multi-threaded simulation utilizing conservative and optimistic synchronization. Using PARSEC as substrate, GloMoSim (Global Mobile system Simulator) [ZBG98] is a collection of simulation models targeting scalable simulation of wireless network. Despite considerable development effort, the public maintenance and development of both PARSEC and GloMoSim ended over a decade ago.

ROOT-Sim: The only simulator discussed here that explicitly supports optimistic synchronization is ROOT-Sim [PVQ11]. It follows the classic state saving approach (full and incremental), yet employs sophisticated memory management techniques to minimize runtime overhead.

Simulation models consist of individual entities, i. e., LPs, which map to simulation kernels, i. e., distributed simulation processes communicating over MPI. ROOT-Sim thus uses space-parallel partitioning for enabling distributed simulation. Originally targeting only distributed simulation, recent efforts extend ROOT-Sim with multi-threaded simulation capabilities [VPQ12].

2.4.1.2 Commercial Simulation Frameworks

Following the success of multi-core computers, commercial simulation frameworks provide parallel event execution capabilities to their customers. However, since the source code of commercial tools is typically not publicly available, the research community depends on white papers to gain an insight into the proprietary techniques underlying these tools.

OPNET Modeler

OPNET Modeler [Cha99, opn] by OPNET Technologies is a commercial simulation tool focusing on communication systems, in particular wireless networks. It provides support for both multi-threaded simulation as well as distributed simulation on computing grids. Moreover, OPNET is compatible with the IEEE 1516 High Level Architecture (HLA), thereby enabling heterogeneous compositions of different simulation frameworks.

Based on experience, parallel simulation on multi-core computers is largely transparent to model developers. The framework merely requires setting a compile time option in order to enable parallel execution. Unfortunately, the algorithms and techniques employed by OPNET are not available to the general research community. Hence, we are not able to accurately analyze and compare the techniques used by OPNET with the contributions of the research community, including our own work.

Qualnet

Qualnet [qua] by Scalable Network Technologies is the commercial version of the GloMoSim simulation framework. Similarly to OPNET, Qualnet also provides a rich set of models for communication systems. In terms of parallelization, it offers multi-threaded as well as distributed simulation execution, the latter also supporting the HLA standard. Although based on GloMoSim, Qualnet might utilize proprietary parallelization techniques which are not publicly available. We hence cannot assess the techniques of Qualnet.

2.4.2 Comparison and Conclusion

Table 2.2 summarizes the properties of the previously discussed simulation frameworks. Clearly, conservative synchronization constitutes the de facto standard in terms of synchronization. We ascribe this to a considerably lower implementation effort. Moreover, conservative synchronization offers a higher degree of transparency to the user by not requiring explicit memory management or reverse event handlers. Overall, this results in a better adoption of parallel simulation.

In addition, due to the origins of parallel simulation in computing clusters, the majority of simulation frameworks employ distributed simulation techniques. By mapping multiple processes of a distributed simulation to multi-core machines, distributed simulation implicitly supports multi-core architectures. However, we argue in this thesis that this simple mapping does not fully and efficiently exploit the processing power and capabilities of multi-core computers. Instead, we claim that parallel simulation on multi-core systems has to explicitly make use of the characteristics of the underlying multi-core hardware platform. In particular, utilizing shared-memory and fast thread synchronization opens new possibilities for novel approaches towards efficient parallel discrete event simulation.

Based on this reasoning, we present parallel expanded event simulation [KLG+10, KLW09] and the corresponding Horizon simulation framework in the next chapter. Parallel expanded event simulation constitutes a new modeling paradigm aiming for

	Synchronization Scheme		Parallel Event Execution	
	Conservative	Optimistic	Distributed	Multi-threaded
OMNeT++	✔	✘	✔	✘
ns-2	✔	✘	✔	✘
ns-3	✔	✘	✔	✔
GTNetS	✔	✘	✔	✘
PARSEC	✔	✔	✔	✔
ROOT-Sim	✘	✔	✔	✔
DaSSF	✔	✘	✔	✔
OPNET	✔	✘	✔	✔
Qualnet	✔	✘	✔	✔
Horizon	✔	✔	(✘)*	✔

Table 2.2 Comparison of state-of-the parallel discrete event simulation frameworks in terms of the supported synchronization and parallel event execution schemes. *: HORIZON is based on OMNeT++ and thus inherits the distributed simulation capabilities of OMNeT++, yet without support for parallel expanded event simulation.

efficient parallelization of tightly coupled systems such as wireless networks. HORIZON, in turn, is an extension of OMNeT++ that implements parallel expanded event simulation and hence puts the new modeling paradigm into practice. Both contributions focus specifically on multi-core systems and heavily utilize their shared memory capabilities. Hence, HORIZON deliberately does not support parallel expanded event simulation in a distributed environment simulation, despite inheriting the distributed simulation capabilities of OMNeT++ (see Table 2.2).

3

Parallel *Expanded* Event Simulation

This chapter presents parallel *expanded* event simulation, a novel modeling paradigm that explicitly represents the processing time of physical processes by extending discrete events to span a period of simulated time. Based on the notion of expanded events, we define a conservatively synchronized parallelization scheme in which overlapping expanded events are eligible for parallel processing. We furthermore show the viability of our approach by implementing and evaluating parallel expanded event simulation in Horizon, a parallel simulation framework extending OMNeT++.

3.1 Motivation

Discrete event-based network simulation currently faces two significant changes: First, recent advances in wireless communication technology demand highly accurate simulation models, resulting in a steep increase in model complexity and runtime requirements. Second, multi-core computers constitute the de facto default hardware platform even for desktop systems, thus providing a cheap yet powerful parallel processing platform. As a result, parallel discrete event simulation significantly gained in importance and is therefore (again) in the focus of active research.

Model Complexity

Simulation models of wireless networks require considerably more detailed models of the lower network layers than models of wired networks [WGG10]. In particular, wireless transmissions rely on precise models of the physical layer and the wireless channel to capture the subtle effects and interactions of advanced wireless communication technologies such as OFDM [KRT02] or turbo coding [BGT93]. With the advent of even more complex systems using MIMO transmissions [GSS+03] or successive interference cancellation [HAW08], this trend will intensify in the near future. In conjunction with these systems, the MAC layer depends on highly accurate

timing. It is hence important to reflect even short delays, such as the processing time of algorithms and hardware components, accurately in simulation. Moreover, the computational complexity of simulation models is further amplified by the fact that the wireless channel is a broadcast domain. This causes a *tight coupling* between simulated network nodes resulting in a higher number of nodes involved in individual transmissions than in wired networks. Consequently, simulation runtimes increase drastically which in turn hampers development and in-depth evaluation of communication systems.

Researchers often work around these issues by trading accuracy for shorter runtimes [HBE+01, JZTB06]. Such trade-offs, however, need to be applied carefully as they may lead to incorrect simulation results [TMB01]. Furthermore, studies of the long-term behavior of complex systems, i. e., the impact of changing system loads over the course of a day, require efficient simulation execution to restrict simulation runtimes to reasonable lengths. Consequently, we identify the need for parallel execution of discrete event simulations to successfully deal with complex simulation models.

Multi-core Computers

The major trend in processor design over the last five to ten years is to move from single-core designs towards multi-core chips. The driving factor behind this development is that the traditional means of increasing performance through higher clock speeds has reached physical limits. In accordance with Moore's Law [Moo65], stating that the density of transistors on a chip doubles every year, miniaturization of circuits allowed for continuously increasing clock speeds, thus resulting in improved performance. As a result, common practice suggested that if a computational problem was not sufficiently fast solvable, waiting for the next generations of CPUs in fact solved the problem. However, due to excessive power dissipation, the steady increase in clock speed could not be upheld.

Still, Moore's Law proved to hold over time. Hence, chip vendors started to invest increasing transistor counts in duplicate functionality, i. e., multiple cores on a single die, while clock speeds remained relatively stable. The direct consequence of this development is that a gain in performance is not for free anymore. Given a sequential program, e. g., a discrete event simulation, its performance does not improve significantly with new generations of CPU, as it did before. Instead, simulations must explicitly employ parallel algorithms to efficiently exploit the processing power of today's and future multi-core systems.

The research community dedicated considerable efforts to investigating the feasibility and scalability of parallel simulation [Fuj90a, Liu09, Nic96, Per06b], thereby laying the foundation for parallel simulation frameworks [pdn, CS05, CON02, Ril03, Var01, BFBC06]. Traditionally, the primary focus of many of these works is on distributed simulation on computing clusters. However, such hardware is not available to the average simulation user and large scale simulations are difficult to setup and maintain [FPP+03]. Thus, parallel simulation is still restricted to very large simulations and does not find wide spread application in small or medium sized simulations on small or medium sized parallel computers. In particular, modern workstations at every networking researcher's desk constitute such medium sized

multi-core computers, lending themselves to parallel simulation. We conclude from these observations the need for a simple-to-use parallel simulation framework that allows researchers to take advantage of the parallel processing power readily available in modern multi-core computers.

3.2 Problem Analysis

In the previous section, we derived the need for parallel simulation techniques which enable an efficient execution of complex simulation models on multi-core systems. This section analyzes the challenges we are facing towards this goal. In particular, we investigate the properties of wireless system models and identify a fundamental modeling mismatch between reality and simulation which jointly hinder efficient parallelization.

3.2.1 Properties of Wireless System Models

Large scale models of wired networks, e. g., peer-to-peer systems, were traditionally the primary application of parallel network simulation. Yet, the proliferation of wireless transmission technology has shifted the focus of interest in the research community from wired to wireless networks.

This shift results in a significant increase in computational complexity: The isolated and almost error-free transmission medium of wired networks allows for abstracting almost entirely from physical layer details. In contrast, models of wireless networks require detailed modeling of the transmission medium, i. e., the wireless channel, to capture interference and physical layer effects such as pathloss, fading, and shadowing. The corresponding computations are complex and cause extensive simulation runtimes which in turn create the need for parallelization. However, the true challenge of parallelization, and thus this thesis, is *not* the computational complexity. In fact, *if* it is possible to execute sufficiently many computationally complex events of a wireless network model in parallel, the speedup and the resulting runtime reduction is excellent. Hence, the true challenge of parallelization is to *identify* whether or not events are independent, i. e., eligible for parallel processing.

In this regard, the key difference between wireless and wired systems is a much tighter coupling of the entities in wireless systems [LN02]. This is due to two properties: i) The wireless channel is a broadcast domain, and ii) wireless networks are generally of smaller size than wired networks. In terms of parallel simulation, these properties have two implications:

Small Lookahead: Models of wireless networks exhibit substantially smaller lookaheads than models of wired networks. Recall that the source for lookaheads in network simulation is typically the propagation delay along links which cross the boundaries of partitions (cf. Section 2.3.3.1). For this reason, (space-parallel) partitioning of large scale wired networks, e. g., the Internet, divides the network along long-haul backbone links. These links, often abstracting

from single network devices, such as routers and switches, feature delays ranging from micro- up to milliseconds [MTK06]. In wireless networks, however, no such abstract backbone links exist. Instead, wireless links just range between tens of meters (e. g., Bluetooth) up to a few kilometers (e. g., Global System for Mobile Communications (GSM)). Since signals along these links travel at the speed of light, propagation delays span merely nanoseconds (e. g., 33 ns for 10 m) to low microseconds (e. g., 3.3 µs for 1 km).

As introduced in Section 2.3.3.1, the lookahead and the event density are of key importance for achieving good performance in conservatively synchronized parallel simulations. While the event density in wireless network models is higher than in wired system models due to accurate modeling of MAC protocols, a key defining factor of the event density is the traffic rate, i. e., number of packets per second in the network. Since the traffic rate is low in relation to the size of the lookahead, i. e., only few events are covered by the lookahead, conservative synchronization algorithms struggle to identify parallelism and achieve good performance [LN02, MB98].

High Connectivity among Network Nodes: Inherently, the wireless channel is a broadcast domain, i. e., every transmission reaches every node in range and also influences every other transmission in range in terms of interference. As a result, many wireless networks, such as wireless mesh networks, comprise a highly connected topology. This in turn hinders space-parallel partitioning as the network cannot easily be divided in loosely connected clusters of network nodes for efficient parallel simulation. Instead, each LP handling a partition is connected to many neighboring LPs, thereby increasing the synchronization overhead in conservative synchronization. Moreover, the high degree of connectivity among LPs increases the chance for receiving a conflicting event from the past in optimistic synchronization, thus, resulting in frequent rollbacks and poor parallel simulation performance.

3.2.2 Modeling Time-Spans in Discrete Event Simulation

We claim that the default approach to modeling virtual time in discrete event simulation does not closely reflect the behavior of real systems. While this mismatch does not result in invalid models, it nevertheless hides dependency information between events which allow for more efficient parallel simulation. In the following, we elaborate on this in more detail.

Physical real-world processes generally span non-zero intervals of wall-clock time. Examples for such processes are switching delays of the hardware, the computing time of algorithms, and physical effects such as signal propagation. In contrast, the fundamental modeling paradigm in discrete event simulation is that events occur at specific points in simulated time, as defined by their timestamp, yet handling an event does not advance simulated time beyond the timestamp (cf. Section 2.1). Consequently, a single event by itself cannot represent a process spanning a period of simulated time. Still, discrete event simulation is of course fully capable of modeling such processes, for instance by means of two separate events indicating the beginning and the end of a process. However, we argue that using two events hides dependency information that is valuable for parallel simulation:

Assume a simulation model that represents a process spanning a period of simulated time by means of an event e_s indicating the start and e_c modeling the completion of the process. In this scenario, we can derive simple facts about the relation between e_s and e_c:

i) if e_s occurs in the model, then e_c occurs as well,
ii) e_c will always succeed e_s,
iii) the time span $t(e_c) - t(e_s)$ is equal to or larger than zero.

Considering these relations in the synchronization algorithm allows for more efficient event scheduling due to a better insight into the ordering and timing of events. Yet, synchronization algorithms are not able to derive similar conclusions since the relationship between e_s and e_c is not known to them. As a result, they handle e_s and e_c separately.

Summarizing, wireless systems constitute a highly challenging application for parallel discrete event simulation because of a tight coupling between simulated entities. Due to small lookaheads and a high connectivity among network nodes, conservative as well as optimistic synchronization algorithms suffer from performance limitations. The problem of small lookaheads is further aggravated by the fact that discrete event simulation cannot express dependencies among events which model time spans. Thus, we conclude the need for an extended modeling paradigm that enables efficient parallel simulation of tightly coupled systems.

3.2.3 Goals

Based on the observations stated in the motivation and the problem analysis, we define three goals for this chapter:

Goal 1 - Define an Extended Modeling Paradigm:
We aim at eliminating the modeling mismatch between physical processes and discrete event simulation. To this end, we need to develop a novel modeling paradigm that extends events with processing durations.

Goal 2 - Exploit the Extended Modeling Paradigm for Parallel Simulation:
The extended modeling paradigm provides additional event dependency information in comparison to classic discrete event simulation. Hence, we target the development of a parallelization scheme that exploits this dependency information.

Goal 3 - Make Efficient Use of Multi-core Computers:
The parallelization scheme should make explicit use of the properties of multi-core systems. This involves in particular the globally shared memory space for efficient synchronization and load balancing.

3.3 Parallel Expanded Event Simulation

This section presents parallel expanded event simulation, our approach to achieve the previously stated goals. At first, we briefly sketch the general idea of our par-

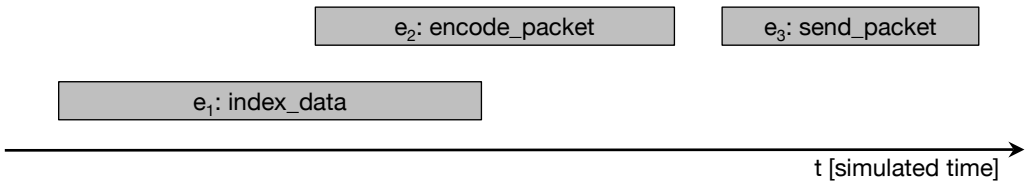


Figure 3.1 Determining independent events: The simple example shows three expanded events that span a certain period of simulation time each. The events e_1 and e_2 cannot depend on each other due to their temporal overlapping and can thus be executed in parallel. e_3 must follow sequentially.

allelization scheme before specifying it formally. We furthermore discuss techniques for obtaining expanded events and review related efforts.

3.3.1 General Idea

The core idea of our approach is to augment simulation models with additional domain specific information to support synchronization algorithms in identifying independent events. As pointed out previously, discrete events occur instantaneously at discrete points in simulation time. However, let's assume we extend this modeling principle with the ability to handle events that span a period of simulated time. Given such functionality, we can annotate events with *durations* which naturally model the delay of simulated processes and algorithms.

Figure 3.1 shows a simple example of three augmented events e_1, e_2, and e_3 representing a "data indexing", a "packet encoding", and a "packet sending" process. We observe that for the particular timing chosen in this example, e_1 and e_2 overlap in time while e_3 follows after the end of e_2. The overlapping implies that e_2 cannot depend on any results generated by e_1 because e_2 already begins while e_1 is still processing, i. e., its results are not yet available. Consequently, we conclude that both events are independent and can thus be processed in parallel. However, we cannot conclude whether or not e_3 is independent from the other two events since it begins after the earlier events have finished. In this example, it is indeed dependent on e_2 which calculates the encoded packet that is sent by e_3.

In general, it is the task of the model developer to specify durations for the events of a simulation model. Although this constitutes additional modeling effort, we argue that model developers have a profound understanding of the system under investigation and can hence specify event durations with reasonably low effort. Hence, our approach requires a larger modeling effort than traditional parallel discrete event simulation, but we aim for exploiting the additional modeling information for improved parallel simulation performance.

3.3.2 Expanded Events

Building on the ideas of the previous example, we now specify a novel modeling paradigm that explicitly augments events with time spans. Since such events extend discrete events to span a period of simulated time, we refer to them as *expanded*

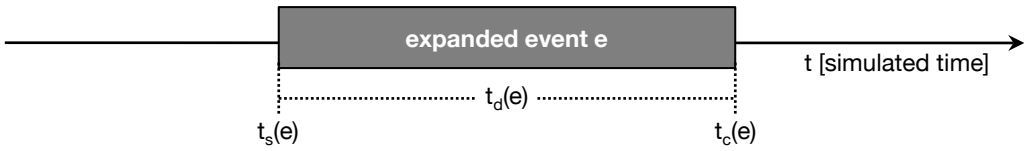


Figure 3.2 An expanded event e spans a period of simulated time ranging from its starting time $t_\mathsf{s}(e)$ to its completion time $t_\mathsf{c}(e)$. The period between $t_\mathsf{s}(e)$ and $t_\mathsf{c}(e)$ is the event duration $t_\mathsf{d}(e)$.

events (see Figure 3.2). Consequently, we denote a simulation implementing this modeling paradigm an *expanded event simulation*.

Definition 5 (Expanded Event)
An expanded event is defined by a distinct *starting time* and a distinct *completion time*. We refer to the difference in simulated time between start and completion time as *event duration*. Formally, we define the following functions:

- $t_\mathsf{s} : E \to T, e \mapsto t$ maps an event e to its starting time in simulated time T,
- $t_\mathsf{c} : E \to T, e \mapsto t$ maps an event e to its completion time in simulated time T,
- $t_\mathsf{d} : E \to T, e \mapsto t$ maps an event e to its duration in simulated time T.

The event duration $t_\mathsf{d}(e)$ of an expanded event e is $t_\mathsf{d}(e) = t_\mathsf{c}(e) - t_\mathsf{s}(e) \geq 0$.

We explicitly allow $t_\mathrm{s}(e) = t_\mathrm{c}(e)$ in order to represent traditional *discrete events* for two reasons. First, this provides backwards compatibility to existing simulation models and enables their seamless transition to duration based modeling. Second, simulation models may use discrete events to perform maintenance tasks such as propagating metadata via side channels. Such tasks do not map to real physical processes and thus do not span concrete event durations. It is nevertheless possible to assign pseudo durations to these events to achieve better parallel performance, as discussed in Section 3.4.4.4.

For now, we assume that the event duration $t_\mathrm{d}(e)$ of a given event e is predefined and static. Section 3.3.3.1 details on how to integrate dynamic event durations in the modeling and event execution process of an expanded event simulation. Nevertheless, given an event duration, we define a restriction on the starting time of newly created events. To this end, we first define a successor relationship among events.

Definition 6 (Successor Event)
Event e_2 is a successor of e_1 if and only if e_1 creates e_2, denoted by $e_1 \curvearrowright e_2$.

Based on the successor relationship, we restrict the starting time of all successor events as follows.

Definition 7 (Starting Time of Successor Events)
For all expanded events e_1, e_2 with $e_1 \curvearrowright e_2$ holds $t_\mathsf{c}(e_1) \leq t_\mathsf{s}(e_2)$.

This means that new events may only start *after* the event (i. e., a physical process) that creates them has finished. This corresponds to the assumption previously stated in the example in Figure 3.1 that the results of an expanded event may only become visible to the entire system after the event has been processed.

3.3.3 Sequential Expanded Event Execution Model

In order to specify a *sequential* execution model for expanded event simulation, we need to consider the extended time information of expanded events. Specifically, we address three questions: i) How to integrate event durations in the modeling process? ii) How to advance the global simulation time when executing an expanded event? iii) How to sort expanded events in the FES? The following discussion illustrates our design choices. Finally, we define a sequential expanded event execution model and show its causal correctness.

3.3.3.1 Modeling Event Durations

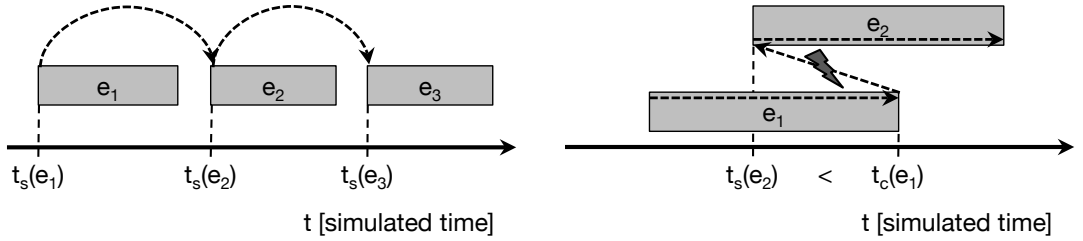
From a modeling perspective, the duration of an expanded event is not static in general. Much like the processing duration of physical processes, it can depend on dynamic inputs, e. g., packets of varying length. Hence, the actual extend of event durations can often only be determined dynamically at runtime. Thus, we integrate dynamic modeling of event durations in the event execution model of expanded event simulation as follows: Just before the event scheduler executes an instance of an expanded event, it hands the event instance to the simulation model. The model then determines the exact event duration based on its current state and the state of the event.

This in turn means that a model developer does not need to specify the event duration when creating a new expanded event. The reasoning for deferring the specification of the event duration from the creation of an event to right before its execution is two-fold:

i) Determining the duration just before executing an expanded event prevents outdated timing information. This is due to the fact that an arbitrary number of events can occur between creating and actually executing an event. These events can change the state of the simulated system, resulting in different event durations. For example, the transmission rate in a wireless system might change between scheduling a packet for transmission and the actual transmission due to variations of the wireless channel.
ii) Assuming again a componentized software structure of the simulation model, the component creating an expanded event does not know which event duration the receiving component would assign to the event. For instance, a sending node in a network cannot know how long it takes a receiving node to process an incoming packet – and in fact it should not know for the sake of the model structure. As a result, the event scheduler queries the *receiving* component for the exact event duration before executing the event handler.

Given this integration of event durations in the event scheduling process, we distinguish three approaches to modeling event durations: i) Static durations per event type, ii) static durations per event instance, and iii) dynamic durations per event instance.

Static Duration per Event Type: For all events of the same type, the duration is static and predefined. Examples for such kind of events comprise operations with a constant runtime, for instance generating packets of fixed length.



(a) Our approach: Advancing the simulated time according to the starting times of the events.

(b) Discarded approach: Gradually advancing the simulated time from $t_s(e)$ to $t_c(e)$ may result in jumps backwards in time.

Figure 3.3 Two possible approaches to advancing the global simulated time in expanded event simulation.

Static Duration per Event Instance: The simulation model determines a different duration for each event instance, yet during event execution the duration remains constant. For example, this method allows for assigning different durations to "send-packet"-events according to the size of the packets or a given probability distribution.

Dynamic Duration per Event Instance: Upon handing over an event to the simulation model, it merely determines a *minimum* event duration. During execution of the event handler, the model explicitly (and stepwise) increases the duration, depending on the actual code paths taken. This allows for a maximum degree of accuracy if the duration of the physical process is not known in advance, for instance when modeling the exact runtime of a routing table lookup.

To simplify the notation in the remainder of this thesis, the terms $t_c(e)$ and $t_d(e)$ for an expanded event $e \in E$ always refer to the *final* completion time and event duration, i. e., the final values after dynamic extension, unless stated otherwise.

3.3.3.2 Advancing the Global Simulated Time

A traditional discrete event simulation advances the global simulation time by setting it to the discrete timestamp of the next event being executed (see Section 2.1). During event execution, the global simulated time furthermore remains unchanged. In contrast, in expanded event simulation, events $e \in E$ carry two timestamps, $t_s(e)$ and $t_c(e)$, and span a period of simulated time. Hence, we need to define how to advance the simulated time when executing expanded events. Possible approaches include setting the simulated time to either $t_s(e)$, $t_c(e)$, or even advancing it gradually from $t_s(e)$ to $t_c(e)$. In view of these alternatives, we define the scheme as illustrated in Figure 3.3(a):

Before executing the event handler of an expanded event e, the event scheduler sets the global simulation time to $t_s(e)$. Furthermore, the global time remains constant at $t_s(e)$ throughout the entire wall-clock processing time of e. Thus, despite spanning a period of simulated time, the global virtual clock does not explicitly advance from $t_s(e)$ to $t_c(e)$. From a model developer's perspective, when requesting the current

simulated time from the simulation framework in an event handler, it always returns $t_s(e)$. In addition, model developers can access and dynamically advance $t_d(e)$ inside an event handler.

Despite the possibility to adjust $t_d(e)$, i. e., the duration of an event, we explicitly decide against gradually advancing the *current* global simulated time from $t_s(e)$ to $t_c(e)$ during the execution of an event handler – neither automatically by the simulation framework nor manually by the model developer. Automatically adjusting the simulated time requires mapping the simulated time to the wall-clock execution time of an event handler. In general, however, due to the abstract nature of simulation, no such mapping exists as there is no relation between the instructions of an event handler and the *modeled* physical process.

Still, even if the model developer advances the simulated time manually from within the event handler, overlapping expanded events make the global virtual clock jump back and forth in simulated time as shown in Figure 3.3(b). Consider two overlapping expanded events e_1 and e_2 with $t_s(e_1) < t_s(e_2)$ and $t_c(e_1) < t_c(e_2)$. If the *sequential* simulation first executes e_1, the simulated time would range from $t_s(e_1)$ up to $t_c(e_1)$. For the subsequent execution of e_2, the simulated time would need to be set back to $t_s(e_2)$ which is smaller than $t_c(e_1)$. This in turn contradicts the definition of a causally correct simulation in which the time advances only monotonically.

For the same reason, we refrain from using the completion time of expanded events as a basis for the simulated time. If, for instance, the completion time of e_2 precedes the completion time of e_1, advancing the simulated time first to $t_c(e_1)$ and then to $t_c(e_2) < t_c(e_1)$ again contradicts causal correctness.

3.3.3.3 Event Ordering in the Future Event Set

As pointed out in Section 2.1, discrete event simulation sorts events in the FES in increasing order according to their timestamp. Since an expanded event $e \in E$ is defined by two timestamps, $t_s(e)$ and $t_c(e)$, sorting can utilize either timestamp, combinations of both, or even the event duration $t_d(e)$. Taking the considerations of the previous section into account, our choice is to use the starting time $t_s(e)$ as relevant sorting key. Moreover, similar to traditional discrete event simulation, the FES is sorted according to increasing timestamps, i. e., starting times.

Note that in practice (discrete/expanded) event simulation frameworks apply additional sorting keys as tie breakers between events with equal (starting) timestamps. These tie breakers, e. g., the insertion order into the FES, user defined priorities, or event IDs, are needed to achieve a deterministic event ordering. In the remainder of this thesis we abstract from these tie breakers.

Based on these definitions, the sequential execution model of an expanded event simulation is as follows:

Definition 8 (Sequential Execution Model of Expanded Event Simulation)
The event scheduler continuously dequeues the first event e from the FES, sets the global time to $t_s(e)$, executes the associated event handler, and inserts newly created successor events in the FES.

Procedure: SequentialEventScheduler()
1: **while** $F \neq \emptyset$ **do**
2: e = first event in F
3: global time = $t_s(e)$
4: determine [minimal] $t_d(e)$
5: execute e [and update $t_d(e)$]
6: enqueue newly created events e' in F
7: **end while**

Algorithm 1 Sequential event execution model of expanded event simulation. The terms in square brackets refer to the case of dynamic extension of event durations.

Algorithm 1 illustrates the sequential event execution model in a more formal manner. Based on this formal representation we argue that sequential expanded event simulation obeys the causality constraint and is hence causally correct.

3.3.3.4 Causal Correctness

Based on the previous definitions, we now show that the sequential execution model of expanded event simulation fulfills the causal correctness criterion stated in Section 2.3.2: A simulation is causally correct if the simulated time increases only monotonically. In the following, we present an informal proof based on the three key properties of expanded event simulation.

i) Analogously to discrete event simulation, the event scheduler in sequential expanded event simulation continuously removes the first event $e \in F$ from the FES for execution. We specified in Section 3.3.3.3 that the FES in expanded event simulation orders events according to their starting time. Hence, the event scheduler in fact dequeues the event e with the *smallest starting time* among all events $e' \in F$, i. e., $t_s(e) \leq t_s(e'), \forall e' \in F$.
ii) We defined in Section 3.3.3.2 that sequential expanded event simulation sets the global simulated time to the starting time $t_s(e)$ of the event $e \in E$ that is currently being processed. As shown in i), the scheduler selects the event with the smallest timestamp from the FES. Thus, the simulated time is set to the smallest timestamp of *all* events currently in the FES.
iii) Definition 7 specifies that for all events $e \in E$, the starting time $t_s(e')$ of all successor events $e' \in E$ with $e \curvearrowright e'$ can only be larger than the completion time $t_c(e)$ of the parent event e. Hence, the starting time of all successor events is larger than the current simulated time. Consequently, the scheduler cannot set the simulated time to a value preceding the current simulated time.

From these observations, we conclude that the simulated time advances only monotonically. As a result, sequential expanded event simulation is causally correct.

3.3.4 Parallel Expanded Event Execution Model

In this section, we extend the sequential event execution model to a parallel one. To this end, we first analyze the impact on the event handling process resulting

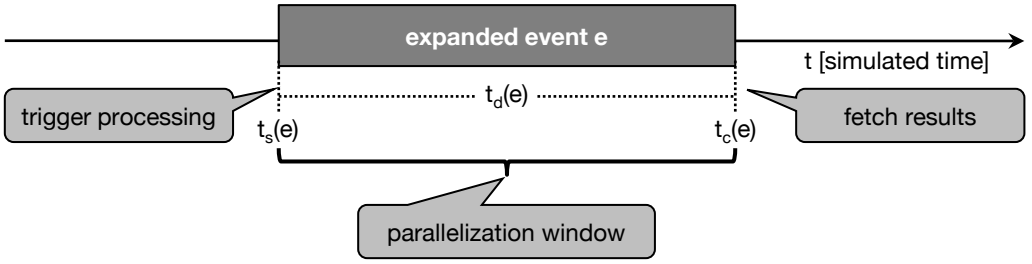


Figure 3.4 Execution scheme of an expanded event e: The results of a simulated continuous task are not needed in the simulation before $t_\mathsf{c}(e)$. Hence, the simulation scheduler offloads e to a worker CPU at $t_\mathsf{s}(e)$ and fetches the results at $t_\mathsf{c}(e)$, allowing other events to be processed in-between.

from associating durations to events. Based on this analysis, we specify the parallel expanded event execution model and finally prove its correctness.

As stated in Section 3.3.1, an expanded event e represents a physical process starting at $t_\mathrm{s}(e)$ and ending at $t_\mathrm{c}(e)$. We assume that the output of such a physical process is neither available nor visible to the entire system before the completion of the process. For instance, the result of a computation is only available for further use after it is complete, a network packet is only available for further processing after it was entirely received and stored, etc. Based on this reasoning, we define that all *overlapping* expanded events, i. e., all events e' starting between $t_\mathrm{s}(e)$ and $t_\mathrm{c}(e)$ of an expanded event e, are independent of e. We argue that an overlapping event e' cannot depend on e because the input of e' cannot include the output of e which is not yet available at $t_\mathrm{s}(e') < t_\mathrm{c}(e)$. As a result, the interval between $t_\mathrm{s}(e)$ and $t_\mathrm{c}(e)$ naturally opens a window for parallelization as shown in Figure 3.4.

In terms of event processing, this means that when the global simulated time reaches $t_\mathrm{s}(e)$ for a given expanded event e, the simulation framework begins executing e. Specifically, the simulation kernel *offloads* e for parallel processing to an available processing unit and continues handling further events. When reaching $t_\mathrm{c}(e)$ in simulated time, the results of e are available and needed in the model. Thus, the simulation blocks at $t_\mathrm{c}(e)$ and waits for the processing unit to finish executing the offloaded event. Figure 3.5 illustrates the resulting parallel event execution model visually.

We formally define overlapping expanded events as:

Definition 9 (Overlapping Expanded Events)
Two expanded events e_1, e_2 overlap in simulated time, denoted by $e_1 \parallel e_2$, if and only if the duration intervals intersect:

$$e_1 \parallel e_2 \Leftrightarrow [t_\mathsf{s}(e_1); t_\mathsf{c}(e_1)] \cap [t_\mathsf{s}(e_2); t_\mathsf{c}(e_2)] \neq \emptyset.$$

Finally, we state the central modeling paradigm of parallel expanded event simulation:

Definition 10 (Parallel Processing of Overlapping Expanded Events)
If two expanded events overlap in simulated time, they are considered to be independent and can hence be executed in parallel.

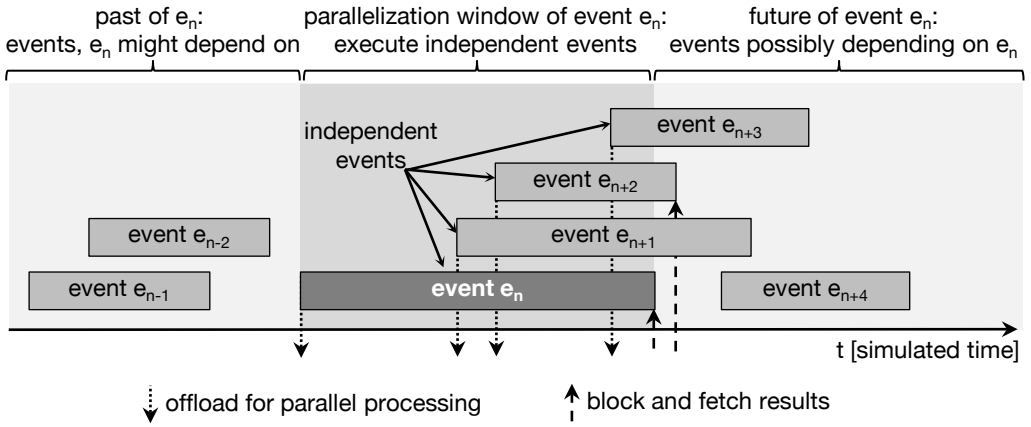


Figure 3.5 Parallel event scheduling: The central scheduler advances the global simulation time by iteratively determining independent events, offloading them to worker CPUs and fetching the results of completed events.

This general definition lays the foundation for our parallel simulation framework HORIZON that puts the concept of parallel expanded event simulation into practice. Specifically designed for multi-core systems and considering ease-of-use for model developers, HORIZON employs a multi-threaded parallelization architecture utilizing a centralized event scheduler. Section 3.4 introduces HORIZON and its architecture in more detail, proves the correctness of its parallelization scheme, and finally evaluates its performance. However, before presenting HORIZON, we first discuss techniques for deriving exact values for event durations in the next section.

3.3.5 Determining Event Durations

Extending events with durations inevitably raises the question of how to obtain detailed information about the timing of events. Few physical processes exhibit a duration that is constant or easily computable at runtime. The propagation delay of a signal, for instance, is given by the distance between sender and receiver and the transmission medium. However, the exact duration of a physical process modeled by an expanded event might depend on more dynamic properties as well, such as the processing speed of the simulated hardware and/or the complexity of the simulated software. As a result, accurately calibrated models may become highly platform dependent. However, we argue that hardware-accurate timing is not necessarily needed in any case, in particular considering the abstract nature of simulation. For instance, it might suffice to meet coarse-grained timing constraints imposed by protocol specifications, e. g., timeouts. Moreover, from a parallelization point of view, approximate durations still increase the lookahead and hence improve the performance.

Thus, calibrating an expanded event simulation model is a trade-off between accuracy and calibration effort. In the following, we sketch five existing and well understood techniques for calibrating expanded event simulation models with respect to timing. We characterize each technique in terms of setup effort, accuracy, cost, and applicability and summarize the results in Table 3.1 using the following scores: ++, +, = , –, – –.

	Hardware	Emulation	Simulation Calibration	Protocols	Domain Experts
Setup Effort	–	– –	=	++	++
Accuracy	++	+	=	–	–
Cost	– –	–	+	++	++
Applicability	=	=	–	+	++

Table 3.1 Overview of five techniques utilizable for determining event durations. We compare these techniques in terms of setup effort, accuracy, cost, and applicability (scores: ++, +, = , –, – –).

Measurements on Real Hardware: The most straightforward approach to obtaining timing information about a specific physical process is measuring a particular process on real hardware. This approach obviously provides highly accurate results (++), yet the setup effort is considerable (–) since this technique requires actual hardware, real-world software, and accurate measurement equipment. The latter restricts this technique in two ways: First, the costs for purchasing the equipment are high (– –). Second, since actual hardware is needed, this technique cannot calibrate simulation models of systems that utilize new hardware which does not exist yet. Hence, the applicability is limited to existing systems (=).

Full System Emulation: Full system emulation [Bel05, Law96, MCE+02, TLP05] employs detailed models to exactly mimic the behavior of hardware. These models allow for executing and measuring the processing time of physical processes similar to real hardware, providing accurate results close to real hardware (+). The downside of full system emulation however is the development effort needed to create accurate hardware models (– –) which in turn results in considerable (development) costs (–). As a result, the availability of such hardware models is limited, in particular regarding new hardware platforms, thus limiting the applicability of this calibration technique (=).

Simulation Calibration: Recent research efforts [KDL+05, LAW08, SHC+04] propose (automatic) calibration techniques for simulation models as a lightweight alternative to emulation. Such techniques (semi-)automatically instrument simulation models with functionality to represent the behavior of a given hardware platform in terms of processing delays and energy consumption. For instance, the TimeTossim simulation framework [LAW08] extracts from each line of source code the number of processor cycles required for execution on a simulated hardware platform. Based on this information, the framework injects bookkeeping functionality into the simulation model to advance the simulated time accordingly after passing a line of source code. A further approach [Sch12] aims at calibrating simulation models in terms of resource utilization and runtime requirements. To this end, a runtime profiler collects detailed information about the resource utilization of a real system for selected workload patterns. The resulting profile is then fed to a resource estimator in the simulation model that determines the simulated runtime based on the currently simulated resource utilization.

This calibration process utilizes previously taken measurements, without the need for detailed hardware models. In conjunction with (semi-)automatic model instrumentation, this technique requires a significantly lower setup effort (=) and causes lower costs than emulation or manual calibration based on precise measurements. However, in order to achieve accurate results, these techniques depend on relatively simple hardware whose behavior can be approximated with static instrumentation mechanisms. This fact limits applicability (–) and degrades accuracy (=) in case of complex hardware platforms.

Protocol Specification: Communication protocols specify timing constraints such as minimum and maximum timeout boundaries, waiting periods, or back-off durations. These properties provide a simple means for obtaining timing information when no specific hardware platform is of interest. For instance, in IEEE 802.11 nodes are required to wait for a minimum period, i. e., Short Interframe Space (SIFS), before they are allowed to acknowledge a received packet. Based on this knowledge, durations can be assigned to events such that their timing still conforms to the protocol specification. The technique of extracting timing information from protocol specifications is a well known approach for maximizing the lookahead in distributed simulation [LN02, MB98, MB99]. Since no measurements are required but the calibration bases purely on the protocol standard, setup effort (++) and costs (++) are excellent. Moreover, this technique is well applicable (+) to communication protocols, yet its accuracy is limited (=).

Domain Experts: Lastly, domain experts can manually calibrate simulation models on the basis of their knowledge. This approach obviously demands a great amount of experience and careful judgment, but it also provides significant flexibility. It is thus particularly well suited for the design space exploration process early during development. Concluding, this technique is very well applicable (++) and features low costs (++) and a low setup effort (++). The downside, of course, is limited accuracy (–) due to a lack of actual measurements.

In conclusion, a wide range of existing techniques enable calibrating expanded event simulation models. Model developers should select or combine appropriate techniques on the basis of the required level of accuracy, applicability to the given scenario, and the calibration effort.

3.3.6 Related Work

Previous efforts in network simulation research also extend classic discrete event simulation with additional timing information to enhance simulation performance and scalability. We review these efforts in the following.

3.3.6.1 Bounded Lag, Propagation Delays, and Opaque Periods

Lubachevsky [Lub88] combines three techniques to improve performance in conservatively synchronized parallel simulation. The first concept is *bounded lag*, denoting

that two events can be processed in parallel if their timestamps are within a known and bounded time window. The second technique comprises the concept of minimum propagation delays. It bases on the observation that the propagation delay of events through a simulated system defines a minimum delay which determines at which time in the future one simulated entity can influence another one. The generalization of this concept is referred to as lookahead throughout this thesis. Lubachevsky combines these two concepts by letting the sum of the propagation delays define the upper bound for the bounded lag.

The third technique, denoted *opaque periods*, is closely related to the idea of expanded events. An opaque period defines a period in simulated time in which a simulated entity "promises" not to generate events. Thus, opaque periods are similar to expanded events in the sense that the starting times $t_s(e')$ of all events e' generated by an expanded event e must begin after $t_c(e)$ (cf. Definition 7) because the results of a physical process are only visible after its end. Due to the fact that opaque periods are not bound to the concept of physical processes, they constitute a more abstract concept. As a result, Lubachevsky needs to manually apply the concept of opaque periods to a given simulation model. In contrast, we deeply embed our approach of expanded events into an intuitive modeling paradigm which fosters the inclusion of expanded events, or opaque periods, by the model developers.

3.3.6.2 Temporal Uncertainty

Fujimoto [Fuj99a] replaces the accurate timestamps of discrete events with time intervals representing a period in simulated time in which an event can occur. This approach is based on the observation that real-world events do not necessarily occur at a fixed and predetermined point in time, but the actual time of occurrence is subject to uncertainty. Thus, in a simulation implementing this approach, an event can occur at any discrete time within the *uncertainty interval*. In the case of overlapping intervals, events can occur in different orders, depending on the actual point in simulated time they take place. The key conclusion is that since the exact order of events is uncertain, they can be processed in parallel. Based on this argumentation, Fujimoto defines an approximate-time partial-order among events along with corresponding synchronization algorithms. However, approximate-time partial-ordering introduces inaccuracies in the simulation results and limits determinism and the repeatability of simulations.

Loper et al. [LF00, LF04] extend this concept by choosing discrete timestamps from uncertainty intervals according to a random distribution. This approach attempts to find a compromise between approximate-time partial-order and traditional discrete timestamp based simulation. Choosing discrete timestamps from uncertainty intervals increases the available lookahead while allowing to re-use existing discrete timestamp based synchronization algorithms. It hence solves the problem of limited repeatability and determinism inherent to approximate time partial ordering.

In contrast to approximate time and Loper's approach, HORIZON uses time intervals to model the duration of a physical process. The respective starting and completion times of expanded events occur at deterministic points in simulated time, thereby guaranteeing repeatability of the results.

3.3.6.3 Interval Branching

Peschlow et al. [PML08] pick up the idea of uncertainty intervals and investigate the effects of different event orderings resulting from overlapping intervals. To avoid executing one individual simulation run for every possible interleaving of events, the authors propose *interval branching*. In this approach, a single simulation run branches for each possible interleaving of overlapping uncertainty intervals. For example, given the events e_1 and e_2 in overlapping uncertainty intervals, interval branching creates one branch in which e_1 precedes e_2 and one branch in which e_2 precedes e_1. Thus, interval branching spans an execution tree representing all possible interleavings. The key performance improvement of interval branching over executing individual runs for each event interleaving is that equal event interleavings in the individual runs collapse to a common path in the tree which is executed only once. From an implementation perspective, the branching operation relies on logical processes and simulation cloning [HF97, HF01] techniques developed for distributed simulation.

For a simple airline simulation, the authors report a speedup of up to 10 on a four-processor computer for the branching approach in comparison to the time needed to conduct the corresponding number of single simulation runs. However, the branching approach suffers from a state explosion problem. Considering accurate simulations comprising millions of events, creating a branch for each possible event interleaving can easily exceed the available memory resources. Hence, interval branching is not generally applicable to complex simulation models.

Previous work [PM07] by the authors in this direction follows a similar branching approach. In contrast to the work outlined above, this earlier work considers only different execution orders of events with *equal* discrete timestamps. Even in this simpler scenario, branching suffers from the same state explosion problem, limiting its applicability to simulation of simple systems.

3.3.7 Summary

This section introduced parallel expanded event simulation as a novel modeling paradigm. Based on the observation that physical processes take time to complete, expanded events extend discrete events by spanning a period of simulated time. We specify a sequential and a parallel event execution model, the latter based on the definition that overlapping expanded events are independent, hence allowing for parallel execution. Finally, we discussed techniques for calibrating simulation models in terms of event durations and reviewed related work.

3.4 The Horizon Simulation Framework

This section introduces Horizon, a parallel simulation framework that puts the concept of parallel expanded event simulation into practice. Horizon targets ubiquitous multi-core workstations and server systems to fully utilize their processing for simulation. Under this premise, the next sections present the architecture of

Horizon in greater detail and discuss the relevant design decisions. In particular, we illustrate the event synchronization scheme of Horizon, prove its causal correctness and briefly sketch implementation details. We then qualitatively compare Horizon to related efforts and quantitatively evaluate its performance by means of synthetic and real-world simulation models.

3.4.1 Centralized Parallelization Architecture

A primary goal of this thesis is to make the parallel processing power of modern multi-core systems available to researchers. The key challenge in this context is to provide a parallelization framework that is simple to use and tailored to the properties of typical simulation models. We address this challenge by proposing a centralized parallelization architecture consisting of a single FES and a global event scheduler. In the following sections, we motivate and discuss the centralized parallelization architecture of Horizon. Specifically, we discuss the fundamental components of parallel simulation as introduced in Section 2.3, namely partitioning, synchronization, and causal correctness.

3.4.1.1 Design Goals

Besides integrating the concept of parallel expanded event simulation in a practical simulation framework, the centralized parallelization architecture of Horizon is a direct consequence of three design goals:

Utilizing Multi-core Systems: Multi-core systems have become the de facto standard hardware for desktop and workstation computers. As a result, model developers and researchers can benefit from parallel simulation directly at their workplace without investing the effort of porting simulations to specialized clusters or compute servers. It is thus the first goal of Horizon to make the parallel processing power of these small to medium scale workplace computers available to model developers. Due to the limited number of processing cores, typically ranging between 4 and 32, a centralized architecture is feasible. Still, there is an upper limit on the total number of processing cores that can be efficiently utilized with a centralized architecture as discussed in Section 3.6.

Simple Usage: In line with the argumentation above, this thesis aims to foster parallel simulation in day-to-day research. Consequently, to achieve a widespread adoption of parallel simulation, it has to be simple to use. Thus, the second goal of Horizon is to eliminate the complexities of parallel simulation such as load balancing and partitioning of simulation models. A centralized approach enables automatic load balancing without the need for partitioning the model.

Basis for Novel Synchronization Schemes: Finally and most importantly, we exploit the centralized parallelization architecture for exploring novel approaches to event synchronization. Synchronization algorithms exploit knowledge of the simulation state to decide which events can be processed in parallel. We argue

that more detailed knowledge allows for making better synchronization decisions, resulting in improved performance. Thus, the third goal of HORIZON is to establish a basis for novel synchronization schemes by providing a maximum amount of state and dependency information to the event synchronization algorithms. multi-core systems provide shared memory, thereby enabling the synchronization algorithm to obtain global knowledge of the simulation model and its state. Thus, in a centralized architecture, consisting of a single FES and a central event scheduler, the entire state of the simulation is directly available to the synchronization algorithm. Based on this property, we develop novel synchronization schemes later in this thesis, namely probabilistic synchronization (cf. Chapter 4) and GPU-based multi-level parallelization (cf. Chapter 5).

In comparison to a traditional framework employing LPs, a centralized architecture sacrifices scalability, of course. We discuss this and other limitations of our approach in Section 3.6 after introducing HORIZON in detail.

3.4.1.2 Partitioning

As outlined in Section 2.3.1, partitioning schemes distribute the workload of a simulation model across the available processing units. The key is to achieve an even workload distribution to avoid idle times and maximize efficiency. Aiming for general applicability, we focus on space-parallel partitioning and discard too specific schemes such as time- and channel-parallel partitioning.

Space-parallel partitioning assigns (groups of) components of a simulation model to processing units. However, the workload inflicted by the events executed on those components is highly heterogeneous (cf. Figure 2.2). As a result, partitioning a given simulation model is a difficult task that requires deep knowledge of event complexities and the distribution of events. Thus, manually partitioning a given model is a considerable additional effort a model developer has to invest to ensure efficient parallel simulation.

Moreover, manual partitioning results in static assignments which are not well suited for *dynamic* systems. To illustrate this, consider a simulation model of a cellular network for example. The simulation model consists of base stations and mobile devices, each associated to exactly one of the base stations. A straightforward partitioning assigns each cell, comprising a base station and all associated mobile devices, to one processing unit, assuming an even distribution of mobile devices. However, due to mobility, mobile devices may leave the range of one base station and associate to another base station, thereby shifting the workload.

In contrast, dynamic partitioning, i. e., load balancing, aims at maintaining an equally distributed workload by adapting the assignment of components to processing units at runtime [PHM07]. To this end, the simulation framework continuously measures the workload and idle times of the processing units and migrates simulated entities between partitions accordingly. However, the measuring infrastructure as well as the migration process add to the complexity and the overhead of the simulation framework, and re-assigning entities might have a negative impact on the lookahead.

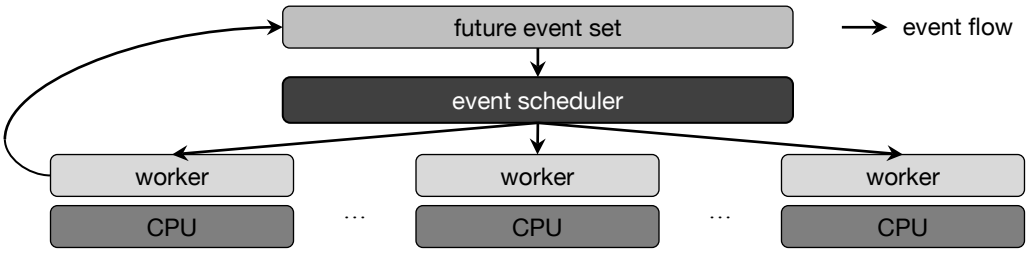


Figure 3.6 Overview of the multi-threaded master-worker architecture of HORIZON. The event scheduler retrieves events from the single FES and distributes independent events to the worker threads.

HORIZON addresses these issues by means of a multi-threaded master-worker architecture that avoids partitioning altogether. Instead of splitting the FES according to partitions, HORIZON retains a single global FES. A central event scheduler thread continuously dequeues events from the sole FES and distributes independent events to worker threads for parallel processing. As a result, the workload of a simulation model is evenly and automatically distributed across all available processing units, thereby eliminating the need for an explicit load balancing mechanism. Figure 3.6 shows an overview of the resulting architecture. This architecture is a direct consequence of the fact that HORIZON specifically targets multi-core systems. Such systems provide a global shared memory space across all processing units and threads, thus enabling any worker to handle any available independent event.

Despite abandoning partitioning, simulation models in HORIZON exhibit a componentized structure. Besides common practice in software engineering, componentization is imperative for ensuring data consistency in our multi-threaded simulation framework: In shared-memory parallel simulation, event handlers are able to change the state of the entire simulation model. Thus, when executing events in parallel threads, the corresponding event handlers should not read from and write to the same state variables to avoid race conditions which result in an inconsistent state. Hence, we employ componentization to encapsulate the state of the simulation model locally within components. An event handler is thus only allowed to modify the state local to the component it belongs to. If an event needs to change the state at a remote component, it has to create a new event that takes place at this particular component. In this regard, components in HORIZON are similar to logical processes in traditional parallel discrete event simulation.

To provide model developers with a maximum of flexibility, HORIZON allows for scheduling overlapping events that take place at the same component. In this situation, the event scheduler in HORIZON ensures that within each component only one event is active at a time. Specifically, HORIZON orders the execution of overlapping events at a common component in starting time order. This follows naturally from the fact that the scheduler always dequeues the first event from the FES which is sorted by starting times.

3.4.1.3 Synchronization

Based on the parallel expanded event execution model introduced in Section 3.3.4 and the centralized architecture of Horizon presented in the previous section, we now design a corresponding synchronization algorithm.

Conservative vs. Optimistic Synchronization

The most essential design decision in this context is whether the synchronization algorithm follows either a conservative or an optimistic approach. Our design constitutes a conservative synchronization algorithm for the following reasons.

Improved Lookahead: The primary purpose of event durations is to increase the lookahead within a simulation model. As the lookahead is of utmost importance to conservative synchronization, our synchronization scheme needs to be conservative to allow for evaluating the improvements of expanded event simulation over traditional parallel discrete event simulation.

Simplicity: Conservative synchronization is significantly simpler than optimistic synchronization. For instance, conservative synchronization does not require additional memory management to enable checkpointing and rollbacks or specific compiler support to facilitate reverse computation. Additionally, conservative synchronization utilizes fewer resources, in particular memory, thereby contributing more resources to the actual simulation model.

Despite favoring conservative synchronization, this scheme is inherently limited by the requirement to strictly avoid causal violations at runtime. Thus, we present a probabilistic synchronization scheme in Chapter 4 that dynamically switches between conservative and optimistic synchronization to combine the best of both worlds while eliminating the respective drawbacks.

Barrier-based Event Synchronization

The design of our conservative synchronization scheme is based on the following reasoning. Recall that in parallel expanded event simulation overlapping events are independent. Hence, the event scheduler continuously dequeues the first event $e \in F$, checks whether or not it overlaps with previously offloaded events and if so, hands it to a worker for parallel processing. Conversely, the scheduler does not immediately offload e if it does not overlap with *all* previously offloaded events. Instead, it waits for the execution of the offloaded events to finish. As a result, the minimum completion time among all offloaded events determines an upper bound, i. e., a *barrier*, for overlapping events. Thus, the synchronization algorithm coordinates parallel event execution by maintaining a barrier computed over all offloaded events. Deciding whether or not an event is offloadable hence boils down to checking if its starting time precedes the barrier.

In order to formally state the event synchronization scheme, we first define the set O of all currently offloaded events.

Procedure: ParallelEventScheduler()

```
1: shared variables: F, O, t_b
2: t_b := ∞, O := ∅
3: while F ∪ O ≠ ∅ do
4:   if F ≠ ∅ then
5:     e := arg min_{e∈F}(t_s(e))
6:     if t_s(e) ≤ t_b then
7:       O := O ∪ {e}, F := F\{e}
8:       determine [minimal] t_d(e)
9:       t_b := min{t_b, t_c(e)}
10:      offload(e)
11:    else
12:      wait for t_s(e) ≤ t_b
13:    end if
14:  end if
15: end while
```

(a) Central Event Scheduler

Procedure: ParallelWorker()

```
1: shared variables: F, O, t_b
2: while true do
3:   e := getNextOffloadedEvent()
4:   execute e [and update t_d(e)]
5:   O := O\{e}
6:   update t_b
7: end while
```

(b) Worker Thread

Algorithm 2 Parallel scheduling of expanded events.

Definition 11 (Set of Offloaded Events)
The set $O \subseteq E$ contains all currently offloaded expanded events, i. e., all overlapping expanded events being executed concurrently on all processing units. The sets F and O are mutually exclusive, i. e., $O \cap F = \emptyset$.

Based on O, we specify the synchronization barrier t_b as follows:

Definition 12 (Synchronization Barrier)
The synchronization barrier $t_\mathrm{b} \in T \cup \{\infty\}$ is the minimum completion time of all events in O or infinity if O is empty:

$$t_\mathrm{b} = \begin{cases} \min\{t_\mathrm{c}(e) | e \in O\} & , O \neq \emptyset \\ \infty & , \text{otherwise} \end{cases}$$

Algorithm 2 gives a formal definition of the barrier-based synchronization scheme, separated into the functionality of the event schedulers and the workers. The algorithm uses pseudo code to describe the fundamental principles. It hence abstracts from technicalities such as thread synchronization or inter-process communication:

Scheduler: Initially, no events are offloaded, hence, O is empty and the barrier t_b is set to infinity (Line 2). A simulation run proceeds as long as there is at least one event in either F or O (Line 3). We explicitly check if $F \neq \emptyset$ (Line 4) to wait (by means of busy waiting in the while loop) for events in O to either finish processing (ending the simulation) or to insert new events in F. At runtime, the event scheduler continuously dequeues the first event from F (Line 5) and checks if its starting time precedes the barrier (Line 6). If so, the scheduler prepares offloading of the event by updating the sets O and F, determining the event duration as described in Section 3.3.3.1, computes a

new barrier considering the newly offloaded event, and finally hands the event to a worker (Lines 7-10). Otherwise it waits for a worker to update the barrier after processing its event (Line 12).

Worker: The workers continuously retrieve a previously offloaded event (Line 3) and execute its event handler (Line 4). After executing the event handler, they update the set O (Line 5) and modify the barrier to reflect the new state of O. These updates eventually allow a blocked event scheduler to proceed (Line 6).

3.4.1.4 Causal Correctness

In this section, we prove the correctness of the parallel event scheduling and synchronization algorithm. We need to show that the algorithm guarantees causal correctness, i. e., non-independent events are processed only in increasing starting time order. To this end, we adopt the definition of causal correctness from Definition 1 and modify it to match parallel expanded event simulation according to Definition 10:

Definition 13 (Causal Correctness of Parallel Expanded Event Simulation)
A parallel expanded event simulation obeys the causality constraint if and only if each pair of non-overlapping events is processed in non-decreasing starting time order.

We prove the correctness of the synchronization scheme along this line of reasoning: First, we show that no events with starting times preceding the barrier are inserted into F (Lemma 1). By means of this lemma we show that the scheduler handles events in increasing starting time order (Lemma 2). Note that this does not imply that events are executed by the worker threads in increasing starting time order. We then show that only overlapping events are executed in parallel (Lemma 3). Finally, by applying Lemma 2 and 3 we proof the causal correctness property of our event scheduling algorithm.

Lemma 1
No event $e' \in E$ with $t_\mathsf{s}(e') < t_\mathsf{b}$ is inserted into F by another event $e \in E$.

Proof. Based on Definition 7, no event $e \in O$ can insert another event $e' \in E$ into F with $t_\mathrm{s}(e') < t_\mathrm{c}(e)$. Since t_b is the minimum over the completion times of all offloaded events, i. e., $t_\mathrm{b} = \min\{t_\mathrm{c}(e) | e \in O\}$ (since $e \in O \neq \emptyset$), it follows that $\forall e \in O : e$ cannot insert an event e' into F with $t_\mathrm{s}(e') < t_\mathrm{b}$. This property also holds for all $e \notin O$ as a non-offloaded event is not executed and hence cannot create new events. Concluding, no new event preceding t_b can be inserted into F. □

Lemma 2
The central event scheduler handles events in increasing starting time order.

Proof. By contradiction. Assume two events $e_1, e_2 \in E$ with $t_\mathrm{s}(e_1) < t_\mathrm{s}(e_2)$, but the scheduler handles e_2 even if it did not handle e_1 before. For all possible combinations of e_1 and e_2 in F, we derive a contradiction from this assumption:

Case 1: $e_2 \notin F$
The scheduler cannot handle e_2 because it is not in the FES. Contradiction.

Case 2: $e_2 \in F, e_1 \in F$
Because of the ordering constraint of F and the fact that the scheduler only removes the event with the smallest starting time from F (Algorithm 2(a), Line 5), it first handles e_1 and then e_2. Contradiction.

Case 3: $e_2 \in F, e_1 \notin F, t_\mathrm{b} < t_\mathrm{s}(e_2)$
The scheduler does not handle e_2, but it blocks at t_b, because the condition $t_\mathrm{s}(e_2) \leq t_\mathrm{b}$ (Algorithm 2(a), Line 6) is not met. Contradiction.

Case 4: $e_2 \in F, e_1 \notin F, t_\mathrm{b} \geq t_\mathrm{s}(e_2)$
The scheduler selects e_2 for processing. However, e_1 cannot be inserted into F afterwards due to Lemma 1. Thus, e_1 was either processed before or it will never be processed. Contradiction.

All possible cases result in a contradiction. Thus, the initial assumption is wrong and the converse is proved. □

Lemma 3
The central event scheduler never offloads non-overlapping events $e, e' \in F, e \nparallel e'$.

Proof. The scheduler offloads an event e to the worker pool for parallel execution only if the starting time of e is smaller than the current barrier t_b:

$$
\begin{aligned}
& t_\mathrm{s}(e) \leq t_\mathrm{b} && (3.1)\\
\Rightarrow\ & t_\mathrm{s}(e) \leq \min\{t_\mathrm{c}(e') | e' \in O\} && (3.2)\\
\Rightarrow\ & t_\mathrm{s}(e) \leq t_\mathrm{c}(e'),\ \forall e' \in O && (3.3)\\
\Rightarrow\ & (t_\mathrm{s}(e) \leq t_\mathrm{c}(e')) \wedge (t_\mathrm{s}(e') \leq t_\mathrm{s}(e))),\ \forall e' \in O && (3.4)\\
\Rightarrow\ & (t_\mathrm{s}(e') \leq t_\mathrm{s}(e) \leq t_\mathrm{c}(e')) \wedge (t_\mathrm{s}(e) \leq t_\mathrm{c}(e)),\ \forall e' \in O && (3.5)\\
\Rightarrow\ & [t_\mathrm{s}(e'); t_\mathrm{c}(e')] \cap [t_\mathrm{s}(e); t_\mathrm{c}(e)] \neq \emptyset && (3.6)\\
\Rightarrow\ & e' \parallel e && (3.7)
\end{aligned}
$$

(3.1) follows directly from Line 6 of Algorithm 2(a). Similarly, (3.2) results from Definition 12 and Lines 9 and 6 in Algorithm 2. (3.3) is a simple logical conclusion from (3.2). We further derive from $e' \in O$ that e' was already handled and offloaded before e. By applying Lemma 2, we conclude that e' exhibits a smaller or equal starting time than e, showing (3.4). The first part of the conjunction in (3.5) is a reformulation of (3.4), the second part results from the simple fact that event durations must not be negative (Definition 5). The ordering of the timestamps in (3.5) shows that the intervals do overlap, resulting in (3.6). Finally, by applying Definition 9, we conclude that e and e' overlap. □

The previous lemmas enable us to show the following theorem:

Theorem 1
The event scheduling algorithm guarantees causal correctness according to Definition 13 by processing each pair of non-overlapping events in non-decreasing starting time order only.

Proof. Assume two events $e, e' \in E$ with $e \not\parallel e'$. Following from Lemma 3, the events are not executed in parallel. Instead, the scheduler offloads them in increasing starting time order according to Lemma 2. Thus, causal correctness is fulfilled for non-overlapping events.

Concluding, the parallel event scheduling algorithm of HORIZON meets the causal correctness requirement. □

3.4.2 Implementation of the Horizon Framework

We now discuss the key properties and design decisions underlying the implementation of HORIZON. At first, we argue for using OMNeT++ as basis for HORIZON, followed by a brief discussion of the integration of HORIZON in OMNeT++.

3.4.2.1 OMNeT++ as Host Simulation Framework

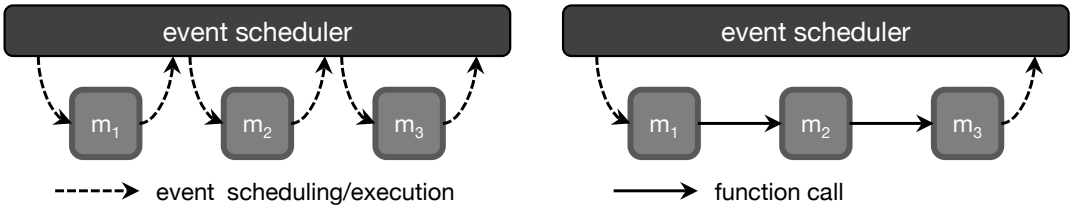
We integrate HORIZON in the existing OMNeT++ [Var01] simulation framework for the following reasons.

Real-world Applicability: It is the goal and the purpose of this thesis to provide a parallelization framework that is attractive for use in practice. However, widespread usage of a simulation framework depends on the availability and quality of compatible simulation models. Since it is neither in the scope of this thesis nor reasonable to re-implement a large base of complex simulation models, our parallelization framework instead must be (largely) compatible with existing models. The OMNeT++ community maintains such a large base of simulation models [omnb, KSW+08].

The downside of building on top of an existing simulation framework is that integrating multi-threaded expanded event simulation creates non-trivial technical challenges. For instance, although OMNeT++ supports traditional distributed parallel simulation, it is not designed for *multi-threaded* simulation. As a result, enabling thread-safe parallel event execution requires thorough analysis and tedious modification and verification of the simulation core.

Code Availability: Implementing parallel expanded event simulation in an existing simulation framework involves extensive changes of the core and Application Programming Interface (API) of the host simulation framework. Thus, we require access to the source code in order to modify it and make it available to the research community. These requirements obviously rule out commercial, closed source simulation tools. In contrast, the source code of OMNeT++ is publicly available.

Model Structure: In addition to OMNeT++, the open source network simulators ns-2 [MF99] and ns-3 [HRFR06] match the criteria stated above. Although those frameworks may very well serve as basis for HORIZON, the structure of the models in OMNeT++ suits our parallelization and modeling paradigm better.



(a) OMNeT++: Consecutive state changes at modules are modeled by means of *individual* events.

(b) ns-3: Consecutive state changes at modules are modeled by means of *one* event and subsequent function calls between modules.

Figure 3.7 Comparison of the event execution schemes of OMNeT++ and ns-3.

Simulation models in OMNeT++ exhibit a highly modular structure of much finer granularity than the models of ns-2 or ns-3. So called *modules* constitute the atomic building blocks of models in OMNeT++ and encapsulate functionality on the granularity of protocols, network cards, or routing tables, for instance. Interaction between modules relies on events, called messages, which traverse the event scheduler and hence allow for a fine-grained control of parallel event execution (see Figure 3.7(a)). In contrast, simulation models in ns-3 compose functionality, e. g., protocol stacks, by means of *function calls* which do not interact with the event scheduler.

This has two consequences: First, event handling in ns-3 is more efficient by reducing the number of events, but this restricts the influence of the event scheduler and hence the parallelization scheme. Second and most importantly, events do not represent individual physical processes, but a *chain* of processes (see Figure 3.7(b)), for instance a network packet traversing the protocol stack. This modeling approach does not well suit the modeling paradigm underlying expanded event simulation if there exist multiple paths through the processes of a simulation model. In case of multiple possible paths, the exact type and order of the processes actually traversed is not known in advanced, thereby making it difficult to a priori associate durations with the entire chain of processes.

We observe that multiple paths occur in a simulation for instance when a network packet is multiplexed to the correct protocols when traversing the protocol stack in upward direction, or when a packet is either correctly or incorrectly received. Still, by dynamically updating the duration of a currently executing expanded event according to the current physical process, ns-3 is in principle able to support parallel expanded event simulation as well. Finally, we decided against using ns-2 for similar reasons as well as the fact that it is now succeeded by ns-3.

3.4.2.2 Integrating Horizon in OMNeT++

In order to foster wide-spread use of Horizon, we aim for a largely transparent integration of Horizon with OMNeT++. For this reason, Horizon is backwards compatible with OMNeT++, hence enabling a convenient transition to duration based modeling. However, in order to make use of expanded events, the model must be ported to Horizon under consideration of the following two aspects.

Event Durations: Model developers need to assign durations to expanded events. To this end, the simulation model can provide one new method per module that determines the duration of a given event and returns it to the event scheduler (cf. Section 3.3.3.1). If a simulation model does not implement this method, the duration defaults to zero, hence rendering the expanded event a discrete event for backwards compatibility.

Random Number Generation: OMNeT++ utilizes global random number generators which are not thread safe. Locking these generators does not suffice since concurrent worker threads can access the generators in an arbitrary order. This breaks determinism across multiple simulation runs despite using the same random seed. Thus, in Horizon every module employs local random number generators. Access to these generators is deterministic due to the deterministic ordering of events in each module.

Local random number generators do not allow for sharing a common random number generator across modules, yet in practice this is rarely needed. Instead, model developers typically use multiple global random number generators to avoid correlation between random numbers. Hence, this change does not impede model development.

In addition, the same modeling restriction applies as in distributed parallelization using OMNeT++: Event handlers must only modify data which is local to the module they execute on to preserve data consistency. Hence, exchanging data between modules relies on events and global variables are forbidden.

3.4.3 Related Work

This section covers related simulation frameworks targeting multi-threaded parallel simulation. It hence extends the general overview presented in Section 2.4. We sketch the respective parallelization techniques of the related frameworks and compare them to Horizon.

3.4.3.1 ns-3

In the context of the ns-3 project [HRFR06], Seguin [Seg09] implemented a multi-threaded extension of the ns-3 simulation engine. Similar to Horizon, this work is driven by the motivation to develop a parallel simulation framework that provides model developers with efficient parallel execution on multi-core computers.

The architecture employs both the Null Message Algorithm (NMA) and a synchronous barrier-based algorithm for conservative event synchronization. Due to the time-creeping problem inherent to the NMA however, Seguin prefers the barrier-based algorithm for synchronization. Both synchronization schemes derive the lookahead from link delays. However, the framework only supports simple point-to-point links with static delays, corresponding to wired connections. In terms of partitioning, the framework handles each network node as a separate partition, thereby eliminating the need for manual partitioning and simplifying load balancing. Building on

this partitioning scheme, the framework trades off thread contention and workload balancing by assigning a subset of the partitions exclusively to worker threads while the remaining partitions are shared among workers.

The project dedicates considerable efforts on reducing the overhead of thread synchronization. This includes synchronization barriers optimized for multi-core computers as well as mechanisms for thread-safe reference counting, which constitutes a fundamental programming paradigm in the ns-3 architecture. The project reports a 20 % performance increase on the DARPA NMS Campus Network model [Nic03] using an eight-core computer. The authors blame the limited performance improvement on the overhead due to locking within the simulation framework. As a result, research on multi-threaded parallelization is discontinued in the ns-3 project, focusing instead on traditional distributed simulation over MPI [BBC+12].

3.4.3.2 HiPWiNS

HiPWiNS (High Performance Wireless Network Simulator) [PVM09] is a multi-threaded simulation framework based on JiST/SWANS [BZvR04] aiming for efficient parallel simulation of IEEE 802.11 networks. It partitions the simulation model on a node-level and statically assigns equal numbers of nodes to LPs which map to worker threads. In addition to this traditional partitioning approach, the authors claim two major contributions that enable efficient parallel simulation.

The first contribution is a conservative event synchronization scheme that employs extended timing information, closely related to expanded event simulation, to increase the available lookahead. Specifically, event synchronization is based on a global barrier that takes protocol- and event-lookahead into account. Protocol lookahead, as pioneered by Liu et al. [LN02] estimates the next transmission time of a node and contributes this timestamp to the global barrier computation. Event lookahead is similar to event durations in the sense that it exploits the fact that physical processes span a period of time in which they cannot influence the surrounding system. As basis for the event lookahead, the authors propose using the delay of switching an IEEE 802.11 transceiver from sending to receiving mode (RxTxTurnaround), since during this switch, the transceiver can neither transmit nor receive. The barrier itself is a tournament barrier optimized for global reductions.

The second contribution is called event-bundling and aims at reducing the number of events exchanged between LPs. Particularly, instead of sending a receive event to each receiving node, event bundling only sends one meta-event to each *LP* which in turn generates receive events locally for all associated nodes.

Both Horizon and HiPWiNS focus on wireless networks. However, in direct comparison the concepts underlying Horizon are more general in nature. Specifically, event lookahead is applied only to the RxTxTurnaround duration which in turn is implemented by means of two events that are specially treated in the simulation framework. In contrast, expanded event simulation is a generalized modeling paradigm, thus making the RxTxTurnaround switching delay equally applicable to Horizon. Furthermore, event bundling is only effective when utilizing LPs, yet, the static assignment of network nodes to LPs as exercised in HiPWiNS severely restricts load balancing in contrast to the dynamic approach of Horizon.

3.4.3.3 Other Related Simulation Frameworks

PRIME (Parallel Real-time Immersive network Modeling Environment) [LLH09] is the latest incarnation of the Scalable Simulation Framework (SSF) [CNO99]. The primary focus of PRIME is on achieving real-time simulation as basis for co-simulation, i. e., the interaction of real networking systems with a simulated network. In order to harvest the required processing power needed for large scale networks, it combines multi-threaded simulation with distributed simulation. To this end, the framework utilizes conservative composite synchronization [NL02] in combination with hierarchical synchronization [LN01] to integrate multi-threaded with distributed simulation. In contrast to PRIME, Horizon does not aim for large scale co-simulation, but instead focuses on speeding up small to medium scale simulations on desktop or workstation computers. Hence, architecture and event synchronization of Horizon is considerably simpler than in PRIME.

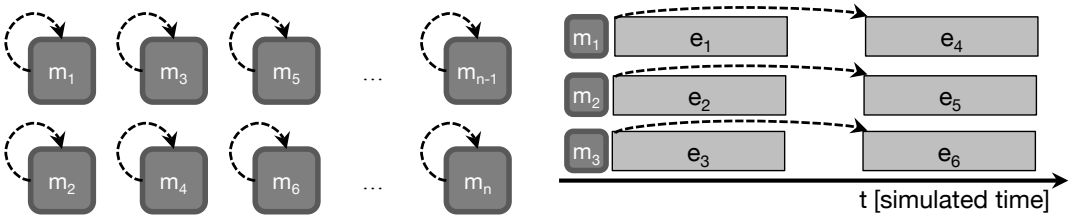
De Munck et al. [MVB10] present a multi-threaded simulator for the evaluation of resource management strategies in computing grids. Similar to the line of argumentation in this thesis, the authors explicitly argue for a parallel simulation framework specifically tailored to multi-core systems. However, the architecture of the simulator closely resembles the architectural properties of a distributed simulator. Most prominently, the simulator makes use of the NMA to synchronize LPs. Thus, although running on shared-memory which allows access to all event queues, the synchronization scheme requires exchanging null-messages as in a distributed scenario. As a result, the simulator unnecessarily suffers from the time creeping problem inherent to the NMA.

3.4.4 Evaluation

We evaluate Horizon in three steps: First, we utilize synthetic benchmarks to characterize the performance properties of Horizon with regard to the number of CPUs and the workload. Second, we compare Horizon to the parallelization capabilities of OMNeT++ which belong to the state-of-the-art in traditional distributed parallelization [Fuj90a, Per06b, SVE03]. Third, we show the applicability of Horizon by analyzing the performance gain in a real-world simulation model of a LTE network.

3.4.4.1 Benchmarking Methodology

Our evaluation of Horizon utilizes two classes of benchmark models, each targeting a unique evaluation goal. The first class comprises purely synthetic benchmarks models. These models do not represent a concrete system, but instead allow for precisely adjusting the workload generated by the model. By investigating a wide range of workload patterns, the synthetic benchmarks span a design space covering many different real-world simulation models. Mapping concrete models into this space allows for deducing their potential for efficient parallelization with Horizon. In contrast, the second class of benchmarks utilizes a concrete model of a cellular 3GPP Long Term Evolution (LTE) network and acts as a case study to confirm the synthetic performance results.



(a) The synthetic model consists of a configurable number of independent benchmark modules that schedule self-events.

(b) Each benchmark module continuously executes and re-schedules one expanded event. All events perfectly overlap.

Figure 3.8 Structure of and event scheduling in the synthetic benchmark model.

Throughout this evaluation, we use the *speedup* [BT02] as a metric to characterize the performance improvement of a parallel simulation over a sequential one. Given the runtimes for a sequential execution $t_{seq}(S)$ and a parallel execution $t_{par}(S)$ of a simulation model S, the speedup is defined as

$$\text{Speedup} = \frac{t_{seq}(S)}{t_{par}(S)}. \tag{3.8}$$

Our evaluation bases on a prototype implementation of HORIZON which builds upon OMNeT++ 4.1. All performance results show average values collected over 30 independent runs and the corresponding 99 % confidence intervals, which are however barely visible. We utilized an AMD Opteron compute server providing 32 GB of RAM and a total of 12 processing cores, organized in two six-core CPUs running a 64-bit Ubuntu 12.04.1 LTS server OS.

3.4.4.2 Performance Characteristics of Horizon

The purpose of this benchmark is to assess the scalability of the centralized architecture of HORIZON. In particular, we investigate the scalability of HORIZON with regard to the number of worker threads and the workload. In terms of workload, the synthetic benchmark model allows for adjusting two parameters: i) the degree of parallelism as well as ii) the computational complexity of the events. To this end, the model consists of a configurable number of independent, i. e., not interconnected, benchmark modules (see Figure 3.8(a)). Each module continuously creates expanded events of specific computational complexity and schedules them for local execution. By maintaining a perfectly synchronous and overlapping timing among the events, we enable parallel execution (see Figure 3.8(b)). Since every module executes one expanded event at a time, we control the degree of parallelism by means of the number of benchmark modules in the model.

Scalability in terms of CPUs

To assess the scalability of HORIZON, we analyze the runtime performance of the benchmark model while varying the number of worker threads and the computational complexity of the events. While the motivation for changing the number of workers is obvious, the event complexity has a less obvious yet equally important influence

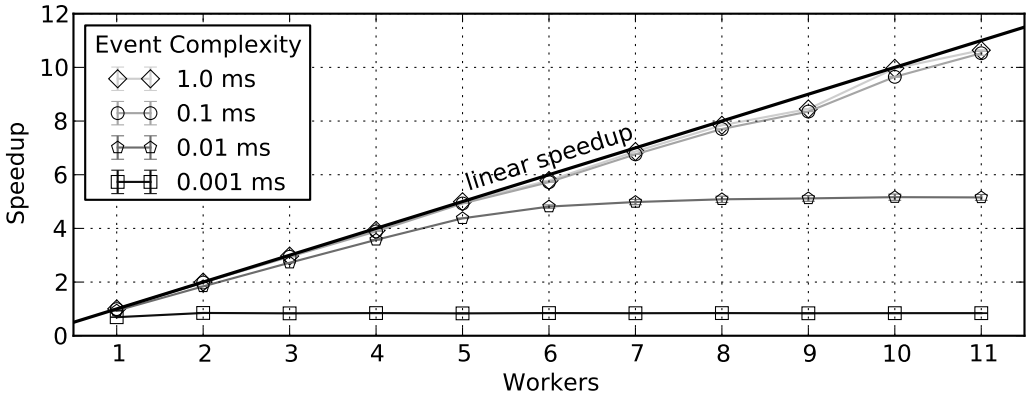


Figure 3.9 Speedup of Horizon in terms of the number of workers and the event complexity using a continuously parallelizable workload of 110 independent events. Horizon achieves a linear speedup for event complexities larger than 0.1 ms, while being limited to a 5-fold speedup with events of 0.01 ms complexity. For computationally insignificant events, Horizon does not deliver a speedup but also avoids a decrease in performance caused by thread contention.

on scalability: Since every event passes through the event scheduler sequentially, the runtime of the offloaded events needs to be long enough to enable the scheduler to offload further events if possible. If the processing time is too short, parallel event execution degrades to sequential execution. Based on runtime performance profiles of publicly available simulation models [mob, KSW+08] and of our own model [NG10] (see Figure 3.13), the event complexity ranges from 1 μs to 1 ms in this benchmark. We furthermore vary the number of workers between 1 and 11. In contrast, the degree of parallelism in this benchmark is fixed to 100 by using a total of 100 benchmark modules. This large degree of parallelism guarantees sufficient parallel workload for keeping the workers busy.

Figure 3.9 shows the resulting speedup for 1 to 11 workers when varying the event complexity from 1 µs to 1 ms. We observe a strong dependency between the speedup and the computational complexities of the events. For event complexities of 1 ms and 0.1 ms, Horizon achieves a speedup that grows linearly with the number of workers. Note that the slight drop in performance for 9 workers results from mapping 110 independent events to 9 workers. To process a set of 110 events, 7 workers execute 12 events while 2 workers handle 13 events. Thus, while just 2 workers process their last event, the remaining 7 workers are idle, resulting in a sub-optimal resource utilization. We observe the same effect also for the other measurements points in which the number of events per round is not divisible by the number of workers, yet it is less pronounced.

When reducing the event complexity to 0.01 ms, we identify a linear speedup for up to 5 workers. Beyond 5 workers, however, the speedup converges to 5. We ascribe this to the fact that the centralized event scheduler handles all events sequentially. In order to offload x events for concurrent processing, the event complexity has to be at least x times the offloading delay. Thus, given an event complexity of 0.01 ms, the scheduler is able to offload only 6 events, i. e., to keep at most 6 workers busy. For the same reason the speedup finally degrades to one when limiting the event complexity to merely 1 µs. In this case the scheduler is only able to offload one event at a time

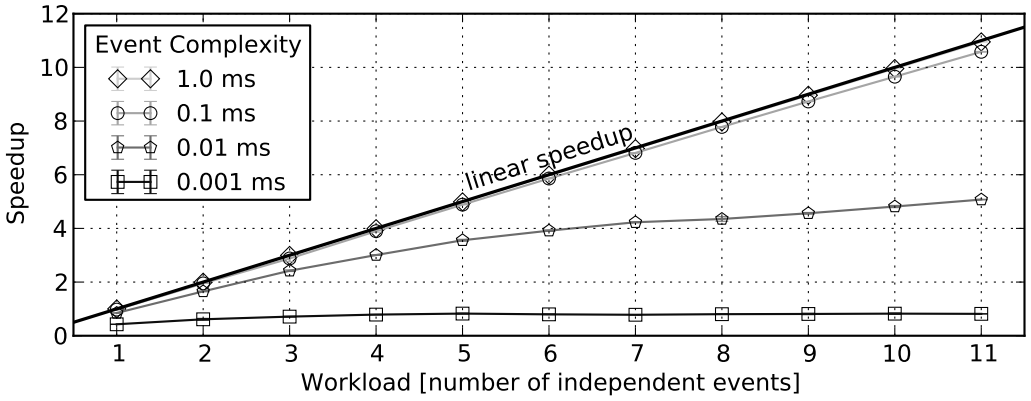


Figure 3.10 Speedup of HORIZON in terms of the parallelizable workload and the event complexity using 11 worker threads.

because the worker finishes handling of the event before the scheduler offloads the next event.

Scalability in terms of Workload

This benchmark evaluates the scalability of HORIZON with regard to the parallel workload given a fixed number of workers, i. e., static thread contention and event handling overhead. Thus, we fix the number of workers to a maximum of 11 while varying the number of benchmark modules and hence the number of parallel events between 1 and 11. As before, the event complexity ranges from 1 μs up to 1 ms.

In Figure 3.10, we observe qualitatively similar results as in the previous benchmark. We nevertheless point out minor differences in the results. Most noticeable, the speedup curves for event complexities of 10 µs and 1 µs increase slower for smaller workloads than in the previous benchmark. We ascribe this to the fact that in this benchmark the number of workers is always larger than or equal to the number of independent events. Hence, some of the worker threads are always idle. These workers cannot contribute to the speedup, but instead utilize system resources and cause synchronization overhead, thereby lessening the simulation performance. This is a consequence of the *push-based event offloading scheme* that utilizes busy-waiting to synchronize the worker threads. The goal of this scheme is to minimize the offloading delay, i. e., the time between the master thread offloads an event for parallel processing and the time a worker thread actually starts processing the event. We present this scheme in more detail in Section 3.5 and show that it considerably improves parallel simulation performance in comparison to a scheme that suspends threads, i. e., without busy-waiting.

Moreover, the performance curves for event complexities of 1 ms and 100 µs show no drops in performance. This, again, is due to the fact that the number of workers exceeds the number of independent events.

3.4.4.3 Comparison with Traditional Parallel Discrete Event Simulation

The goal of this benchmark is to analyze the performance improvement of parallel expanded event simulation over traditional parallel discrete event simulation. To this

Parameter	Value(s)
Number of benchmark modules	100
Number of events per module	10,000
Lookahead/link delay	10 ms - 1 µs
Send/receive event duration	10 ms - 1 µs
Send event interarrival time distribution	exponential, mean 1 ms
Send/receive event complexity	0.1 ms
Number of workers/partitions	11

Table 3.2 Configuration parameters of the synthetic benchmark model for comparing Horizon and OMNeT++.

end, we compare the performance of Horizon with the performance of OMNeT++ utilizing its parallel discrete event simulation capabilities [omna, SVE03]. In order to execute simulations in parallel, OMNeT++ uses the Null Message Algorithm (NMA) [CM79] and space-parallel partitioning. Communication between LPs relies on MPI which in turn exploits shared memory synchronization and interprocess communication when running on multi-core computers. Hence, despite not specifically designed for multi-core systems, the parallelization architecture of OMNeT++ is suited for such systems. Considering moreover the large user-base of OMNeT++ and recent research efforts towards parallel simulation [BRM12, KBV09, SRTR09, SVE03, VSE03], OMNeT++ belongs to the current non-commercial state-of-the-art in terms of parallel simulation.

In addition, comparing Horizon to OMNeT++ allows us to focus on investigating the performance of the parallelization schemes while excluding differences in the structure of the frameworks. Specifically, the simulation core and modeling API of Horizon and OMNeT++ are, by design, nearly identical. This enables us to use one basic benchmark model for both frameworks. In contrast, a comparison between ns-3 and Horizon requires two separate simulation models that differ in structure and run on two different simulation engines. These differences in model structure and event handling naturally affect the performance of the benchmark, thereby preventing a precise comparison of parallel expanded event simulation with traditional parallelization. Hence, we refrain from a comparison with ns-3.

Benchmark Model

The key performance factors of expanded and discrete event simulation are the lookahead and event durations. Thus, we extend the synthetic benchmark to model these properties as follows: First, we interconnect all benchmark modules via links with configurable delay since traditional parallelization in OMNeT++ derives the lookahead from link delays. These links form a fully meshed topology, allowing benchmark modules to send abstract packets to randomly selected neighbors.

Second, we add “send” and “receive”-processes spanning a period of simulated time to each module. In Horizon, we model these processes by means of expanded events while the OMNeT++-model resorts to using discrete start and end-events

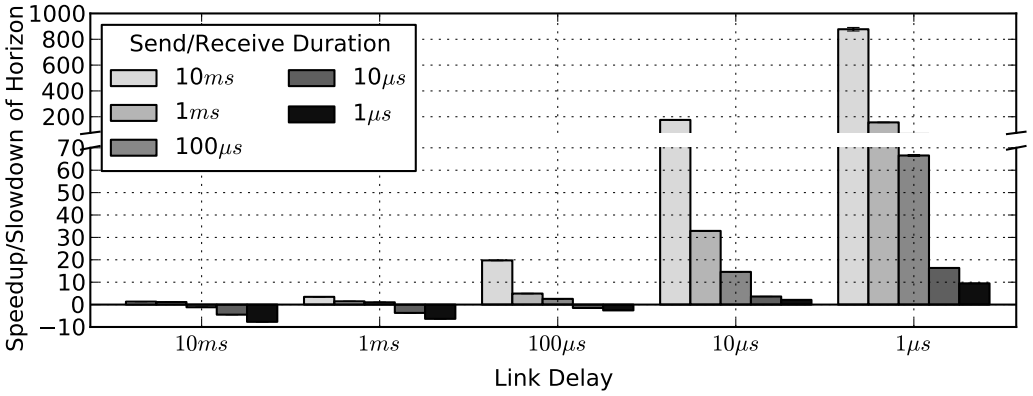


Figure 3.11 Illustration of the performance difference between HORIZON and the Null Message Algorithm (NMA) as implemented in OMNeT++. The NMA slightly outperforms HORIZON for large link delays and small event durations. For small link durations, however, the runtime of the NMA increases drastically in relation to HORIZON as shown by the large speedup.

for both processes. We furthermore vary the duration of the send and receive events analogously to the link delay since HORIZON uses this information to identify independent events. The benchmark model thus provides both parallelization schemes with the same degree of timing information, yet differently embedded in the model.

Third, each module generates (start-)send events with uniformly distributed interarrival times. These events trigger the sending process, however, only in 10 % of the cases, a packet is actually sent to one neighboring module, where it initiates the receive process. This behavior resembles the widely used and accepted PHOLD benchmark [Fuj90b] used to profile parallel simulations. Finally, the computational complexity of the events is 0.1 ms, which corresponds to the computational complexity found in simulation models of wireless systems (see Figure 3.13). For brevity, Table 3.2 summarizes the parameters of the model.

Results

Figure 3.11 shows the speedup (positive values) and slowdown (negative values) of HORIZON in comparison to OMNeT++ for all combinations of link delays and send/receive durations. In general, we observe for all link delays that the performance of HORIZON relative to OMNeT++ improves with increasing event durations. Analyzing the individual results in more detail reveals that for large link delays of 10 ms and 1 ms, OMNeT++ clearly outperforms HORIZON in the case of small send/receive durations (10 µs and 1 µs). Yet, for larger send/receive durations ranging from 10 ms up to 100 µs, HORIZON slightly outperforms OMNeT++. On average, however, OMNeT++ achieves a better performance than HORIZON.

In contrast, for link delays of 100 µs and smaller, HORIZON shows a significant performance improvement over OMNeT++. Specifically, for a link delay of 1 µs, HORIZON outperforms OMNeT++ by a factor of 10 for send/receive durations of 1 µs, and up to a factor of nearly 900 for send/receive durations of 1 µs. We ascribe this behavior to two properties of the frameworks: i) the interplay of the event density and the lookahead, and ii) the use of synchronization messages.

Event Density and Lookahead: The event density in combination with the lookahead is a key performance factor in conservative synchronization (see Section 2.3.3.1). If the event density is high and the lookahead is large, the latter covers (on average) many events, enabling their parallel execution and hence good performance.

As the event density in this benchmark is static and given by the exponentially distributed interarrival time of the send and receive events, the performance depends solely on the lookahead. Hence, for large link delays (providing the lookahead in OMNeT++) and large event durations (providing the lookahead in Horizon), the respective simulation frameworks achieve a noticeably better performance than for the corresponding small values. For this reason, OMNeT++ outperforms Horizon for large link delays (10 ms and 1 ms) and small send/receive durations (10 µs and 1 µs) as these configurations constitute the worst case for Horizon in this benchmark.

Synchronization Messages: The NMA exchanges null-messages between LPs to guarantee deadlock-free distributed and parallel event execution. If the lookahead does not cover an event, the NMA needs to send (multiple) null-messages in order to advance the simulated time such that an actual simulation event can be processed. This behavior, also known as time-creeping problem (cf. Section 2.3.3.1), increases the runtime overhead.

In contrast, parallel expanded event simulation avoids synchronization messages altogether by focusing solely on multi-core systems which enable shared-memory synchronization. As a result, the central event scheduler in Horizon has a global view of the FES and can thus immediately advance the simulated time to the next event. Horizon hence explicitly trades off distributed simulation capabilities for a higher efficiency on our target platforms.

We observe the performance impact of the time-creeping problem when the performance of Horizon improves considerably over OMNeT++ for link delays of 100 µs and less. In these scenarios, the exponentially distributed interarrival time of the events (mean: 1 ms) is much longer than the lookahead.

We like to stress that short link delays of 1-5 µs are typical for wireless systems which cover distances ranging from meters (e. g., Bluetooth, 33 ns for 10 m) up to a few kilometers (e. g., GSM, 33 µs for 10 km). In contrast, to obtain a link delay for which OMNeT++ outperforms Horizon, i. e., 1 ms or 10 ms, the wireless links would have to span 300 km or 3000 km, respectively. Despite work on wireless communication systems covering such distances, e. g., satellite and space communication [RBF08, SB07], the majority of research efforts focus on network technologies covering smaller distances, e. g., IEEE 802.11 or IEEE 802.15.4.

In contrast, many physical processes in real systems span durations that exceed the aforementioned link delays. For instance, the duration for sending a packet of 1500 byte at 54 MBit/s in IEEE 802.11g takes 220 µs. Hence, by utilizing event durations, model developers can augment simulation models with valuable additional timing information in order to improve parallel simulation performance.

We conclude from the synthetic benchmarks that Horizon i) benefits from non-trivial processing complexities, ii) is able to equally distribute workload across worker

Parameters	Value(s)
Carrier frequency	2 GHz
Channel bandwidth	10 MHz
Mode	Time Division Duplex (TDD)
Number of resource blocks	55
Subcarriers per resource block	12
Subcarrier spacing	15 kHz
Scenario	Single-Input Single-Output (SISO)
Inter Site Distance	500 m
Fading model	Rayleigh

Table 3.3 Configuration parameters of the LTE simulation model.

CPUs, and iii) achieves considerably better performance than the NMA in networks with small lookaheads.

3.4.4.4 Case Study: LTE Network Model

In order to underline HORIZON's applicability to real simulation models, this section presents a case study using a complex simulation model of an LTE network.

System Model

For this case study, we utilize a model of a 3GPP-LTE [LTE06] compliant cellular communication system, generally referred to as 3GPP Long Term Evolution (LTE). The system model features c cells, each containing a base station (denoted Evolved NodeB (eNodeB)) serving m mobile stations (denoted User Equipment (UE)). Moreover, the network uses a Time Division Duplex (TDD) scheme which divides the simulated time into downlink and uplink frames, called Transmission Time Intervals (TTIs). Each TTI has a duration of 1 ms, however, only the downlink TTIs are used for data transmission. In each cell, the eNodeB queues incoming packets sent by a traffic generator and destined for the associated UEs. Prior to each downlink TTI, the eNodeB schedules the queued packets for transmission under consideration of the available transmission resources and a specific optimization goal, e. g., delay minimization. This optimization problem is of considerable computational complexity, resulting in events with considerable runtimes (see Figure 3.13).

On the physical layer, the system uses Orthogonal Frequency-Division Multiple Access (OFDMA) as transmission scheme with 55 resource blocks. Each resource block consists of 12 subcarriers, equivalent to a frequency width of 180 kHz. For each terminal/resource block pair, the channel gain varies randomly over time and frequency, i. e., it depends on a deterministic component (path loss) and a random, time- and frequency-variant fading component. We assume this gain to be exponentially distributed based on a multi-path propagation environment with no dominant path. For a more detailed discussion of the system model and the optimization problem,

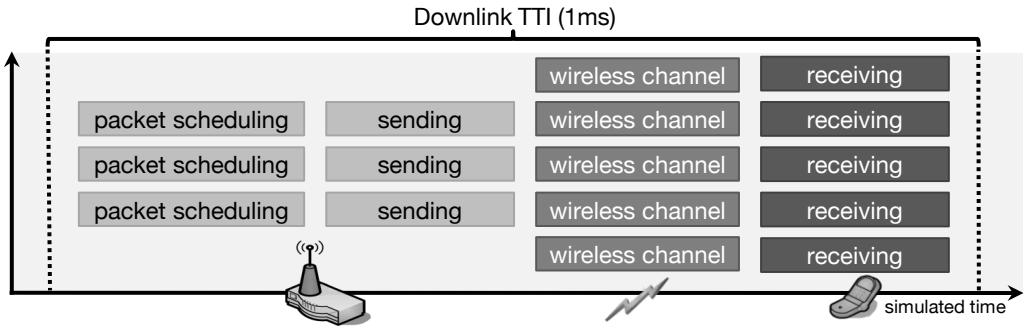


Figure 3.12 Integrating parallel expanded event simulation into a time-slotted LTE model. Based on the specification that each TTI lasts 1 ms, we assign pseudo durations that jointly span the entire TTI. Moreover, the event durations of equal event types overlap to enable parallel execution.

we refer to the work of Bohge et al. [BGMW07]. A summary of the key system parameters is shown in Table 3.3.

Time Calibration

The LTE model utilizes a TDD multiplexing scheme which divides the simulated time in up- and downlink TTIs of 1 ms each. Consequently, the timing of the events in the model is highly regular: All events regarding one downlink TTI take place at the beginning of the respective TTI at the *same* point in simulated time. In fact, a timer event indicates the beginning of each downlink TTI and triggers a recursive creation of all subsequent events, i. e., a send event creates a channel event which in turn creates a receive event and so on. All these events occur at the same point in simulated time and the ordering of the events is naturally given by their successor relationship.

Since all events of a TTI exhibit the same timestamp, we actually do not require explicit event durations to enable parallel execution. Instead, we consider discrete events taking place at the same point in time as expanded events with zero-time duration that in fact overlap. Hence, we execute all events occurring at the same point in time in parallel. Furthermore, event durations do not unlock additional performance in this model since all events eligible for parallel execution occur at the same timestamp anyway. In particular, the model needs to finish one TTI, i. e., all events at the current point in time, before it can proceed to the next TTI, i. e., events with future timestamps. For this reason, we leave the LTE model unchanged in terms of timing and do not associate durations with events.

Nevertheless, we stress that parallel expanded event simulation is applicable to this kind of system model. To this end, we briefly sketch how to assign event durations to the simulation model based on the *protocol specification* technique outlined in Section 3.3.5. The general idea is to exploit the fact that TTIs last exactly 1 ms. Specifically, we assign pseudo durations to the events of a TTI such that the resulting expanded events i) jointly span an entire TTI, and ii) overlap according to physical processes that occur concurrently. Figure 3.12 shows a schematic visualization of this approach. We observe that the expanded events spread across the entire length of the TTI while equal event types overlap in simulated time, thereby enabling parallel execution.

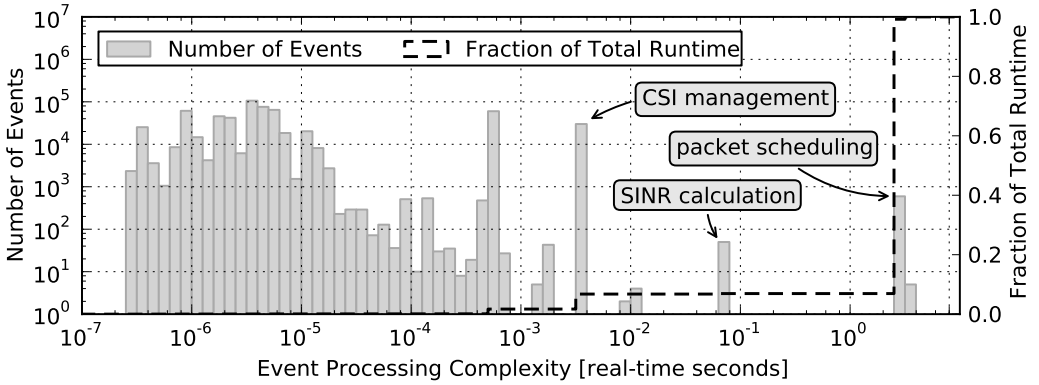


Figure 3.13 Distribution of event processing complexities in the LTE model (12 cells, 50 MS/cell) and their corresponding fraction of the total simulation runtime (black dashed CDF). The figure further indicates the three most complex types of events: Channel State Information (CSI) management, packet resource scheduling, and Signal-to-Interference-plus-Noise Ratio (SINR) calculation. Note the logarithmic scales on the x and the left y axis.

Methodology

This case study only studies the performance of Horizon but does not attempt to compare Horizon with OMNeT++. The primary reason for this decision is that the simulation model contains a central component that interacts with *all* eNodeBs with zero lookahead. It hence cannot be assigned to any partition, i. e., cell, without degrading the lookahead to zero for the whole model. We can circumvent this problem by applying pseudo-link-delays similar to the pseudo-durations as outlined above. Yet, as for the pseudo-durations, this requires considerable re-engineering and re-validation of the model. We hence refrain from modifying the model.

We furthermore analyze the workload characteristics of the LTE model to establish a foundation for understanding the performance results and to put them into perspective with respect to the findings of the previous section. Figure 3.13 shows a histogram illustrating the distribution of events processing complexities in the model. The majority of events comprise a processing complexity of 1 µs to 100 µs (note the logarithmic scales). These events account for simple functionality such as traffic generation, queue management, etc. Moreover, the figure shows that a large number of events exhibit considerable complexity, ranging from approx. 400 µs to 4 s (as indicated by the labeled peaks). The dashed line in the figure is a Cumulative Distribution Function (CDF) over the total runtime, illustrating that the events with processing complexities larger than 400 µs contribute almost exclusively to the total runtime of the simulation model, hence making them a primary target for Horizon's offloading scheme.

Summarizing, the LTE model exhibits the two key properties of modern wireless simulation models: i) a small/zero propagation delay (i. e., lookahead) and ii) computationally complex modeling of physical layer effects, making it a well-suited use-case scenario for Horizon.

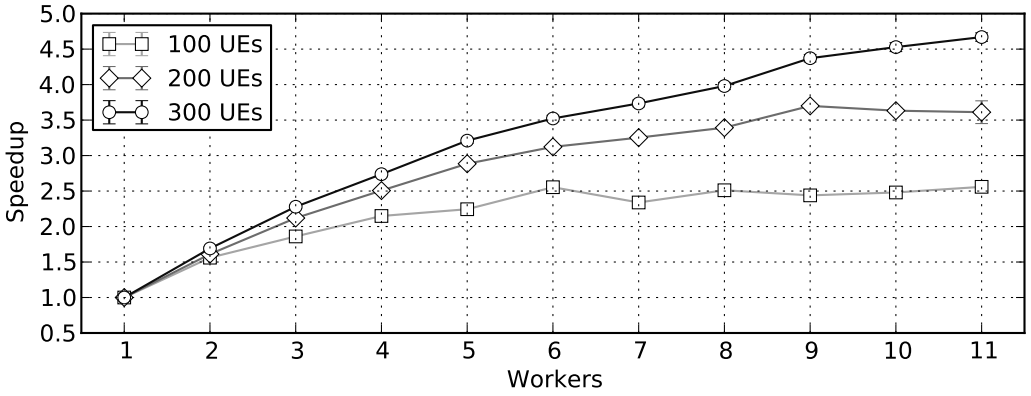


Figure 3.14 Speedup over sequential execution for workers for a network of 10 cells with a total of 100, 200, and 300 UEs.

Results

At first, we investigate the performance of Horizon for a variable number of worker threads over fixed workloads. To this end, we define three workload scenarios comprising 10 eNodeB and 100, 200, and 300 UEs, respectively. We execute all three scenarios sequentially and in parallel using 1 to 11 worker threads. Figure 3.14 shows the resulting speedups.

We observe that the speedup increases in all scenarios in conjunction with the number of worker threads, reaching a speedup of 2.5 for 100 UEs, 3.5 for 200 UEs, and 4.5 for 300 UEs. Hence, the speedup strongly depends on the workload provided by the simulation model. Analyzing the workload and the simulation model in more detail furthermore reveals that the workload is characterized by two components: The number of (parallel) events as well as the event complexity.

When increasing the number of UEs in the given simulation model, both properties change as follows. A larger number of UEs requires more channel state computations. These computations are independently modeled by individual events, hence increasing the number of parallelizable events in the model and thus the speedup. Moreover, the resource allocation algorithm performed on each eNodeB gains in complexity with increasing numbers of UEs. Since the algorithms execute in parallelizable events, the efficiency of the parallel simulation improves due to larger chunks of parallel work.

We confirm this reasoning by varying the workload over a fixed number of CPUs. Specifically, we vary the number of eNodeBs between 5, 10, and 15 and distribute a total of 100, 200, and 300 UEs among those eNodeBs. Figure 3.15 illustrates the resulting speedups. We observe that the speedup again increases with the number of UEs as well as with the number of eNodeBs.

In conclusion, this case study shows the viability of horizontal parallelization and the successful application of Horizon to complex wireless network simulation models.

3.4.5 Summary

This section presented Horizon, a parallel simulation framework on the basis of OMNeT++ that puts parallel expanded event simulation into practice. Horizon

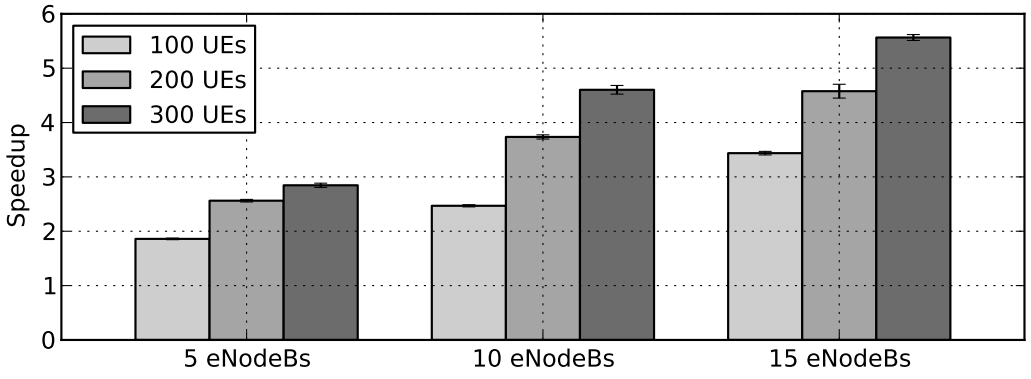


Figure 3.15 Speedup of HORIZON using 11 workers over sequential execution for different workloads comprising networks of 5, 10, and 15 cells with a total of 100, 200, and 300 UEs.

features a centralized parallelization architecture aiming to i) make use of ubiquitous small to medium scale multi-core systems, ii) simplify usage by eliminating load balancing and partitioning, and iii) form a basis for novel approaches towards event synchronization as discussed in the next two chapters.

The evaluation of HORIZON illustrates that the framework scales with the number of CPUs and the workload, however, the centralized architecture requires non-trivial event complexities to generate a speedup. HORIZON significantly outperforms OMNeT++ for small lookaheads due to enhanced time information conveyed within expanded events and the absence of the time-creeping problem.

3.5 Minimizing the Parallelization Overhead

Independent of the actual parallelization concepts, e. g., modeling paradigms and synchronization algorithms, *implementing* a parallel simulation framework that delivers satisfying speedup is challenging. The key reason is that parallel event execution imposes considerable overhead on the event handling routines of a parallel simulation framework due to thread synchronization and/or inter-process communication. As a result, we face a dilemma: We want to gain performance by means of parallelization, but this comes at the price of increased event handling overhead. The latter has a specifically negative impact on simulation models whose individual events exhibit only small computational complexity such as peer-to-peer networks, for example. In those models, the ratio of event handling overhead to actual event processing is particularly disadvantageous.

HORIZON specifically targets multi-core systems. As part of this philosophy, HORIZON employs a centralized event handling architecture. While this architecture avoids the need for explicit load balancing mechanisms, its downside is that all events need to traverse the central event scheduler sequentially. Thus, the event scheduling process is critical in terms of performance as it can easily become a bottleneck. In simulation models of wireless systems, events exhibit a non-trivial complexity, thereby lending themselves to HORIZON's architecture. However, the event complexity in other classes of simulation models, e. g., peer-to-peer networks, is considerably smaller. Thus, by minimizing the event handling overhead, HORI-

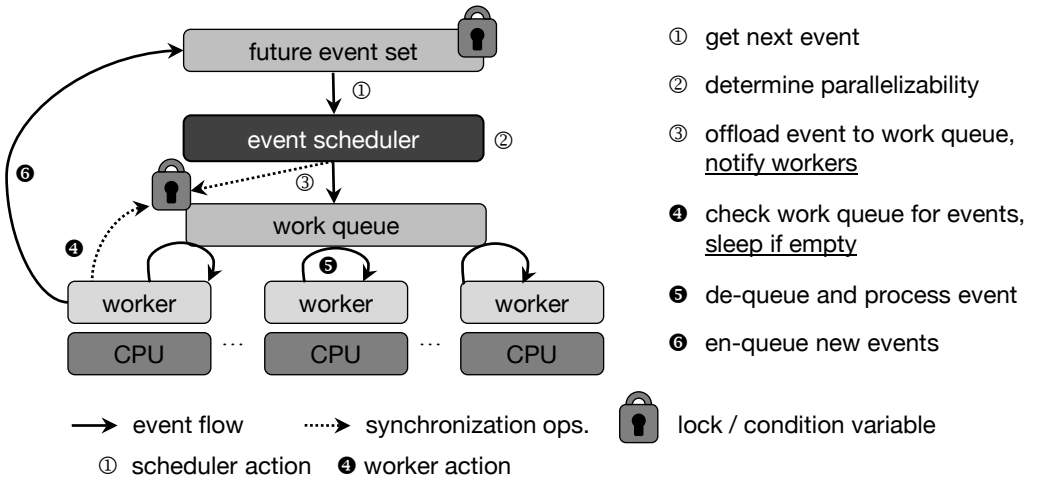


Figure 3.16 Event offloading and synchronization operations in a straightforward, pull-based scheme. Concurrent access to the work queue is coordinated by classic locks and condition variables.

zon not just increases its performance, but also becomes an attractive simulation platform for simulation models of low complexity.

Because of these reasons, we take special care to minimize the parallelization overhead of Horizon [KSGW11]. The following sections illustrate the essential optimizations towards this goal.

3.5.1 Analyzing the Parallelization Overhead

The centralized event scheduling approach of Horizon exhibits two primary advantages in comparison to the classic distributed event handling of parallel simulators. First, it drastically reduces the overhead incurred by distributed synchronization algorithms [CM79, Fuj90a, Per06b]. Instead, all required information for deriving safe event execution is readily available in one place (the event queue) and processed by one entity (the event scheduler). Second, it allows for an even distribution of work load across CPUs without the need for load balancing algorithms which again impose overhead. Horizon takes a master-worker approach in which a pool of worker threads dynamically handles events marked for parallel execution.

Nevertheless, the parallelization architecture of Horizon causes two particular kinds of parallelization overhead: The *event offloading overhead* and the *barrier synchronization overhead*. In the following, we discuss both kinds of overhead in detail.

3.5.1.1 Event Offloading Overhead

Figure 3.16 illustrates a straightforward implementation of the centralized event handling scheme of Horizon. In this implementation, the scheduler buffers all events eligible for parallel processing in a work queue from which worker threads retrieve tasks. Moreover, the workers insert new events in the central FES. Consequently, both queues need protection by locks to prevent data corruption. For this purpose,

multi-threaded operating systems provide synchronization primitives such as locks, barriers, or condition variables to achieve thread synchronization in multi-threaded programs. The fundamental principle underlying these synchronization primitives is that threads, which need to block, are suspended by the operating system until they can proceed. The reasoning behind this approach is that threads which cannot perform useful work release the CPU for use by other threads that in fact can proceed.

In case of Horizon, worker threads block and suspend when either the locks protecting the queues are occupied or when the work queue is empty. In both scenarios, the sleeping period is short, thereby resulting in frequent suspend and resume operations. While suspending and resuming of threads is considered resource efficient due to freeing up the CPU, it generates significant threading overhead because of a large number of *context switches* and system calls to the Operating System (OS) kernel. This in turn significantly increases the *offloading delay*, i. e., the time between offloading an event and actually processing it: Assume an empty work queue which causes all worker threads to suspend. Upon inserting an event into the queue, the OS wakes up one of the worker threads, loads its context into the registers of the CPU, and finally resumes the thread. Subsequently, this thread in turn needs to acquire the lock protecting the work queue, dequeue an event, and release the lock again before it can finally execute the event. Performing these non-trivial operations hence delays the execution of the event. This is particularly disadvantageous for simulation models that mainly comprise events of short processing times.

3.5.1.2 Barrier Synchronization Overhead

Expanded events that span a period of simulated time are a fundamental design property of Horizon. Since such events exhibit distinct starting and completion times, a straightforward integration in the simulation framework builds upon using two discrete events to represent both extremes of the interval. Thus, for every expanded event in a simulation model, the framework *transparently* maintains two discrete events in the central event queue accordingly. The scheduler then continuously removes the first event from the queue which can be of either type: If it represents the start of an expanded event, it is handed to the workers, i. e., the event is offloaded and executed in parallel. If it indicates the completion of an expanded event, the scheduler blocks until the associated worker has finished processing this event. Hence, we denote the latter *barrier events*.

This approach, although easy to understand and implement, imposes a considerable performance bottleneck on the simulation framework. It effectively doubles the number of events the simulation framework needs to handle – including operations such as creation, deletion, insertion to and removal from the event queue. In particular, complex simulations suffer from this extra amount of work because they already generate and maintain a large number of events in the FES. Hence, handling barrier-events in the FES further stresses the scalability of the event-queue data-structure. However, complex simulations are the primary target for parallelization, thus requiring an efficient handling of barrier synchronization.

3.5.2 Goals and Achievements

In addition to introducing the theoretical concept of expanded events, this thesis puts a special focus on *implementation efficiency*. We thus address the parallelization overhead from an implementation perspective. Our goal is specifically to minimize the event offloading delay and eliminate barrier messages. With these goals in mind, we make the following contributions:

- We present a *push-based event offloading* scheme that reduces the offloading delay, i. e., the time between offloading and actually processing an event, by enabling the event scheduler to explicitly and directly assign events to worker threads.
- We introduce a simplified *event synchronization algorithm* that eliminates the need for barrier events to indicate the end of expanded events. As a result, the algorithm removes 50 % of the total number of events and the corresponding overhead.

3.5.3 Efficient Event Scheduling

This section details on the design of our improved event handling framework. We first present our approach to reduce the event offloading overhead of the thread pool. Then, we introduce an improved event scheduling algorithm that eliminates the need for barrier events.

3.5.3.1 Cutting the Event Offloading Overhead

As stated in the previous section, suspending idle threads is considered resource efficient by freeing up CPUs for useful work. However, we argue that high-performance parallel simulations run on dedicated hardware, i. e., servers, and thus do not need to free up CPUs for other tasks when a worker is blocked. Even if utilizing desktop or workstation machines, all CPUs can be committed to parallel simulation at night while restricting the number of CPUs used exclusively by a simulation during office hours. Thus, instead of freeing up CPUs, it is more important to swiftly process events as soon as they become available without the need for waking up worker threads. Hence, we explicitly trade CPU resources for shorter offloading delays.

Our approach thus replaces the *pull-based event offloading* algorithm (Figure 3.16), in which the worker threads pull jobs from the work queue, with a *pushed-based* scheme (Figure 3.17): When offloading an event, the central scheduler checks the current processing state of all workers and explicitly assigns the event to an idling one. To this end, all worker threads maintain a local buffer (`job` in Figure 3.17) that can hold exactly one event. If a buffer is empty, the corresponding worker is currently not processing any event, allowing the event scheduler to consequently put an event into the buffer. Simultaneously, the worker threads poll their local buffer. As soon as a buffer is filled, the corresponding worker starts processing the event and finally empties the buffer again. By means of this busy waiting scheme, workers immediately recognize newly offloaded events and the time between assigning a new event and processing it is reduced to a minimum.

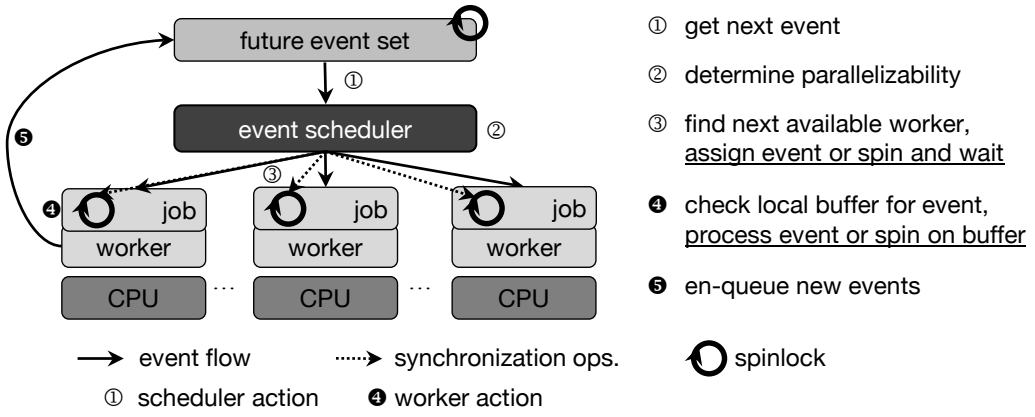


Figure 3.17 Event offloading and synchronization operations in an optimized, push-based scheme. The event scheduler directly assigns events to idling workers which actively spin on a local buffer.

It is important to point out that this approach demands a static mapping of exactly one worker thread to each CPU in order to achieve maximum performance. Nonetheless, a particular side effect of the push-based assignment of events is that the scheduler is able to identify the case that all workers are busy. In this situation, the scheduler thread may either wait for a worker to become available or it may handle the event itself. The latter case effectively adds a further CPU, i. e., the one the scheduler is running on, to the total number of worker CPUs. However, this optimization needs to be used carefully due to the apparent risk that the scheduler is blocked with handling a long running event while the workers are idling after finishing their events.

3.5.3.2 Eliminating Barrier Events

We eliminate barrier events based on the observation that it is not necessary to store the completion time of every expanded event in the global event queue. Instead, it is sufficient to maintain the completion times local to each worker for *only* those events which are currently being processed, i. e., the events in O. The scheduler selects the smallest completion time among all events in O as this represents the first barrier event that would be encountered. All events starting between the current simulated time and the barrier are safe for parallel execution.

We introduced the method of computing the barrier t_b based on O instead of barrier events as part of the parallel event execution scheme already in Section 3.4.1.3. Nevertheless, revisiting this approach here serves three purposes:

i) We underline the reasons for computing t_b based on O.
ii) We show the tight integration of the barrier computation with the push-based event offloading scheme.
iii) We illustrate the development of HORIZON throughout the course of this thesis from using barrier events as a simple yet slow means of synchronization towards a highly optimized architecture.

Algorithm 3 shows the optimized event handling architecture which combines push-based event offloading with barrier-based synchronization. Let W denote the set of

Procedure: simulate()
1: $t_\mathsf{b} := \infty$
2: $O := \emptyset$
3: **while** $F \cup O \neq \emptyset$ **do**
4: **if** $F \neq \emptyset$ **then**
5: $e :=$checkBarrier()
6: $O := O \cup \{e\}$
7: $F := F \setminus \{e\}$
8: offloadEvent(e)
9: **end if**
10: **end while**

(a) Simulation loop

Procedure: checkBarrier()
1: **while** true **do**
2: $e := \arg\min\{t_\mathsf{s}(e) | e \in F\}$
3: **if** $t_\mathsf{s}(e) \leq t_\mathsf{b}$ **then**
4: return e
5: **else**
6: waitFor($w|_{t_\mathsf{b}}$.job=$\perp$)
7: $t_\mathsf{b} := \min\{t_\mathsf{c}(e') | e' \in O\}$
8: **end if**
9: **end while**

(b) Checking the barrier

Procedure: offloadEvent(e)
1: $t_\mathsf{b} := \min\{t_\mathsf{c}(e), t_\mathsf{b}\}$
2: $w :=$ one$\{w \in W | w.\mathsf{job} = \perp\}$
3: $w.\mathsf{job} := e$

(c) Event assignment

Procedure: worker()
1: waitFor(job$\neq\perp$)
2: process(job)
3: $O := O \setminus \{\mathsf{job}\}$
4: job $:= \perp$

(d) Event processing

Algorithm 3 The main building blocks of the optimized event handling architecture. The scheduler (a) continuously checks the current barrier (b) and assigns an event to one of the workers (c) for processing (d).

workers, $\perp$ an empty buffer, and $one : \mathcal{P}(W) \rightarrow W$ a function returning either an idling worker or blocking until one such worker is available.

simulate: At runtime, the scheduler (Algorithm 3(a)) continuously i) checks the current barrier t_b, and ii) offloads independent events to the worker threads as long as there is still an event in either F or O. The algorithm checks explicitly for $F = \emptyset$ to wait for the events in O to finish execution (ending the simulation) or to insert new events in F.

checkBarrier: Checking the barrier (Algorithm 3(b)) involves accessing the first event e from the event queue and comparing its starting time $t_\mathrm{s}(e)$ to the minimum barrier t_b. If the barrier precedes e, the scheduler blocks at the barrier and waits for the event whose completion time defines the barrier to finish. To this end, the scheduler spins on the buffer of the particular worker thread that handles the aforementioned event ($w|_{t_\mathrm{b}}$) until the buffer is set to $\perp$ by the worker. Subsequently, the scheduler determines a new minimum barrier and again compares the first event from the event queue to the new barrier.

offloadEvent: Event offloading (Algorithm 3(c)) bases on our push-based event offloading approach. The scheduler first updates the current barrier if the newly offloaded event e finishes before the established barrier. In this case, the barrier must be moved to $t_\mathrm{c}(e)$ to ensure the correctness of the algorithm. The scheduler then either determines an idle worker to which it assigns e for processing or blocks until a worker becomes available. As mentioned in Section 3.5.3.1,

the scheduler might also process the event itself. Yet this might block offloading of further events if this event is of high computational complexity, thereby leaving worker threads idle after they finished processing their own events.

worker: Finally, the last component of the overall algorithm constitutes the workers (Algorithm 3(d)) which continuously process events according to the push-based event offloading scheme introduced previously.

3.5.4 Related Work

In the following, we review efforts aiming for a reduction in the event and thread synchronization overhead in multi-threaded simulation.

3.5.4.1 GVT Approximation

Synchronization algorithms require a consistent view of the global state of a parallel or distributed simulation. Mattern [Mat93] presents a selection of algorithms that efficiently determine or approximate the Global Virtual Time (GVT), i. e., the smallest timestamp among the events in all LPs, without the need for globally blocking the simulation and exchanging state information. The GVT is fundamental for both optimistic and conservative synchronization algorithms: In optimistic synchronization, the simulation framework can discard all checkpoints with a timestamp preceding the current GVT since no rollbacks to a time before the GVT can occur. In conservative synchronization, knowledge of the GVT and the lookahead allows for computing the Lower Bound on incoming Time Stamps (LBTS) and to eliminate the time creeping problem.

Moreover, Mattern's GVT approximation algorithms act as substrate for implementing butterfly barriers in conservative synchronization [RFA99]. Butterfly barriers synchronize a set of worker threads by blocking any arriving thread until all threads have arrived, and scale with logarithmic complexity with regard to the number of worker threads [Bro86]. Moreover, aiming at conservative synchronization, butterfly barriers exchange messages during a synchronization round, to compute the Lower Bound on incoming Time Stamps (LBTS) instead of the GVT. However, in order to exactly compute LBTSs, butterfly barriers need to consider transient events, i. e., events currently being sent while synchronization takes place. To handle transient events, barrier algorithms need to maintain and exchange additional meta-data, such as event counts for instance, thereby increasing the complexity of the algorithms.

In contrast, the barrier synchronization algorithm implemented in Horizon trades off scalability for simplicity. Due to the centralized parallelization architecture of Horizon, the event scheduler keeps track of the current barrier by checking if the current barrier is still valid or if it needs to be re-computed because the blocking event has been processed. In the latter case, re-computing the barrier corresponds to finding the minimum completion time over all events in O. Since O is limited to 4 to 32 events on typical target hardware, finding the minimum is reasonably fast.

3.5.4.2 Lock-less Synchronization

In order to avoid locking mechanisms altogether, Liu et al. [LNT01] present a lock-free event scheduling algorithm for parallel simulations on shared memory machines. Assuming a significant larger number of LPs than processing units, the algorithm asynchronously computes safe time bounds and determines which LPs should execute next on an available processing unit. To this end it uses a token passing mechanism: LPs exchange tokens to determine critical LPs, i. e., LPs which block the progress of other LPs in the simulation. Hence, critical LPs should execute their events first to unblock the remaining LPs.

Since the algorithm targets shared-memory multi-core systems, it makes use of atomic `fetch&add` operations to implement token passing between LPs. `Fetch&add` operations allow for atomically updating registers directly in the hardware, thereby eliminating the need for coarse grained locking mechanisms implemented in software to protect multi-threaded modification of shared data. Consequently, by means of these atomic operations, the algorithm is non-blocking since all participating LPs execute the scheduling algorithm without utilizing (spin-)locks.

The proposed algorithm achieves a considerable speedup over an equivalent lock-based algorithm. A major disadvantage of the lock-free algorithm, however, is its complexity, making it hard to understand, implement, and maintain. In addition, the overhead of the algorithm increases noticeably with the number of threads in the system. Because the number of LPs exceeds the number of processing units, the algorithm frequently switches between LPs. This in turn increases the number of context switches and the corresponding runtime overhead. Moreover, the token passing scheme strictly requires a static topology regarding the LPs since every LP needs to know its neighbors. As a result, it is well suited for wired networks with a fixed topology, but it is not applicable to wireless networks involving node mobility and changing communication partners, such as in ad-hoc networks.

3.5.4.3 Hardware-aided Synchronization

Motivated by the observation that the synchronization overhead increases with the number of processing units, Lynch et al. [LR09] address event synchronization from a hardware perspective. The authors propose a dedicated hardware unit that enables low overhead access to global state information relevant for conservative synchronization. For each processing unit, i. e., LP, the hardware unit provides a set of registers holding information such as minimum event timestamps. Based on these registers, the hardware unit atomically computes global minimum timestamps for use as safe time bounds by the LPs.

Targeting optimistic synchronization, Fujimoto proposes a rollback chip [FTG92]. The chip performs state saving and rollbacks while controlling resource utilization, i. e., reclaiming of memory occupied by outdated checkpoints. In a related effort, Quaglia et al. [QS03] develop an asynchronous checkpointing scheme that does not block the CPU. The scheme exploits asynchronous memory transfer capabilities, i. e., DMA, of special purpose hardware, for instance Myrinet network cards of high performance computing clusters. Both approaches aim for mitigating the overhead of

memory management which constitutes the key drawback of optimistic synchronization. HORIZON, instead, applies conservative synchronization which is less susceptible to this kind of overhead. Nevertheless, we present a probabilistic synchronization scheme in Section 4 as a combination of optimistic and conservative synchronization. In the context of this work, carefully adapted versions of the above schemes may improve simulation performance.

In comparison to software-based synchronization, such as HORIZON, hardware-aided approaches provide unchallenged speed. However, besides being more expensive, the most prominent downside of dedicated hardware is a lack of flexibility and extensibility. Since hardware extracts much of its performance from being specifically tailored to a particular algorithm, adapting a given hardware component to new algorithms or requirements is difficult and costly. A promising yet expensive solution to this problem is the use of Field Programmable Gate Arrays (FPGAs) which offer software-based flexibility on the hardware level.

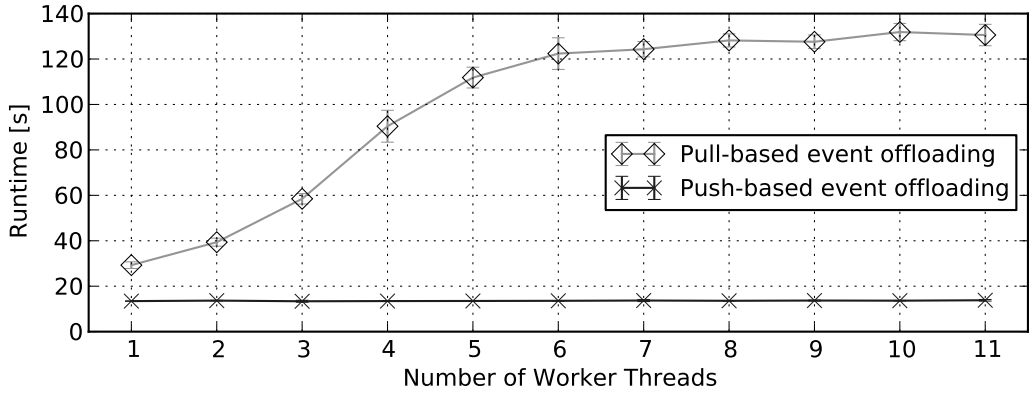
3.5.5 Evaluation

We now evaluate the performance of our event handling optimizations. Before discussing the actual results, we first introduce the evaluation setup and methodology. Then we measure the speedup gained by employing push-based event offloading and eliminating barrier events. Finally, we underline the importance of our optimizations by conducting a case study based on a real-world simulation model.

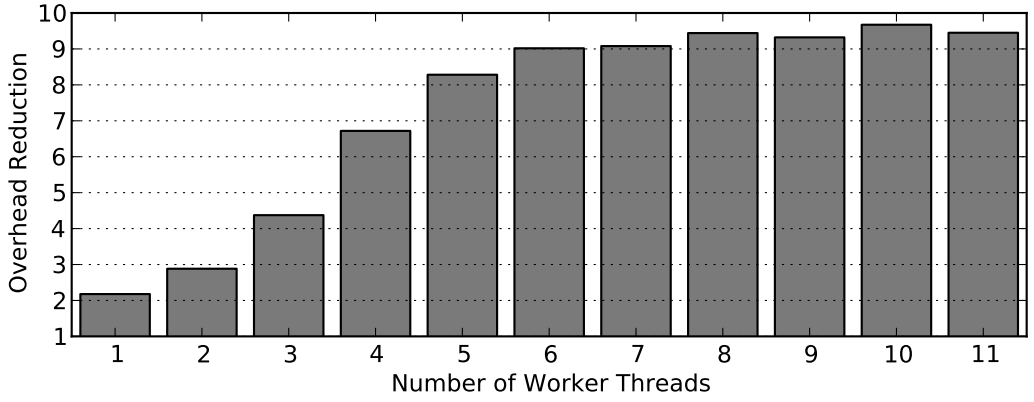
3.5.5.1 Setup and Methodology

Our goal is to measure the event handling overhead of the simulation framework. This overhead comprises all management operations performed by the simulation framework without the processing time spent in the simulation model. However, measuring the event handling overhead of a multi-threaded simulation framework is challenging because of two reasons: i) It is difficult to accurately measure the overhead imposed by thread synchronization primitives such as locks. There is no simple way of determining the time consumed by the function call without also potentially measuring the time a thread was suspended. ii) Event handling operations are split across the scheduler and the workers. Hence, determining the overhead per event as the sum of both does not accurately reflect the performance behavior as observed by the user. Instead, due to the parallelization of the scheduler and the workers, both overheads actually overlap.

As a result, we utilize a "null" simulation model whose events do not perform any computations except for re-inserting themselves in the event queue. Thus, the null-model exclusively generates overhead which allows us to derive the overall event handling overhead, including overhead parallelization effects, by measuring the total runtime of this model (without setup and teardown times). In order to provide parallelism for the workers, the model consists of 110 independent modules. Further, the expanded events span a duration of 1 s and are timed at fixed 20 s intervals 50000 times per module, resulting in a total of 5.5 million events. However, since the events of the null-model are of extremely low complexity, the event processing times are



(a) Total simulation runtime.



(b) Overhead reduction.

Figure 3.18 Performance comparison of the pull-based and push-based event offloading implementation.

exceptionally short, thereby limiting the amount of achievable parallel performance. Consequently, we do not expect a performance increase when adding more workers, but instead a performance degradation due to increased contention.

This evaluation of Horizon is based on OMNeT++ 3.3. All performance results show average values collected over ten independent runs and the corresponding 99 % confidence intervals. We utilized an AMD Opteron compute server providing 32 GB of RAM and a total of 12 processing cores, organized in two six-core CPUs running a 64-bit Ubuntu 9.10 server OS.

3.5.5.2 Event Offloading

In this section, we compare the pull-based event offloading scheme to the push-based one. Figure 3.18(a) shows the total simulation time for both approaches over a varying number of worker threads. For the pull-based implementation, we observe a linear to super-linear growth in simulation time when using up to six worker threads. As predicted, due to the workload characteristics of the null-model, we do not gain a speedup by adding more workers. Instead, additional worker threads increase the contention on the shared work queue and its synchronization primitives, hence resulting in significantly longer simulation times. When utilizing six up to eleven workers, the total simulation time remains relatively constant. We

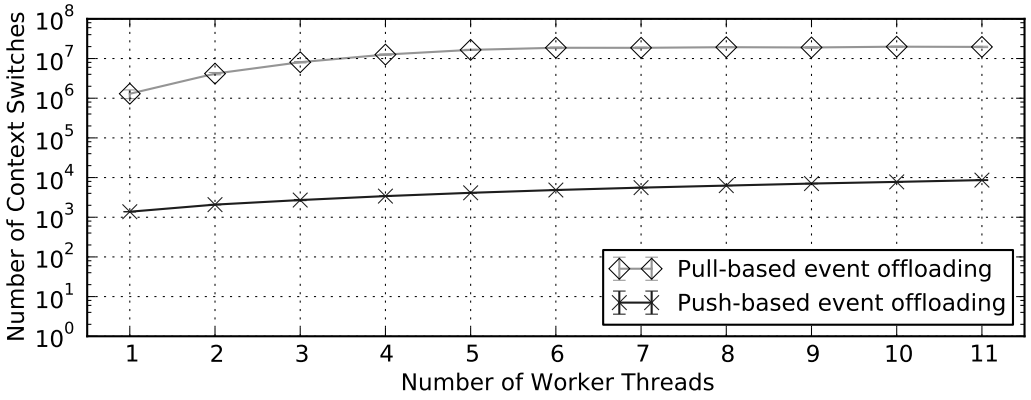


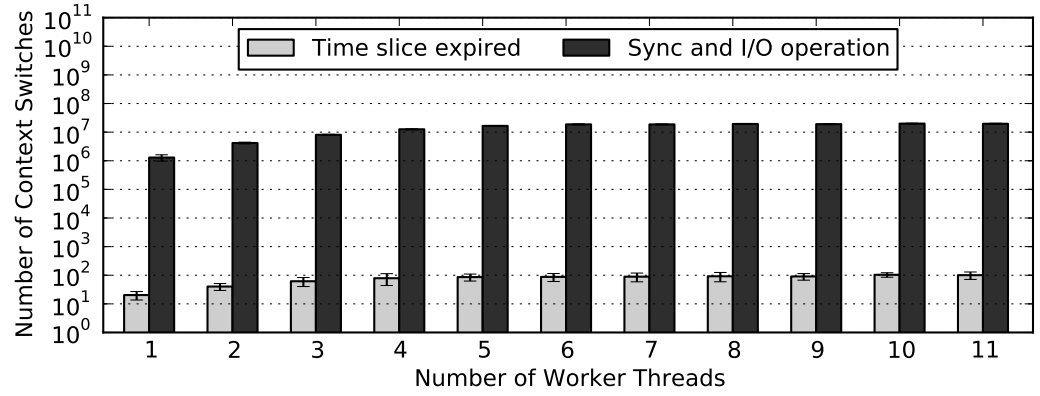
Figure 3.19 Comparison of the pull-based and push-based event offloading implementation in terms of the number of context switches.

attribute this to the fact that due to the small amount of workload the additional workers remain mostly suspended, thus limiting the level of contention. In contrast, we observe constant simulation runtimes for the push-based scheme regardless of the number of worker threads. Moreover, the total simulation runtimes are considerably shorter. From this we deduce a significant reduction in the event offloading overhead. Figure 3.18(b) compares the simulation times of both approaches in terms of the overhead reduction, i. e., the speedup factor achieved by our event offloading scheme. For a single worker thread, the modified scheme achieves a speedup of 2 and gains a maximum speedup of approximately 9.5 for eight and more workers.

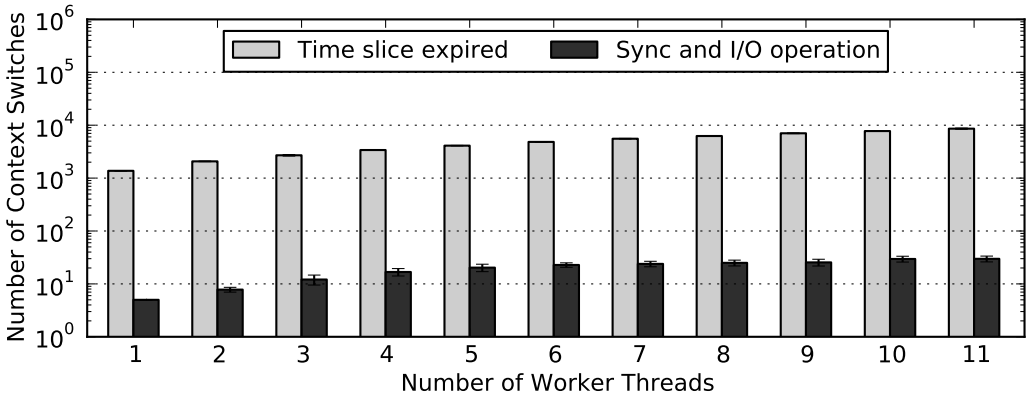
In addition to the computation time, we also measured the amount of context switches during a simulation run. In Figure 3.19, we observe a reduction in the number of context switches by three orders of magnitude. This confirms that the push-based event offloading scheme prevents excessive numbers of context switches caused by frequent thread synchronization.

In general, the amount of context switches grows in both schemes with the number of worker threads. However, in the pull-based scheme, the number of context switches stagnates for six or more workers while it still increases notably in the push-based event offloading scheme. In order to understand this behavior, we recall the conditions that initiate a context switch: i) The time slice allocated to a thread has expired: If a thread utilizes a CPU for too long, it is suspended by the operating system. ii) Synchronization and I/O operations: A thread voluntarily suspends itself while waiting for a signal from another thread or the completion of an I/O operation.

Figure 3.20(a) clearly illustrates that the number of context switches in the pull-based scheme is dominated by synchronization related context switches as a result of using classic locks. However, the amount of both types of context switches stabilizes for six and more workers due to the fact that additional threads remain mostly suspended because of the low level of parallelism achievable by the null-model. In contrast, we observe in Figure 3.20(b) that in the push-based approach the number of context switches caused by expired time-slices linearly increases while the number of synchronization based context switches levels off for more than six workers. The former suggests that due to busy waiting, the worker threads run until their time slices expire and the operating system enforces a context switch. The remain-



(a) Number of context switches in the pull-based scheme.



(b) Number of context switches in the push-based scheme.

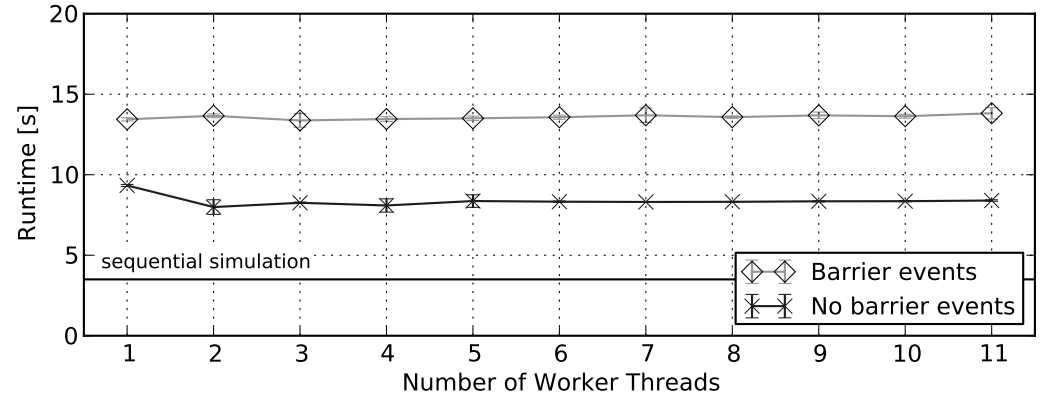
Figure 3.20 Detailed examination of the type and the number of context switches.

ing synchronization related context switches are caused by barrier events and I/O operations of the simulator.

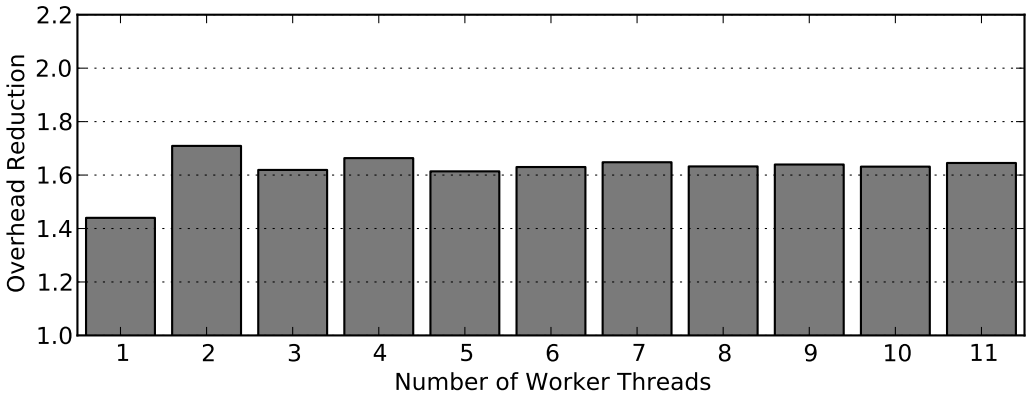
We finally derive from both figures that the relationship between the two types of context switches changes: While the synchronization related context switches dominate the total number of context switches in the pull-based implementation, their number drops to a small fraction in the push-based event offloading scheme. Concluding, these results indicate that in a highly specialized parallel simulation framework, busy waiting allows for a much more efficient utilization of the available CPU cycles than suspending and resuming of threads.

3.5.5.3 Event-free Barriers

This section analyzes the performance gain achieved by eliminating barrier events from the push-based event offloading scheme. Figure 3.21(a) illustrates the total simulation runtimes of both implementations. The event-free barrier algorithm clearly outperforms the scheduling approach that depends on barrier events. In particular, we observe a drop in the runtime when adding a second worker. We derive from this that the modified event scheduling algorithm allows for a better parallelization of its overhead between the scheduler and the worker threads. Overall, the simulation runtime for the null-model decreases by a factor of 1.4 for one worker thread and up to a factor of 1.6 for two and more workers as shown in Figure 3.21(b). Consid-



(a) Total simulation runtime.



(b) Overhead reduction.

Figure 3.21 Performance comparison of the event handling schemes with and without barrier events.

ering that the modified algorithm removes 50 % of the total number of events, this constitutes a satisfying result.

Additionally, the performance improvement is also reflected in the number of context switches as depicted in Figure 3.22. The event-free barrier algorithm generates slightly fewer context switches than the non-optimized version. Still, both algorithms show the same growth in the number of context switches which is dominated by time-slice-related context switches as discussed previously.

From the results in Figure 3.21(a), we can finally compute the average event handling time of Horizon. Given that Horizon handles 5.5 million events in approx. 8 seconds, we derive an average event handling time of 1.5 µs. Comparing this to the processing time distribution shown in Figures 3.13 and 3.24(a) indicates that after our optimizations the event handling overhead is low in comparison to typical event processing times. Moreover, the straight black line in Figure 3.21(a) illustrates the runtime of a purely sequential simulation. Hence, the difference between this line and the one of the improved scheduling algorithm reveals the remaining parallelization overhead. However, by efficiently parallelizing a real-world simulation model, we highlight in Section 3.5.5.5 that this additional overhead is indeed worth paying for.

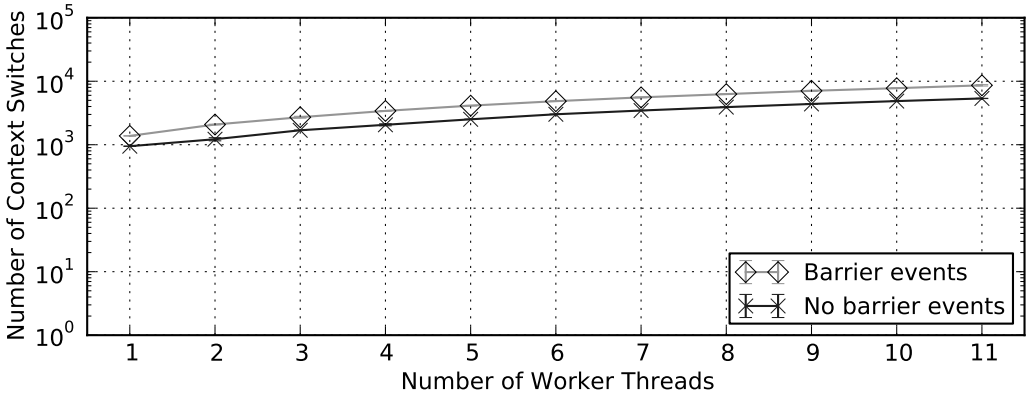


Figure 3.22 Performance comparison of the event handling schemes with and without barrier events.

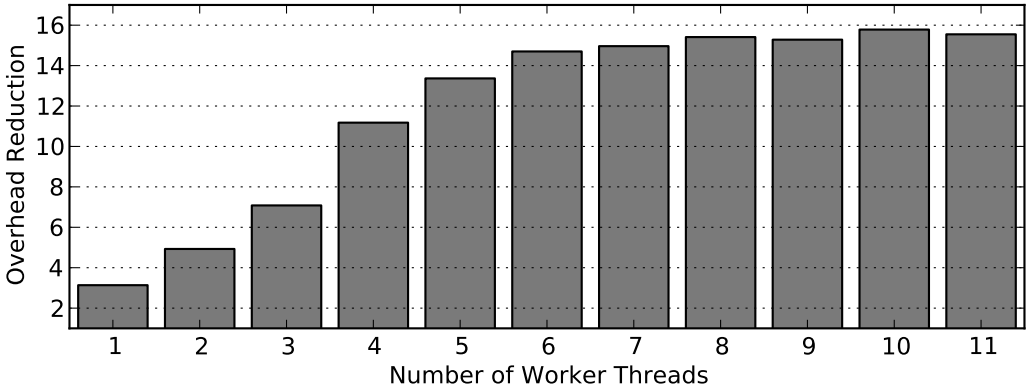


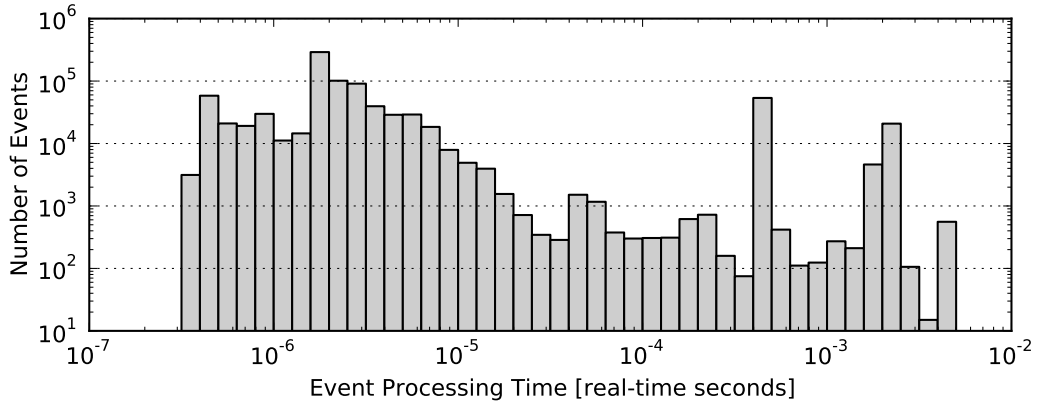
Figure 3.23 Overhead reduction with combined optimizations.

3.5.5.4 Combined Speedup

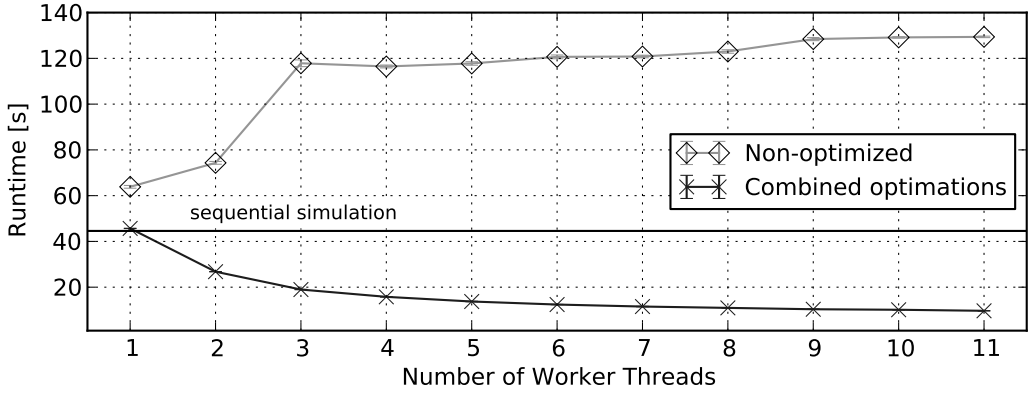
Finally, we briefly evaluate the combined performance improvement of both event handling optimizations. To this end, we compare the performance measurements obtained for the initial prototype implementation to those for the optimized version with explicit event assignment and event-free barriers. Figure 3.23 shows the corresponding results in terms of the reduction in the event scheduling overhead. In accordance with the previously presented results, the total performance gain ranges between a 3-fold speedup, obtained for just one worker, up to a peak value of approximately 16-fold for eight or more workers.

3.5.5.5 Case Study: LTE Network Model

All benchmarks up to now focused solely on determining the event handling overhead without evaluating the actual parallel performance of the simulation framework. To fill this gap, we now conduct a brief case study based on a modified version of the LTE network simulation model used in Section 3.4.4.4. We demonstrate that i) our parallelization framework indeed achieves a considerable parallel speedup, and ii) the performance improvements presented in this thesis are necessary to enable an efficient parallelization of simulation models comprising events of low computational complexity.



(a) Event processing time distribution in the LTE model.



(b) Total simulation runtime of the LTE model.

Figure 3.24 Performance analysis of a parallel LTE Network Model.

In comparison to the evaluation in Section 3.4.4.4, the model utilizes a simpler resource allocation algorithm and channel model. Furthermore, the evaluation scenario for this case study consists of a network of ten cells, each containing just five mobile stations which handle VoIP calls. These changes results in significantly shorter event processing times as illustrated in Figure 3.24(a). As a result, the selected scenario is more demanding w.r.t. the efficiency of the simulation framework.

Figure 3.24(b) plots the runtimes for simulating 1 s of network traffic. The graphs show that the non-optimized prototype implementation is not able to generate any speedup, but instead the runtimes increase with the number of workers as seen in the null-model. In contrast, the optimized event handling scheme converges to a maximum speedup of approximately five when increasing the number of CPUs in the selected scenario. This underlines the viability of our approach as well as the efficiency of its event handling algorithms.

3.5.6 Summary

We presented two optimizations to reduce the event handling overhead of our parallel simulator Horizon. The first optimization replaces the classic pull-based event offloading scheme with a push-based one, in which the event scheduler explicitly assigns events to worker threads. In combination with actively spinning worker threads, this approach results in a significant overhead reduction of a factor of up

to 9.5. The second optimization eliminates the need for barrier events to represent the end of event durations. Instead, a lightweight scheduling algorithm continuously determines the relevant barrier, thereby reducing the scheduling overhead by a factor of up to 1.6. Finally, by combining both optimizations, we yield an overhead reduction of a factor of up to 16 in comparison to our initial implementation.

3.6 Discussion and Limitations

The evaluation of parallel expanded event simulation and HORIZON presented in the previous sections demonstrate their general viability. Still, both approaches exhibit limitations we discuss in the following.

3.6.1 Parallel Expanded Event Simulation

Parallel expanded event simulation is backwards compatible to traditional discrete event simulation. Yet, porting a discrete event simulation to expanded events requires additional modeling effort. This effort can be quite low, e. g., in the case one expanded event replaces two discrete events which already model a time span. However, expanded event simulation fosters the inclusion of additional time information in simulation models which did not consider such information before. As a result, expanding formerly discrete events changes the timing of the model, thus requiring careful adjustments and (re-)validation of the model.

In addition to aiding parallel simulation, modeling physical processes by means of expanded events directly supports the development of accurate energy models. Particularly in the context of wireless systems, the energy consumption of battery driven mobile devices is a primary performance metric. Hence, it is imperative to accurately represent the energy consumption of communication systems in the corresponding simulation models [AKLW10]. Since the energy consumption is often directly linked to the duration of a process, e. g., the transmission time of a packet, expanded event simulation provides a solid foundation for detailed energy models.

Furthermore, we argue that overlapping expanded events are independent because their results only become visible to the system after the completion of the events. This is in general true in well structured simulation models in which expanded events take place in different modules. Nevertheless, this rule does not hold if events take place on the same module. Because of the ordering constraint of the FES, overlapping events on one module execute in increasing starting time order, thus the results of the preceding event apply to the state of the module before the results of the subsequent event. Model developers have to keep this ordering in mind when utilizing global modules. For instance, the wireless channel is traditionally modeled as a single global module to ease the computation of interference. However, we argue that single global modules must be avoided anyway in simulation models targeting efficient parallel execution.

Finally, the parallelization scheme of parallel expanded event simulation takes *only* the event durations into account. As a result, the event scheduler does not offload expanded events which do not overlap yet are in fact independent. Previous

efforts [LN02, MB98, MB99] showed that larger lookaheads can be derived when analyzing the interaction of simulated entities and combining their respective lookaheads. These combined lookaheads extend beyond simple event durations, thereby exposing additional parallelism in a simulation model. However, the smallest lookahead in a simulation model is typically the performance limiting factor (cf. time creeping problem). In this context, the event durations of parallel expanded event simulation aim at increasing the minimum available lookahead. Moreover, we address this limitation explicitly in the next chapter by developing a probabilistic synchronization scheme that aims at determining event dependencies beyond event durations.

3.6.2 Horizon

Horizon's centralized master-worker architecture avoids the need for explicit load balancing mechanisms by exploiting the shared-memory space of multi-core systems. We believe that the increasing availability of multi-core computers renders such systems a cheap and valuable alternative to full-sized computing clusters for small to medium sized simulations. This approach, however, limits scalability in terms of i) the size of the simulation model and ii) the number of processing units: First, large scale simulation models, e. g., peer-to-peer networks, exhibit a considerable memory footprint, easily exceeding the total memory available in one multi-core system [FPP+03, WGLW12]. Second, high performance multi-core computers will feature tens to hundreds of processing units. However, increasing the number of workers correspondingly results in extreme contention on the central FES and a bottleneck at the scheduler that cannot distribute events fast enough to keep all workers busy. Based on the measured event handling overhead of 1.5 μs (see Section 3.5.5) and the observed event complexities in our simulation models ranging up to multiple (micro)seconds (see Figure 3.13 and Figure 3.24(a)), we believe that the limit in scalability is at 50 to 100 processing cores.

However, parallel expanded event simulation is orthogonal to existing distributed simulation schemes, hence enabling a hybrid of both: In a hybrid approach, each partition of the distributed simulation model runs Horizon locally on a multi-core cluster node, while the distributed simulation framework handles synchronization and communication across nodes. Previous efforts in the research community successfully investigated hybrid synchronization schemes [LN01], hence backing the feasibility of a hybrid framework involving Horizon and expanded event simulation. Thus, by utilizing existing parallel simulation mechanisms, Horizon transparently integrates with distributed computing clusters.

Lastly, the event scheduler of Horizon does not consider caching effects or the physical memory layout when assigning events to CPUs. Currently, the scheduler offloads a given event to the next available worker, i. e., CPU core, it can find. However, this can cause events of the same event type taking place at the same module to constantly move between different CPU cores. Besides inefficient utilization of the CPU cache, processing events at changing CPU cores can result in prolonged memory access times on the prevalent Non-Uniform Memory Access (NUMA) architectures due to the need to copy data from a possibly remote memory location.

3.7 Conclusions

This chapter introduced parallel expanded event simulation as a novel modeling paradigm for efficient parallel simulation of communication systems. By modeling the duration of physical processes by means of one expanded event instead of two discrete events, expanded event simulation improves the timing information available in a simulation model. This timing information enables a parallel event scheduler to derive dependency information about expanded events, eventually allowing for conservative parallel execution of independent events. Expanded event simulation hence aims particularly at wireless network systems, which constitute a worst case scenario for conservative parallel simulation due to a tight coupling of the simulated entities, resulting in small lookaheads.

We furthermore presented HORIZON, a parallel simulation framework that puts expanded event simulation into practice. Building on top of the existing OMNeT++ simulation framework, we showed that parallel expanded event simulation integrates well with existing discrete event simulation. HORIZON aims for making the processing power of ubiquitous multi-core systems available to model developers and networking researchers. It thus employs a simple multi-threaded master-worker architecture, thereby avoiding explicit partitioning and load balancing mechanisms. The evaluation of HORIZON by means of synthetic and real-world models underlines the viability of parallel expanded event simulation. Finally, we analyzed and optimized the event handling overhead of HORIZON to achieve maximum performance.

Still, the conservative synchronization scheme of parallel expanded event simulation is often too strict. In particular, since it does not consider the *actual* dependencies among events, but instead derives dependencies conservatively just on the basis of event durations, it does not offload non-overlapping yet independent events. We address this limitation in the next chapter by presenting a probabilistic synchronization scheme that determines the actual event dependencies at runtime and exploits this information to guide speculative parallel execution of events.

4

Probabilistic Synchronization

The parallel event execution model proposed in the previous section relies on purely conservative event synchronization. However, conservative synchronization is often too strict, thus blocking the simulation and wasting processing power. In contrast, optimistic synchronization just reacts to causal violations, hence often being overly aggressive. We address these issues in this chapter by designing a general *probabilistic synchronization scheme* [KSGW12b, Sto11]. The scheme learns the repetitive patterns in the behavior of simulations at runtime and utilizes heuristics to derive event dependencies which allow for more efficient synchronization decisions.

The remainder of this chapter is structured as follows. At first, Section 4.1 motivates the need for probabilistic synchronization and sketches the general idea of our approach. We then analyze the drawbacks of traditional synchronization techniques in detail in Section 4.2 and review related efforts in Section 4.3. Based on the previous problem analysis, Section 4.4 derives the general design of probabilistic synchronization and in particular the three heuristics. We then discuss limitations of the synchronization scheme in Section 4.5, followed by an evaluation in terms of prediction quality, overhead, and performance gain in Section 4.6. Finally, Section 4.7 concludes this chapter.

4.1 Motivation

The primary goal of PDES is enabling parallel execution of events while at the same time guaranteeing deterministic results. To achieve this goal, *dependent events* essentially need to be executed in a deterministic sequential order to avoid causal violations [Fuj90a]. In practice, parallel simulation frameworks employ synchronization algorithms to ensure this requirement. These algorithms implement one (or even both) of two opposing synchronization paradigms: Conservative synchronization strictly avoids out-of-order execution of dependent events at any time during the simulation. In contrast, optimistic synchronization speculatively executes events,

but provides means of detecting and rectifying out-of-order execution. However, the key limiting factor of both synchronization paradigms is their lack of knowledge of event interactions within the simulation. For instance, conservative synchronization regularly prevents parallel event execution due to limited knowledge of future events, i. e., short lookaheads [Per06b]. Similarly, overly optimistic parallel execution of events causes frequent rollbacks to previous states, thereby impeding the overall progress of the simulation.

The research community invested considerable efforts in resolving these issues. For instance, lookahead maximization techniques [CK06, CS90, LN02, MB98, MB99] aim at expanding the lookahead of conservative synchronization schemes by analyzing the simulation model at or before runtime. However, due to the conservative nature of this synchronization paradigm, the resulting speedup still constitutes a lower bound for the actual degree of parallelism in the model [JR91].

Further efforts aim at restricting the degree of speculative execution in optimistic synchronization. For instance, Moving Time Window [SWM91] and Bounded Time Warp [TX92] use a window in simulated time and only allow optimistic execution of events which reside in this window. Similarly, Breathing Time Warp [Ste93] defines soft and hard limits on the number of events which are eligible for processing after a global synchronization phase. These limits are typically user-defined values or based on empirical measurements. Hence, they impose artificial restrictions on parallel event execution since they do not base on the observed simulation behavior and the dependencies among events. Finally, pioneering efforts towards probabilistic synchronization [Fer95, FC94, SS98] analyze the timing between events, yet such patterns do not convey enough information to accurately reconstruct event dependencies. Section 4.3 discusses all related efforts in more detail.

General Idea

In the previous chapter, we developed a parallel discrete event simulation framework, named Horizon, that builds upon a centralized event scheduling architecture [KLG+10]. Based on this work, we present in this chapter a probabilistic synchronization scheme that gathers extensive *knowledge* of the simulation behavior at runtime in order to make *educated* event scheduling decisions. The scheme exploits the repetitive behavior of simulation models by continuously collecting event scheduling information at runtime to gain an insight into event dependencies. Moreover, for each event not eligible for parallel execution according to conservative synchronization, a *heuristic* decides based on the derived dependency information whether or not the event should be processed in parallel anyway. Specifically, the heuristic determines the probability that speculative event execution results in an out-of-order execution. If this probability is below a user-defined threshold, the event is executed speculatively. This allows overcoming the restrictive scheduling of conservative synchronization while at the same time avoiding overly optimistic event execution.

Learning the behavior of a simulation model at runtime and querying a heuristic as part of the event handling process creates a crucial design trade-off. On the one hand, a heuristic should be as simple as possible to minimize its runtime overhead. On the other hand, a complex heuristic is able to derive more accurate synchronization decisions. Finding the right trade-off moreover depends on the particular

simulation model under investigation as a model involving complex events can tolerate a complex heuristic and actually benefit from better decisions. As a result, we present three different heuristics that differ in complexity and accuracy:

Arrival Pattern Heuristic: The Arrival Pattern Heuristic is of low complexity and accuracy. It observes patterns in the events arriving at the modules of the simulation model and predicts the type of the next incoming event. If the predicted event type matches the type of the next event available for execution, it optimistically offloads it. Otherwise it blocks and waits for an event of matching type to arrive.

Global Order Heuristic: The Global Order Heuristic is of medium complexity and accuracy. It derives its offloading decisions from the expected behavior of the currently executed events. Specifically, it determines the probability that one of the currently executing events creates a new event with a timestamp preceding the next available event in the FES. If this probability is below a threshold, it offloads the next available event.

Local Order Heuristic: The Local Order Heuristic is of high complexity and accuracy. Similarly to the Global Order Heuristic, this heuristic computes the probability for new events preceding the next event in the FES. However, it does not only take the currently executing events into account, but also recursively all subsequently created events as well as the modules they occur on. It hence creates a dependency tree to compute the probability that a new event precedes the next available event on the *same* module.

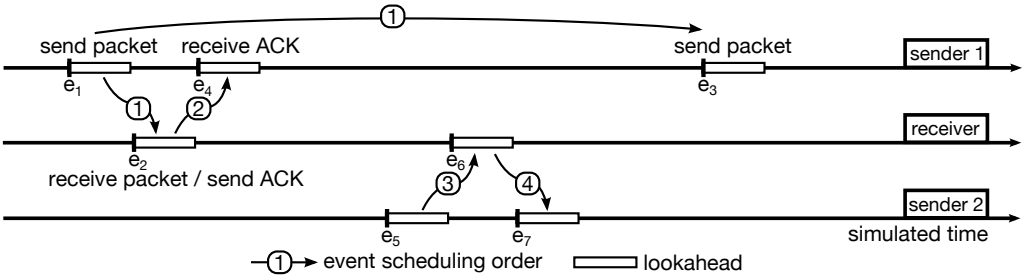
Our evaluation shows that all three heuristics are able to learn the behavior of a simulation model and considerably improve simulation performance over traditional schemes.

4.2 Problem Analysis

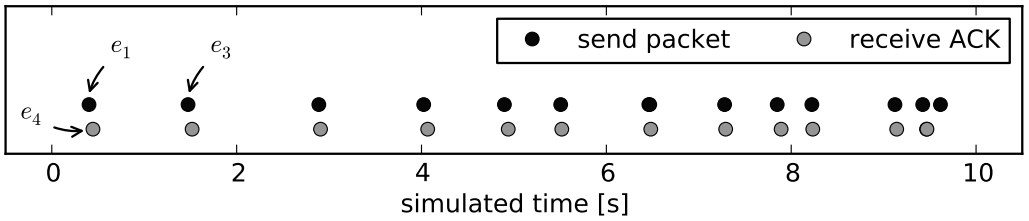
In this section, we illustrate the need for probabilistic synchronization by means of a simple example. Furthermore, we clarify the reasons for designing three different heuristics.

4.2.1 Limitations of Classic Synchronization

Assume a simple simulation model consisting of three network nodes – two senders and a receiver. The senders continuously send packets with random interarrival times to the receiver which immediately replies with an Acknowledgment (ACK). The lookahead is given by the propagation delay on the link and is thus significantly smaller than the packet interarrival times. Figure 4.1(a) shows the sequence of events modeling two separate send-packet/receive-ACK patterns. In the following, we demonstrate the weaknesses of conservative and optimistic synchronization in this scenario.



(a) Global sequence of events modeling send-packet/receive-ACK transmissions between two senders and one receiver. A send-packet event transmits a single packet to the receiver and creates the next send-packet event. The receiver simply replies with an ACK-packet.



(b) Sequence of send and receive-events occurring *locally* at sender 1. Both event types alternate, i. e., a send-event is always followed by a receive-event and vice versa.

Figure 4.1 Global and local sequence of events in a simple network model consisting of two senders and a receiver. In both cases, we observe repetitive patterns among the events. Probabilistic synchronization aims at analyzing these patterns at runtime to guide event synchronization.

Conservative Synchronization: Initially, the execution of event e_1 ("send packet") creates two successor events, e_2 ("send ACK") and e_3 (next "send packet"). In this situation, conservative synchronization solely executes e_2 because e_3 is beyond the lookahead of e_2 and another event might arrive in-between. This is indeed the case when e_2 creates e_4 ("receive ACK"). For the same reason, e_3 is not executed in parallel to the events of the subsequent send-packet/receive-ACK transmissions (e_5, e_6, and e_7). However, e_3 in fact does not interfere with those events, thus actually permitting parallel execution of e_3 with any of those events. Hence, conservative synchronization is too pessimistic in this case due to a limited lookahead, giving rise to the "blocked waiting problem" [Per06b].

Optimistic Synchronization: In contrast, optimistic synchronization speculatively executes both successors of e_1 in parallel (e_2 and e_3). However, e_2 creates e_4, thereby inflicting a causal violation and a corresponding rollback. Hence, optimistic synchronization is too aggressive in this scenario, thus limiting progress because of rollbacks.

We argue that the limitations of both schemes are partly founded in the fact that they do not take the runtime behavior of the simulation into account. However, simulation models typically exhibit highly repetitive event patterns which allow for deriving accurate knowledge of event dependencies. By means of this dependency information, synchronization algorithms can significantly improve simulation performance. For instance, Figure 4.1(b) shows the sequence of events occurring locally at sender 1. We immediately observe a pattern in the event sequence: After executing one "send packet" event, the synchronization scheme needs to wait for the pending

acknowledgment. Upon arrival and execution of the ACK event, the subsequent "send packet" event is eligible for parallel execution because no other event arrives in-between anymore. Thus, our goal is to design a synchronization scheme which analyzes such event patterns in order to improve parallel simulation performance.

4.2.2 Complexity vs. Accuracy

The centralized architecture of HORIZON is highly sensitive to event handling overhead as the event scheduler can easily become a bottleneck. Since a heuristic considerably increases the event handling overhead, it is imperative to minimize its complexity. Yet, a complex heuristic using detailed information about the simulation model may be able to derive more accurate predictions. Better predictions in turn may allow for offloading more events while at the same time reducing the number of rollbacks. Hence, we face a design trade-off between prediction accuracy and overhead.

One important parameter in this trade-off is the computational complexity of the simulation model. Conservative synchronization does not offload an event e if the currently offloaded events $e' \in O$ *might* create a causal violation. Thus, probabilistic synchronization achieves a performance gain over conservative synchronization if the heuristic correctly decides to offload an event *in addition* to the events in O, i. e., *before* all $e' \in O$ that block conservative synchronization have been processed. This blocking period grows with increasing complexity, i. e., processing time, of the individual events in the simulation model. As a result, complex models tolerate more complex heuristics. In fact, such models actually benefit from more accurate predictions since an incorrectly offloaded and computationally complex event wastes more computing resources than a computationally simple event. We accommodate simulation models of different complexity by developing three heuristics that implement different complexity vs. accuracy trade-offs:

i) an *Arrival Pattern Heuristic* of low complexity and accuracy,
ii) a *Global Order Heuristic* of medium complexity and accuracy, and
iii) a *Local Order Heuristic* of the highest complexity and accuracy.

4.3 Related Work

The community has spent considerable efforts on optimizing the two fundamental synchronization paradigms and mitigating their respective shortcomings. This chapter reviews closely related work and the current state-of-the-art in probabilistic synchronization.

4.3.1 Limiting Optimism By Means of Time Windows

An early approach by Sokol et al. [SBW88] aims for mitigating the destructive effects of overly optimistic event execution based on moving time windows. The synchronization scheme defines a time window Δ, reaching from the current GVT

into the future. Consequently, only events e with $t(e) \leq \text{GVT} + \Delta$ are eligible for execution. Since the window Δ remains static throughout a simulation run, the scheme requires evenly distributed event timestamps as the static window cannot handle large jumps in simulated time.

Breathing Time Warp [Ste93] relaxes strict window-based synchronization by dynamically deciding to either send out an event immediately or buffer it. This scheme builds on the observation that events sent to neighboring LPs shortly after determining the Global Virtual Time (GVT) (in real time) cause far less rollbacks than events sent later. In contrast to probabilistic synchronization, however, this scheme does neither consider the actual behavior of the simulation nor event dependencies.

4.3.2 Probabilistic Synchronization

In their pioneering work on probabilistic synchronization, Ferscha et al. [Fer95, FC94] analyze the arrival patterns of events. Using statistical methods such as the arithmetic mean, exponential smoothing, or median approximation over a history of arrival times, the proposed schemes estimate the timestamp of the next event. Hence, these schemes are similar to the Arrival Pattern Heuristic and thus also inherit its drawbacks. Additionally, they do not distinguish between different event types, thereby limiting insight into causal dependencies even further.

Similarly, Som et al. [SS98] construct Probability Density Functions (PDFs) from the time differences of committed events at each LP. Like in our approach, the synchronization scheme calculates the probability for a causal violation based on these PDFs and selects the next event accordingly. However, by sampling only time differences without taking event types into account, this scheme cannot derive detailed knowledge of event dependencies.

4.3.3 Lookahead Extraction

A large body of research focuses on maximizing the lookahead through manual or automatic techniques to boost the performance of conservative synchronization. For instance, Cota et al. [CS90] represent the *internal* behavior of simulation model components by means of a control flow graph. Nodes in this graph constitute different states of the component while the edges hold lookahead values which eventually allow calculating an extended lookahead on a path through the graph. However, the authors do not discuss the construction of the graph nor provide a proof-of-concept implementation.

Meyer et al. [MB98] extend this work by modeling the data flow *between* components of a simulation model in a dependency graph. Again, the edges hold the minimum lookahead in-between components and hence a path through the graph gives the total minimum lookahead between two components. In contrast to our work, the construction of the data flow graph requires manual specification of paths instead of automatic learning. Additionally, the complexity of constructing, maintaining, and traversing the graph limits performance. To reduce this overhead, the authors restrict the number of computed paths between components to a single one in follow-up work [MB99]. Still, the effort of manually defining paths remains.

Similar efforts focus on deriving larger lookaheads from domain specific model properties. For instance, Liu et al. [LN02] extract extended lookaheads from wireless networks based on packet transmission times and multi-hop propagation delays. Furthermore, Chung et al. [CK06] simulate the parallel execution of programs on multi-processor systems. Their approach performs branch predictions on top of the simulated program code to increase the knowledge of future instructions and hence the lookahead. Although both approaches can achieve considerable speedups, they are specifically tailored to a certain domain and hence lack general applicability.

Finally, relaxed synchronization [Fuj99a] allows out-of-order execution of events if their timestamps are close to each other. While this approach mitigates the restrictions of small lookaheads, it cannot guarantee repeatability across simulation runs and limits the validity of simulation results.

4.3.4 Hybrid Synchronization Schemes

Nicol et al. propose the concept of composite synchronization [NL02] as an optimization for conservative synchronization. Based on the observation that the performance of a particular conservative synchronization algorithm depends on the properties of the model, composite synchronization applies either a synchronous (global) or an asynchronous (local) algorithm to the channels between LPs in order to adapt to the particular model. In contrast to our approach, this scheme however does not incorporate optimistic synchronization.

In addition to those approaches, combined synchronization [JB94] integrates the two classic techniques into a unified synchronization scheme. The basic idea is to selectively apply either conservative or optimistic synchronization to the LPs in a simulation model to accommodate different workloads and timings per LP. Hence, by selecting the best fitting scheme for each LP, combined synchronization is able to clearly outperform both classic schemes. As a result, combined synchronization is widely used in general-purpose parallel simulation frameworks and languages such as Maisie [BL94], the High Level Architecture [Fuj98], and µsik [Per05]. Although those frameworks support dynamic switching between conservative and optimistic schemes, each scheme is applied to a whole LP or a group of LPs [RAT93]. In contrast, our approach dynamically decides for each event individually whether conservative or optimistic execution is the most favorable option to maximize performance. It hence allows exploiting differences in the workload and the timing on a much finer scale.

4.4 Probabilistic Synchronization

We now define the goals and the concept of probabilistic synchronization and present the design of each heuristic in detail.

4.4.1 Design Goals and General Concept

The primary goal of probabilistic synchronization is speeding up parallel discrete event simulations by learning the behavior of a simulation model and exploiting this

knowledge to guide speculative parallel event execution. To achieve this, we state three distinct design goals: The probabilistic synchronization scheme should

i) *guarantee the causal correctness* of the simulation despite utilizing speculative event execution,
ii) *maximize the number of correct predictions* to enable a speedup over conservative synchronization while minimizing the number of rollbacks,
iii) *minimize the prediction complexity* to limit its negative impact on simulation performance.

In its traditional mode, the central scheduler of HORIZON employs a conservative synchronization scheme to determine if the first event e in the central FES can safely run in parallel to currently executing events. If this is not the case, it blocks until all conflicting events finished processing. Instead of blocking, we extend this scheme by querying a heuristic for the probability of inducing a causal violation if e is executed anyway. In case the resulting probability is below a given threshold, the scheduler in fact offloads e speculatively. Since the execution of e is stalled while the heuristic computes a decision, we denote e as *pending event* e_p in the remainder of this thesis.

Because speculative execution inflicts causal violations, the simulation framework periodically stores checkpoints of the simulation state and checks for causal correctness. According to the proof of correctness presented in Section 3.4.1.4, we achieve causal correctness by ensuring that the local simulation time at each module of the simulation model increases monotonically. In case of an event arriving at a module with a timestamp smaller than the local time, the simulation framework initiates a rollback to a previous causally correct state.

In the remainder, we distinguish the set of events E from the set of event types $\hat{T}$ of a simulation model. Each event $e \in E$ is an instance of exactly one event type $\tau \in \hat{T}$. Moreover, τ_e denotes the type of event e which can be uniquely identified at runtime.

4.4.2 Arrival Pattern Heuristic

The general idea underlying the *Arrival Pattern Heuristic* is to analyze the patterns in which event types arrive at the individual modules of the simulation model. This approach is similar to current state-of-the-art in probabilistic synchronization [Fer95, FC94, SS98]. Specifically, the heuristic tracks at each module

i) the type $\tau \in \hat{T}$ of the event which arrived last,
ii) for every event type $\upsilon \in \hat{T}$, its number of occurrences $n_\upsilon \in \mathbb{Z}^{\geq 0}$,
iii) for every pair $(\tau, \upsilon) \in \hat{T} \times \hat{T}$, the number $n_{\tau\upsilon} \in \mathbb{Z}^{\geq 0}$, indicating how often an event of type υ followed an event of type τ.

Depending on the type τ of the last event, the heuristic determines the probability $p_{\tau\upsilon}$ that the type $\upsilon := \upsilon_{e_\mathrm{p}}$ of the pending event e_p occurs next as

$$p_{\tau\upsilon} := \begin{cases} \frac{n_{\tau\upsilon}}{n_\tau} & \text{, if } n_{\tau\upsilon} \neq 0 \\ 0 & \text{, else.} \end{cases}$$

The event scheduler then offloads the pending event if the complementary probability $1 - p_{\tau\upsilon}$ (υ does not follow τ) is below a given threshold.

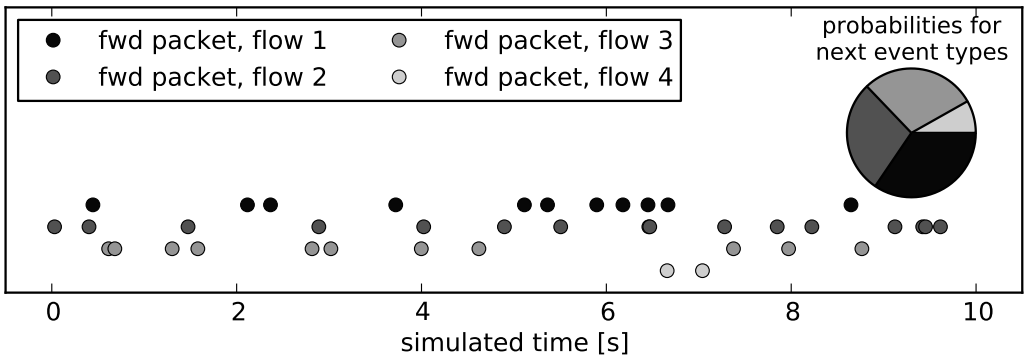


Figure 4.2 The local sequence of events on a network node that forwards four uncorrelated data streams. The pie chart on the right illustrates the computed probabilities that the next event is of the correspondingly colored type.

Applying this heuristic to the previously discussed example shown in Figure 4.1(b) yields the following synchronization decisions: The type of the last event is "send packet" (right-most point in the sequence) and both event types occur equally often, but "receive ACK"-events occur almost only after "send packet"-events and vice versa. Thus, the probability for the next event being of type "receive ACK" is nearly 1 while it is close to 0 for being another "send packet"-event. Hence, in contrast to conservative synchronization, the heuristic allows offloading the next event if it is of type "receive ACK", but it prevents erroneous offloading of further "send packet"-events as opposed to purely optimistic synchronization.

The decision making and learning process of the Arrival Pattern Heuristic is notably simple. Updating and querying the learned data only requires accessing and modifying the corresponding counter variables n_τ and $n_{\tau v}$ for the respective event types. Hence, these operations exhibit constant complexity. Similarly, the memory overhead is limited to those counter variables. Every simulation module maintains one counter for each occurring event type and one for each combination of such event types. Finally, since all data is collected and queried locally at each module, this heuristic is suitable for distributed simulation and thus does not depend on the centralized architecture of Horizon.

Nevertheless, the simplicity of the heuristic comes at the price of reduced accuracy. Its most severe drawback is that the mere occurrence of an event does not convey any information about causal dependencies, i. e., which event caused another event to occur at a particular module. Consider, for instance, four uncorrelated packet streams passing through a module as shown in Figure 4.2. The Arrival Pattern Heuristic tries to identify patterns among the uncorrelated event arrivals which however do not exist. Thus, the predictions remain inconclusive, i. e., the heuristic computes similar probabilities for most event types (see the segments of similar size in the pie chart in Figure 4.2). In addition, the heuristic does not distinguish between event instances, but just event types. Hence, even if the type of the pending event matches the predicted event type, another event of the same type but with a smaller timestamp might still precede the pending event, thereby inducing a rollback.

We conclude that analyzing the local arrival patterns of events, as also done in related efforts, does not allow for accurately predicting future events and hence limits synchronization efficiency.

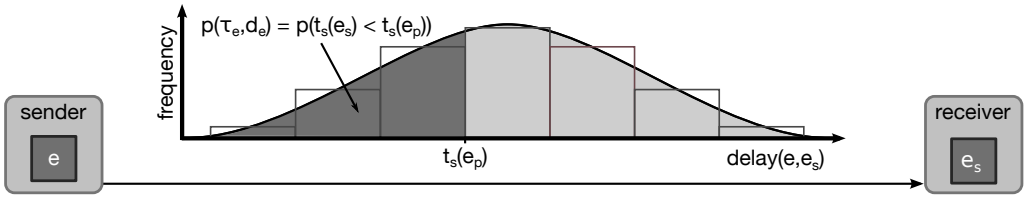


Figure 4.3 The delays between an event e and its earliest successor event e_s constitute the samples of a delay distribution. Each sample belongs to one bucket of the underlying histogram. The highlighted area under the curve represents the probability that an event e_s precedes e_p.

4.4.3 Global Order Heuristic

To obtain a better insight into causal dependencies among events, the *Global Order Heuristic* analyzes the "successor" relationship among events. Revisiting the initial example shown in Figure 4.1, we observe that a causal violation occurs at a module m only if

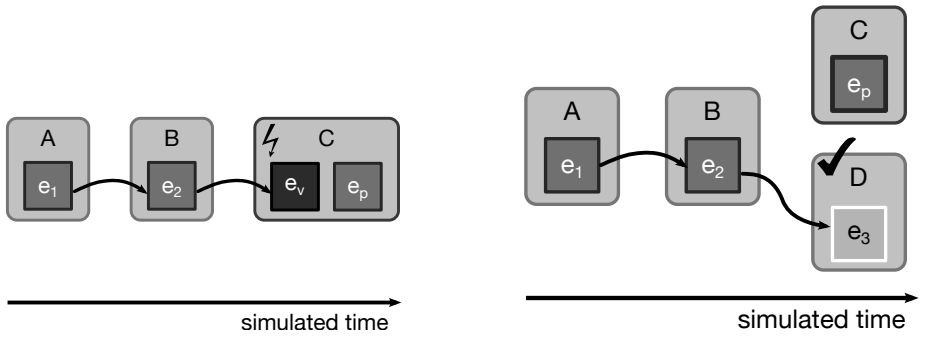
i) two events e_1, e_2 execute speculatively in parallel, and
ii) e_1 with $t_\mathrm{s}(e_1) < t_\mathrm{s}(e_2)$ creates event e_3 at module m with $t_\mathrm{s}(e_3) < t_\mathrm{s}(e_2)$.

Therefore, when deciding whether or not to offload a pending event e_p, the Global Order Heuristic has to determine the probability that any of the currently executing events creates a successor event e_s with $t_\mathrm{s}(e_\mathrm{s}) < t_\mathrm{s}(e_\mathrm{p})$. To this end, the heuristic tracks the minimum time difference (i. e., delay) between an event e and all events scheduled by e. Since the delay between two events might follow a random distribution, the heuristic has to reconstruct this distribution from a set of samples. We assume that all instances of a particular event type behave similarly during a simulation run, thus allowing the heuristic to collect samples from recurring event instances of the same type. From these samples, the heuristic constructs the (empirical) Probability Density Function (PDF) as visualized in Figure 4.3. The probability $p(\tau_e, d_e)$ that an event e of type τ_e schedules another event within a delay of $d_e := t_\mathrm{s}(e_\mathrm{p}) - t_\mathrm{s}(e)$ is given by the highlighted area under the PDF. The heuristic then aggregates these probabilities over the set $O \subseteq E$ of all currently offloaded events to compute the *conflict probability* p_c for a causal violation as

$$p_\mathrm{c} = 1 - \prod_{e \in O} \Big(1 - p(\tau_e, d_e) \Big).$$

If this probability is below a user-defined threshold, the probabilistic synchronization scheme speculatively executes the pending event e_p.

In comparison to the Arrival Pattern Heuristic, the Global Order Heuristic is of higher complexity. The heuristic maintains all PDFs in the form of histograms, each consisting of a set of buckets B (see Figure 4.3). Hence, determining p_c depends on $|B|$ (for computing $p(\tau_e, d_e)$) and the number of offloaded events $|O|$, yielding a complexity of $\mathcal{O}\big(|O| \cdot |B|\big)$ per query. Furthermore, recording a sample requires finding the matching bucket in a histogram, resulting in a complexity of $\mathcal{O}\big(\log(|B|)\big)$ using binary search. A large number of buckets allows a finer grained resolution at the price of increased memory usage and computational overhead. In HORIZON, $|O|$ is limited by the number of available CPUs, thus ranging between 4 and 32



(a) Causal violation: e_v interferes with e_p since it arrives at the same module with a lower timestamp.

(b) No causal violation: e_3 does not interfere with e_p since it occurs at a different module.

Figure 4.4 Causal correctness is a local property of each module.

for typical target platforms. The number of buckets, however, allows to trade off accuracy for memory usage and computational overhead. For each histogram, the memory usage is one integer variable per bucket. Furthermore the heuristic stores one histogram per event type and module. In Section 4.6.2.3, we illustrate that $|B| = 10$ buckets provide sufficient accuracy.

Despite allowing a considerably better insight into event dependencies than the Arrival Pattern Heuristic, the Global Order Heuristic still wastes potential for parallelization. In fact, the Global Order Heuristic is highly conservative in the sense that it prevents offloading of e_p if it predicts that at least one event e_s precedes e_p *anywhere* in the model. Figure 4.4(a) illustrates the underlying reasoning by showing a sequence of events eventually resulting in a causal violation. As a result, this heuristic effectively achieves global in-order execution of events. However, causal correctness is a *local* property of each simulation module: A simulation run is causally correct if the order of events at each module increases monotonically [Fuj90a]. Hence, global in-order execution is too strict. Specifically, if a sequence of earlier events never crosses the module of e_p no causal violation occurs, thus allowing offloading (see Figure 4.4(b)). We conclude that in order to predict the probability that an event e in O induces a causal violation at a given module, we have to analyze the path of subsequently created events through the simulation model.

4.4.4 Local Order Heuristic

In order to follow the aforementioned path through the simulation model, the *Local Order Heuristic* needs to sample the delay distribution of the successor events *and* the modules they take place on. Specifically, at a given module m, the heuristic tracks for each event e and all successors e_s created by e

i) the target module m_s on which e_s takes place and
ii) the difference in simulated time between e and e_s.

The latter constitutes the sample data for constructing delay distributions analogous to the Global Order Heuristic. However, instead of sampling just the minimum delay to all newly created events, this heuristic maintains for each event type separate delay distributions for all successor events of different type *and* different target modules.

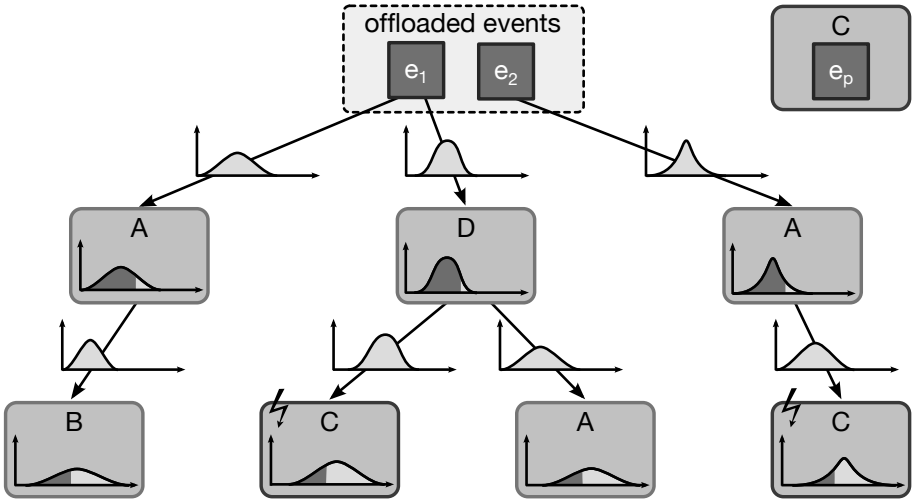


Figure 4.5 Successor tree constructed by the Local Order Heuristic during the decision making process. The edges show the delay distributions between consecutive event types on different modules. Each node contains the sum of the delay distributions along the path from the root to itself. Aggregating the probabilities of all paths to all conflicting nodes (module C) gives the final conflict probability.

4.4.4.1 Successor Tree Construction

Upon a query for computing the conflict probability regarding a pending event e_p, the heuristic constructs a successor tree G (see Figure 4.5). We first give an intuitive description of the tree construction algorithm before defining it formally:

i) For all currently offloaded events $e \in O$, it adds a node to the tree containing the event type and target module of e.
ii) For each combination of event type and target module succeeding the events in O, it appends a new node to the tree.
iii) The heuristic traverses the tree in a breadth-first manner and adds for all existing nodes new nodes containing the respective successor event type and target module to the tree.
iv) The tree construction stops when the tree contains all combinations of event types and target modules reachable from O.

Within this tree, we denote the set of nodes which represent event types occurring on the same module as e_p, *conflicting nodes* N_c. Furthermore, the edges between nodes contain the delay distributions between event types, i. e., the sampled PDFs, while each node contains the *sum* of the delay distributions on the path from the root event in O to itself. Using the aggregated delay distributions in every conflicting node, the heuristic determines the probability that an event creates a conflicting event e_c with $t_\mathrm{s}(e_c) < t_\mathrm{s}(e_\mathrm{p})$ as

$$p_\mathrm{c} = 1 - \prod_{n_e \in N_c} \Big(1 - p(\tau_e, d_e)\Big).$$

Formally, we define a successor tree G as follows:

Definition 14 (Successor Tree)
A successor tree $G = (N, H)$ comprises a set of nodes N connected by edges H.

- Each node $n_e = (\tau_e, m_e, PDF_{n_e}) \in N$ contains the event type τ_e of an event $e \in E$, the target module m_e where e takes place, and a PDF PDF_{n_e} defined below.
- An edge $h \in H$ connects two nodes $n_e, n_{e'} \in N$ if and only if $e \curvearrowright e'$.
- An edge $h \in H$ connecting two nodes $n_e, n_{e'} \in N$ is labeled with a PDF PDF_f representing the delay distribution between τ_e and $\tau_{e'}$.
- The PDF PDF_{n_e} of a node n_e is the sum of the PDFs along a unique path of edges from a root node $n_{e''}$ with $e'' \in O$ to n_e:

$$PDF_{n_e} = \sum_{\substack{f \in \text{path}(n_{e''}, n_e), \\ e'' \in O, e \in E}} PDF_f$$

Computing an exact value for p_c requires constructing the *complete* successor tree in order to find all conflicting nodes N_c, thereby imposing considerable overhead. Instead, we iteratively compute lower and upper bounds (p_l, p_u) for the conflict probability *while* traversing the tree. We obtain the offloading decision as soon as both bounds are either below or above the threshold.

The upper bound p_u is given by the probabilities over all current leaf nodes $L \subseteq N$ in the tree

$$p_\text{u} = 1 - \prod_{n_e \in L} \Big(1 - p(\tau_e, d_e)\Big).$$

Since the *aggregated* delay distributions continuously shift to the right down the tree, $p(\tau, d_e)$ steadily decreases as well. Hence, when adding new child nodes to the tree, p_u can only decrease. For the same reason, the heuristic stops exploring the tree below a conflicting node as any subsequent conflicting node yields a smaller $p(\tau, d_e)$ than its parent. The heuristic computes the lower bound p_l analogously to p_u, yet solely based on the conflicting nodes $\hat{N}_c \subseteq L$ among the leafs:

$$p_\text{l} = 1 - \prod_{n_e \in \hat{N}_c} \Big(1 - p(\tau_e, d_e)\Big).$$

This probability only increases when encountering new conflicting nodes which contribute a delay distribution with a larger $p(\tau_e, d_e)$.

4.4.4.2 Complexity Analysis

By analyzing the dependencies among events throughout the whole model, this heuristic can predict the probability of a causal violation with significant accuracy. However, the major limitation of this heuristic is its computational overhead. Successor trees can be huge and for every node in the tree we have to convolve two delay distributions in order to compute their sum. The complexity of such a convolution is $\mathcal{O}\Big(|B|^2 \cdot \log(|B|)\Big)$ due to pairwise combining all buckets of both input histograms and sorting the resulting $|B|^2$ buckets. For a tree with $|N|$ nodes, we hence determine an overall complexity of $\mathcal{O}\Big(|N| \cdot |B|^2 \cdot \log(|B|)\Big)$ per query. In general, the size of the successor tree heavily depends on the simulation model. As all events in O as well as all their successors are part of the tree, we expect $|N| \gg |O|$. Nevertheless, the

complexity for recording a sample remains at $\mathcal{O}\big(\log(|B|)\big)$, similarly to the Global Order Heuristic.

In terms of memory consumption, the Local Order Heuristic maintains considerably more histograms than the Global Order Heuristic: For each event type occurring at a module, it stores one histogram for each successor event type. Furthermore, the heuristic requires additional memory for constructing one successor tree per request. In order to mitigate the performance overhead of this heuristic, we develop two optimizations. Both optimizations improve the performance in the average case and do not influence the prediction quality.

Determinism Recognition: We eliminate costly convolution operations along the path down the successor tree if one or both of the input delay distributions represent a deterministic process with static delays. In this case, we can obtain the target distribution by simply shifting the input distributions according to the static delays. This optimization reduces the complexity of calculating the sum of two delay distributions to $\mathcal{O}(|B|)$ if one delay is static or even down to $\mathcal{O}(1)$ if both delays are static.

Distribution Function Cut-off: Figure 4.3 illustrates that only those buckets of a PDF are of interest which are below the delay in question. All buckets beyond the delay do not contribute to the sought probability. Thus, it is not necessary to include those buckets in a convolution. Instead, we can ignore those buckets in the input distributions and the resulting target distribution.

4.5 Discussion

After introducing probabilistic synchronization and each heuristic in detail, we now discuss how this scheme integrates into the overall context of this thesis.

4.5.1 Relation to Parallel Expanded Event Simulation

Probabilistic synchronization and the corresponding heuristics integrate seamlessly with parallel expanded event simulation as well as traditional discrete event simulation:

Arrival Pattern Heuristic: This heuristic only considers the type and the number of occurrences of arriving events. Both types of information are independent from the concept of expanded event simulation, thus making the heuristic applicable to both modeling paradigms. In particular, this observation also holds true if one expanded event replaces two discrete events in a ported simulation model.

Global Order Heuristic: The Global Order Heuristic samples the delays between parent and child events. In expanded event simulation, this delay is defined as the difference between $t_\mathrm{c}(e)$ and $t_\mathrm{s}(e')$ for $e \curvearrowright e'$. Similarly, this delay is given in discrete event simulation by the difference between the two discrete timestamps of e and e'. Hence, in both modeling paradigms this difference

specifies the time when a new event enters the system which *can* create a causal violation, i. e., precede an already processed event at a given module.

Local Order Heuristic: Similarly to the Global Order Heuristic, the Local Order Heuristic samples the time differences between parent and child events, but additionally also tracks the modules these events take place on. For the first kind of sampling information, we apply the same reasoning as for the Global Order Heuristic. Regarding the second sampling, we again note that the location of an event is independent of whether it is an expanded or a discrete event.

As a result, we conclude that probabilistic synchronization is orthogonal to expanded as well as discrete event simulation. This makes it applicable to both modeling paradigms, thereby underlining the relevance of our approach.

4.5.2 Relation to Horizon

The centralized parallelization architecture of HORIZON sparked the development of probabilistic synchronization and particularly of the Global Order Heuristic and the Local Order Heuristic. The primary reason is that a single central event scheduler only offloads one event at a time. Hence, while the scheduler queries the heuristic whether or not to offload a particular event, the set O of offloaded events can only shrink because workers finished processing their events. As a result, the heuristics have a *conservatively* consistent view on the state of event execution, i. e., they can only overestimate the probability for a causal violation by considering events in O that are in fact already completely processed. Furthermore, since only the event scheduler thread accesses the sample data local to each module, no locking of this data is required. In this sense, the simple architecture of HORIZON indeed fostered the development of a novel approach to parallelization, as stated as a goal in Section 3.4.1.1.

Nevertheless, on shared-memory multi-processor systems all three heuristics can be applied to a partitioned, i. e., LP-based, architecture as well. In a straightforward extension of the centralized architecture, the event scheduler of each LP queries an own instance of the selected heuristic.

Arrival Pattern Heuristic: As this heuristic analyzes the arrival patterns of events at each module, it can collect and maintain this information locally at every module. Similarly, given a pending event e_p, it only accesses and analyzes the local information at the module where e_p occurs in order to derive an offloading decision. It is hence independent of O and thus natively supports a partitioned architecture and distributed event execution. For this reason, related work on probabilistic synchronization, which focuses on distributed simulation, is limited to analyzing event arrival patterns similar to this heuristic.

Global Order Heuristic and Local Order Heuristic: Both, Global Order Heuristic and the Local Order Heuristic require access to the set O of offloaded events and the sample data, which poses no problem on shared-memory systems. It is worth mentioning that the sample data needs no protection by

locks despite being accessed by multiple heuristics simultaneously. Each module stores sample data locally and only the LP responsible for a given module actually modifies this data, while the heuristics of the remaining LPs merely read from it. Thus data corruption or lost updates do not occur.

However, determining a consistent view of the simulation state is considerably more complex in a *partitioned* multi-threaded architecture. In contrast to a centralized architecture, an LP lp_1 can add a new event e to O *while* an LP lp_2 runs its heuristic. As a result, we distinguish three cases:

i) e arrives in O in time and is hence considered by the heuristic of lp_2. Furthermore, e is still being processed when the heuristic of lp_2 derives a final decision. In this case, the result of the heuristic is as accurate as in a centralized architecture.

ii) e arrives in O in time and is hence considered by the heuristic of lp_2. However, e was completely processed before the heuristic of lp_2 derives a final decision. In this case, the result of the heuristic is outdated and lp_2 might have offloaded another event e' in the meantime. If e' creates a causal violation whereas e does not, the heuristic mistakenly offloads its pending event, causing a rollback. Hence, the heuristic underestimates the conflict probability in this scenario.

iii) e arrives in O too late and is hence not considered by the heuristic of lp_2 which already derived a final decision. If the heuristic of lp_2 decided to offload its event e', it depends on the behavior of e whether or not a causal violation occurs. In this case, the heuristic might underestimate the conflict probability. If it however determines to not offload e', lp_1 will block and consequently avoid a causal violation.

Concluding, all three heuristics integrate with a partitioned architecture. While the Arrival Pattern Heuristic natively supports distributed event execution, the Global Order Heuristic and the Local Order Heuristic suffer from reduced accuracy caused by concurrent updates of O. Still, the obvious advantage of a partitioned architecture is its ability to distribute the overhead of the heuristics across multiple LPs and hence processing units. Finally, a simple measure to counteract the tendency of underestimating the conflict probability is to adjust the offloading threshold to a higher value.

4.6 Evaluation

In this section, we evaluate the probabilistic synchronization scheme and the three heuristics in terms of prediction quality, overhead, and performance gain. To this end, we first briefly discuss relevant properties of the implementation to create a context for the subsequent evaluation. Conceptually, the evaluation consists of two parts. First, we evaluate the accuracy and overhead of the three heuristics by means of a simple evaluation model. This model enables us to precisely adjust parameters such as workloads and delay distributions. Second, we investigate the user-perceived performance gain. To this end, we conduct a case study using a wireless mesh network model.

All measurements were performed on a dedicated simulation server equipped with two six-core AMD Opteron 2431 CPUs and 32 GB of RAM, running a 64-bit Ubuntu 10.04 LTS server OS. Each data point shows the mean and the 99 % confidence interval over 30 independent runs.

4.6.1 Implementation

Our implementation bases on Horizon for OMNeT++ 3.3 [Var01]. The following sections detail on the implementation of the data sampling and checkpointing algorithms.

4.6.1.1 Maintaining Sample Data

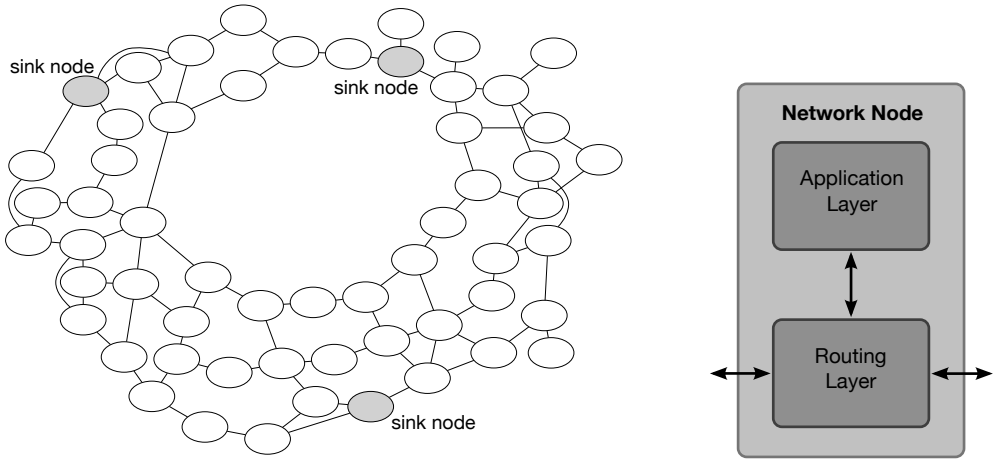
The histogram-based learning process assumes that communication patterns between modules are either static or follow fixed distributions with static parameterization. Based on this assumption, we simply add new samples to the existing histograms during the course of a simulation run. Consequently, this simple learning process cannot accurately reflect rapid and/or large shifts in the patterns of samples. While a complete change of the type of distribution is unlikely, shifts in patterns can indeed occur due to mobility (e.g., varying propagation delays). The learning process can accommodate such changes by detecting the occurrence of a large number of outliers among the samples and subsequently purge and re-sample the histograms.

4.6.1.2 Checkpointing

Since the focus of our work is on avoiding rollbacks, we use a simple `fork`-based checkpointing and rollback scheme to reduce the implementation complexity. A checkpoint corresponds to periodically forking and immediately suspending a new process, while a rollback kills the causally incorrect currently running process and resumes a previously suspended one. Although `fork` utilizes copy-on-write, it is still far less efficient than dedicated memory management frameworks [PVQ09, TQ08, VPQ10]. In addition, our probabilistic event scheduling scheme could be nicely complemented with a probabilistic checkpointing mechanism [Qua01] to drastically improve the performance over our simple periodic checkpointing mechanism. In general, our contribution is the probabilistic synchronization scheme and the three heuristics. The underlying simulation framework and its implementation merely provide a basis for evaluating the viability of our approach. By utilizing the rich set of optimizations available in the literature, the performance of the underlying implementation could be further improved, hence providing a more efficient runtime environment.

4.6.2 Synthetic Benchmarks

At first, we perform synthetic benchmarks to measure the prediction accuracy and the overhead of the three heuristics.



(a) The network structure of the evaluation model. All nodes send packets to the three highlighted receiver nodes.

(b) Each network node consists of two modules.

Figure 4.6 Components of the synthetic evaluation model.

4.6.2.1 Evaluation Model and Methodology

The synthetic evaluation model is based on a modified example provided by OMNeT++ and represents a simple network of 57 nodes (see Figure 4.6(a)). Each node consists of an application module sending data packets and receiving acknowledgments, and a routing module, forwarding incoming packets towards the destination (see Figure 4.6(b)). The application modules generate packets according to a Poisson process with a mean interarrival time of 1 s and randomly select one of three possible receiver nodes as destinations. Routers forward packets along the shortest path through the network according to pre-computed static routing tables. In addition, each router introduces a normally distributed delay with a mean of 2 ms and a standard deviation of 0.5 ms.

In order to measure the prediction quality, we compare the decisions of each heuristic against the correct decision previously computed based on a sequential simulation run. Furthermore, to avoid divergent behavior among the heuristics, we discard the actual decision of the heuristic and offload events only according to the correct decision. Moreover, we synchronize the computation time of each event to the computation time of the heuristics to factor out timing effects caused by differences in the complexity of the heuristics. Finally, all results are derived from the steady-state phase of the simulation after collecting at least 40,000 samples per distribution during the initial learning phase. Note that these modifications are only applied for evaluation purposes, and are not intended for use in real simulations.

We express the degree of optimism of a heuristic in terms of the *Positive Rate (PR)*:

$$\mathrm{PR} = \frac{\#\text{positive decisions}}{\#\text{requests}}$$

A PR of 0 means purely conservative synchronization while a PR of 1 corresponds to completely optimistic synchronization. Furthermore, we measure the prediction

quality by means of the *False Positive Rate (FPR)* which is the ratio of actually false positive decisions to *all possible* false positive decisions:

$$\text{FPR} = \frac{\#\text{false positives}}{\#\text{true negatives} + \#\text{false positives}}$$

Again, conservative synchronization always yields an FPR of 0 while the FPR in optimistic synchronization is 1. We vary the probability threshold from 0.1 % to 99.9 % with a special focus on both extreme ends. Additionally, we measure the simulation runtime. Comparing this value to a simulation run without heuristic provides the overhead added by the prediction and learning components.

4.6.2.2 Arrival Pattern Heuristic

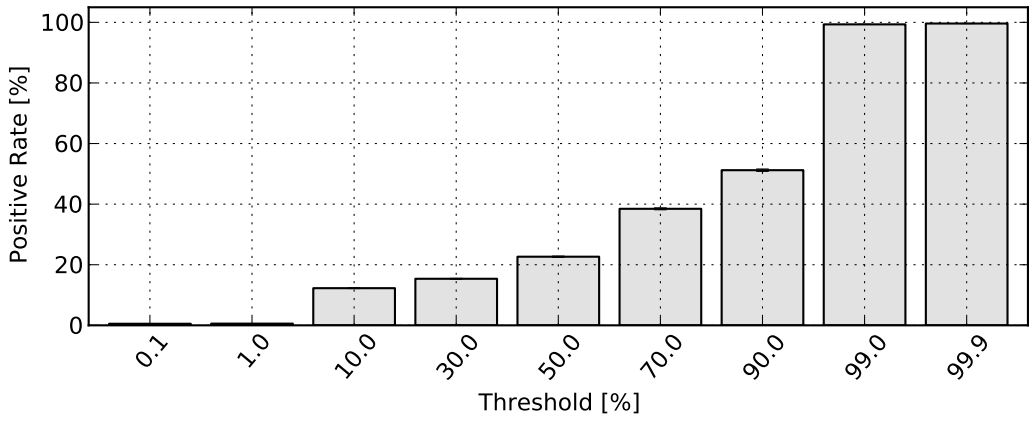
Figure 4.7(a) shows that for thresholds on both ends of the scale, the heuristic behaves almost like the corresponding conservative and optimistic schemes. For intermediate thresholds, the PR raises from 17 % to 40 % while the FPR increases from 1 % to just 4 % (see Figure 4.7(b)). Thus, despite its simplicity, the Arrival Pattern Heuristic is able to predict the next incoming event type with reasonable accuracy. Figure 4.7(c) illustrates the overhead added by the heuristic to each event handling operation. This overhead is independent of the threshold and raises event handling costs by 40 %. Concluding, this simple heuristic achieves a surprisingly good prediction quality for a wide range of threshold values while adding a reasonable overhead.

4.6.2.3 Global Order Heuristic

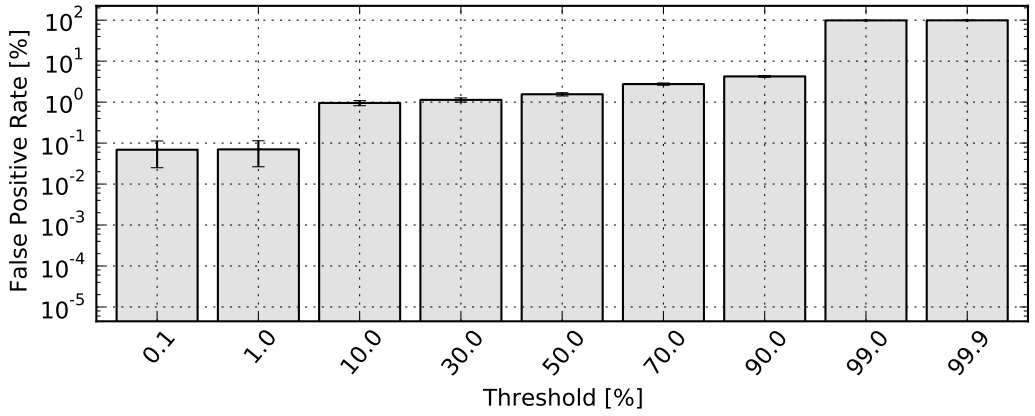
In addition to the threshold, the histogram size influences the prediction quality as more buckets allow for a finer grained resolution. We hence vary the histogram size between 3 and 50 buckets. Figures 4.8(a) and 4.8(b) indicate that 3 buckets do not provide enough accuracy to achieve reasonable predictions. However, both PR and FPR show only negligible differences for histogram sizes of 10, 20, and 50 buckets. We thus conclude that 10 buckets suffice to model a distribution reasonably well. Over the whole range of threshold values, the FPR increases to a maximum of just 1 % with a corresponding PR of 27 %. As expected, this heuristic is quite conservative due to considering conflicts from a global perspective, but it is still able to identify a considerable amount of parallelism in the evaluation model, allowing offloading of nearly one third of all pending events. Again, the overhead does not depend on the threshold, but instead on the histogram size as shown in Figure 4.8(c). On average, this heuristic increases the event handling overhead by 60 %. Concluding, the Global Order Heuristic achieves a significantly better prediction quality than the Arrival Pattern Heuristic at the price of slightly more overhead.

4.6.2.4 Local Order Heuristic

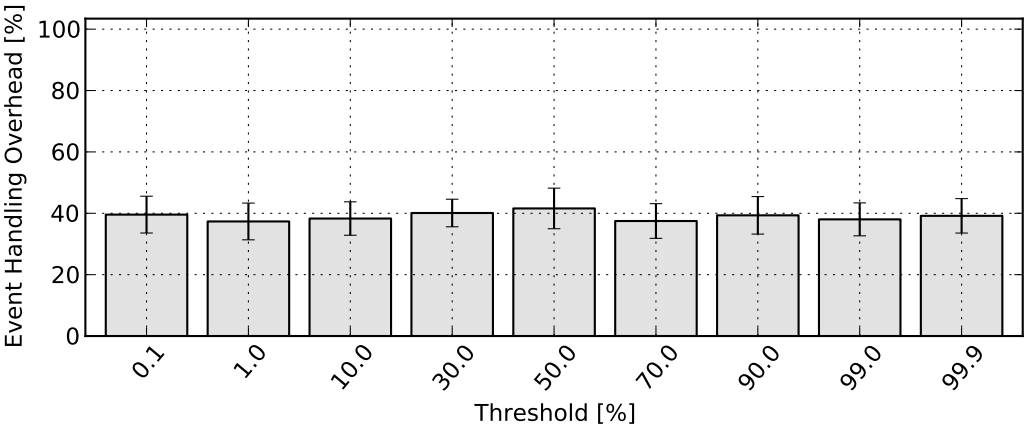
The PR of the Local Order Heuristic exhibits a relatively constant value of approx. 50 % for threshold values ranging from 0.01 % up to 90 %, before finally increasing to 70 % for extremely large thresholds (Figure 4.9(a)). Thus, already for



(a) Degree of optimism: Positive rate.



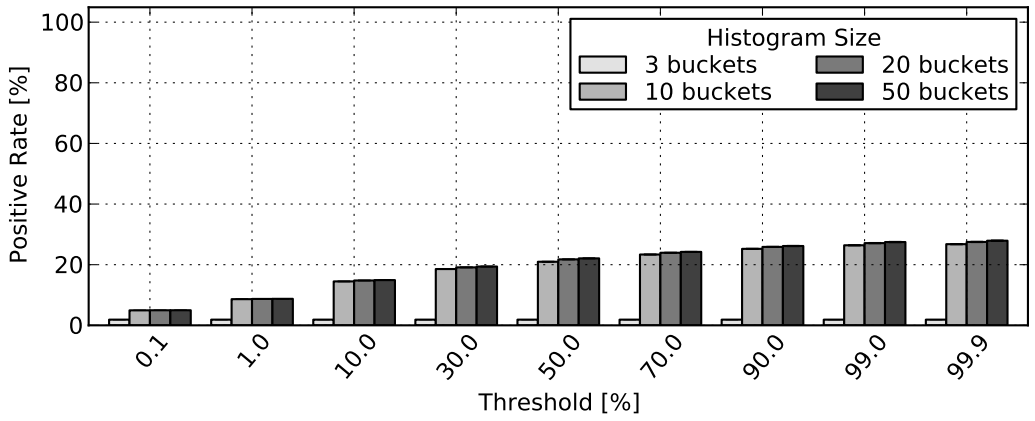
(b) Quality: False positive rate.



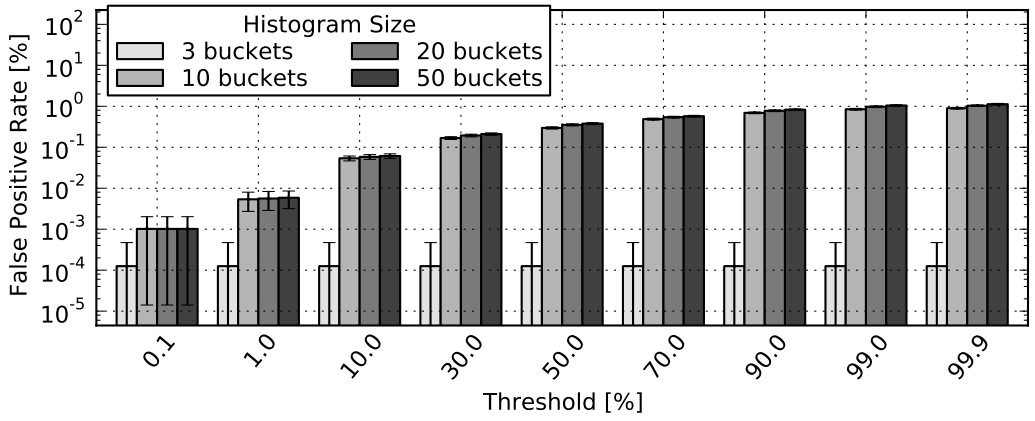
(c) Event handling overhead.

Figure 4.7 Degree of optimism, prediction quality, and overhead of the Arrival Pattern Heuristic. For thresholds between 10 % and 90 %, the Arrival Pattern Heuristic achieves an FPR of 1 %. The overhead averages at 40 %.

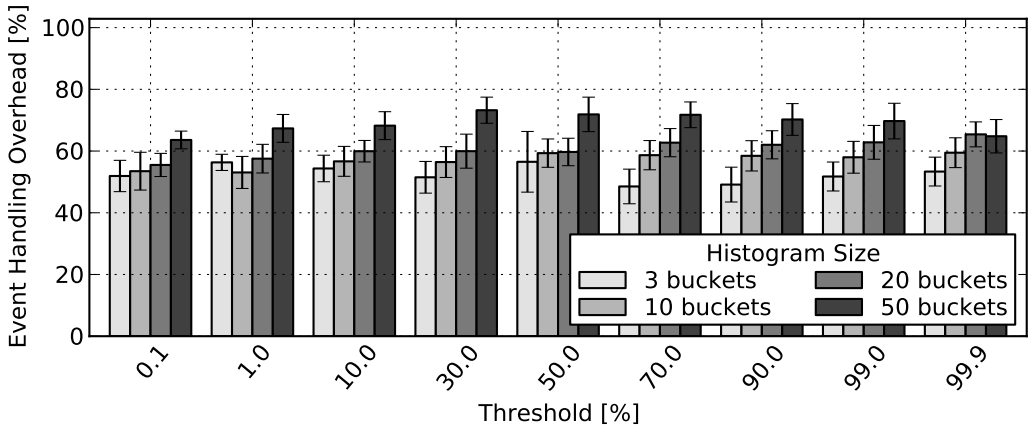
very small thresholds, the heuristic offloads every second event. Despite the large PR at such small thresholds, the FPR is initially only 0.006 % (0.0004 % for 3 buckets), but then sharply increases up to 40 % (Figure 4.9(b)). Although the latter seems disappointing at first, we conclude from these results that the heuristic is actually able to very accurately predict the conflict probability: On the one hand, the heuristic computes small conflict probabilities for events which are in fact independent,



(a) Degree of optimism: Positive rate.



(b) Quality: False positive rate.

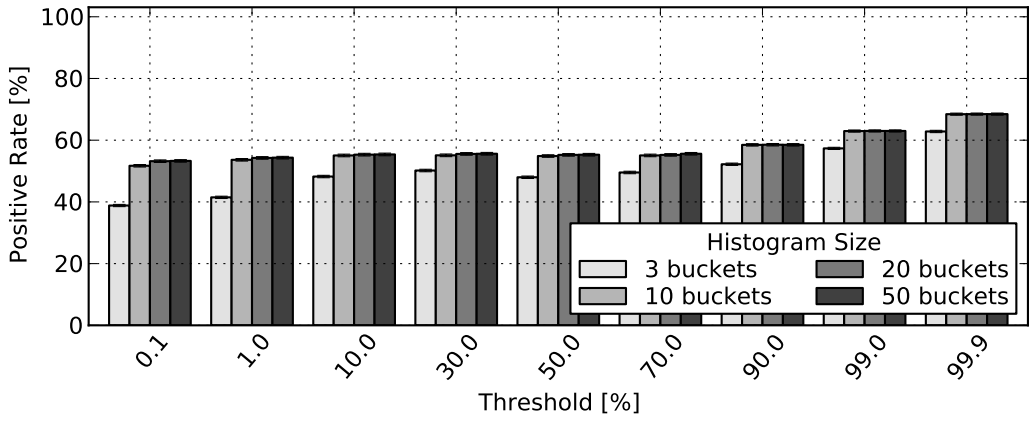


(c) Event handling overhead.

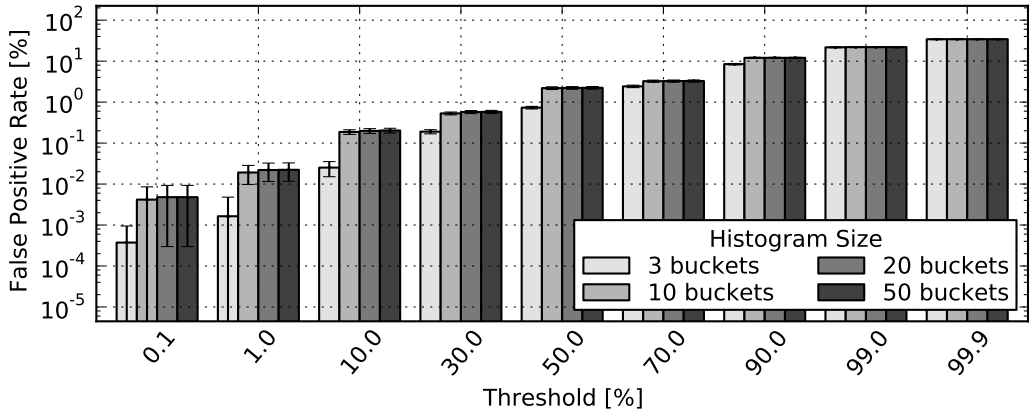
Figure 4.8 Degree of optimism, prediction quality, and overhead of the Global Order Heuristic. This conservative heuristic limits the PR to a maximum of 27 % resulting in a low FPR of merely 1 %. The overhead averages at 60 %.

hence allowing for safely offloading those events with small thresholds. On the other hand, large thresholds force the heuristic into taking too optimistic decisions since offloading an event with high conflict probability indeed induces a causal violation.

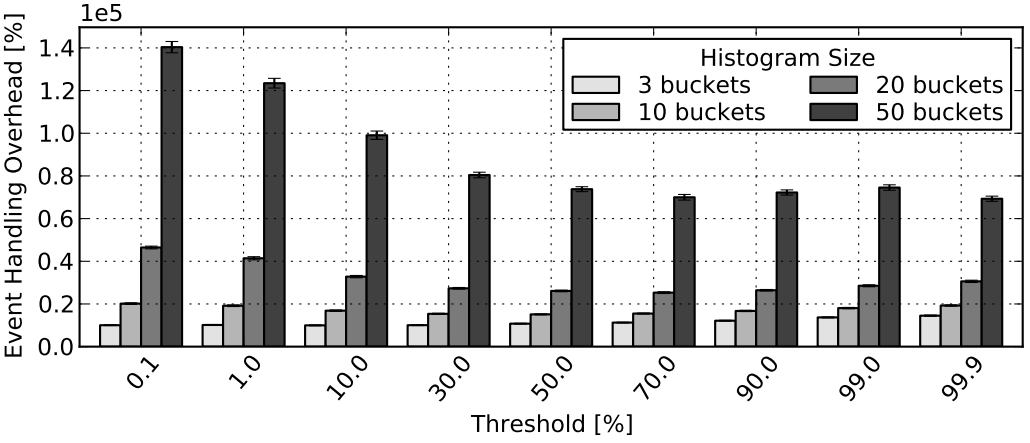
The overhead of this heuristic grows with increasing histogram sizes and decreasing thresholds. The latter is due to the fact that for smaller thresholds, the heuristic needs to further traverse the successor tree in order to make sure that no conflicting



(a) Degree of optimism: Positive rate.



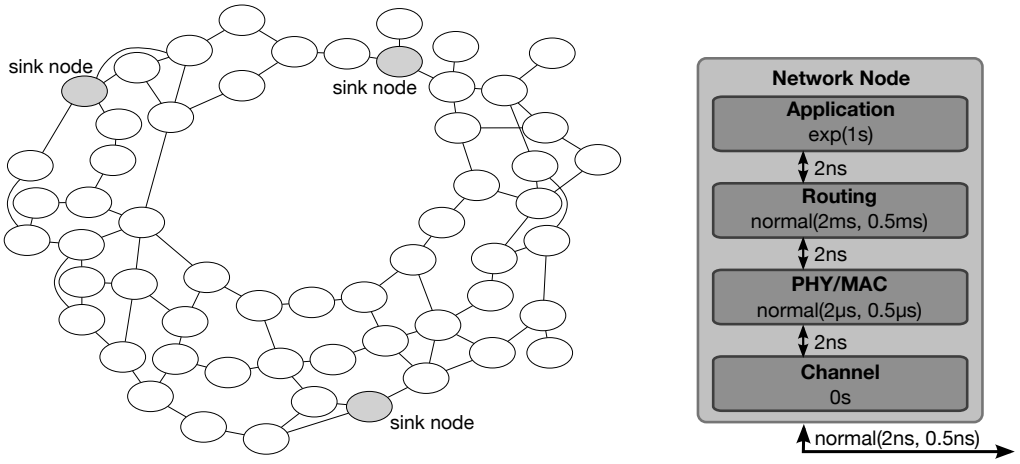
(b) Quality: False positive rate.



(c) Event handling overhead.

Figure 4.9 Degree of optimism, prediction quality, and overhead of the Local Order Heuristic. Due to accurate knowledge, the heuristic offloads every second event (PR of 50 %) for low thresholds with very low FPRs.

event exists in the tree. Overall, the overhead of this heuristic exceeds the overhead of the other two heuristics by orders of magnitude. Hence, this heuristic should only be used in conjunction with models of non-trivial computational complexity. Nevertheless, we show in the next section that given such a model, this heuristic outperforms the other two.



(a) The network structure of the evaluation model. All nodes send packets to the three highlighted receiver nodes.

(b) Delay and processing time distributions used in the extended nodes of case study.

Figure 4.10 Components of the network model used in the case study.

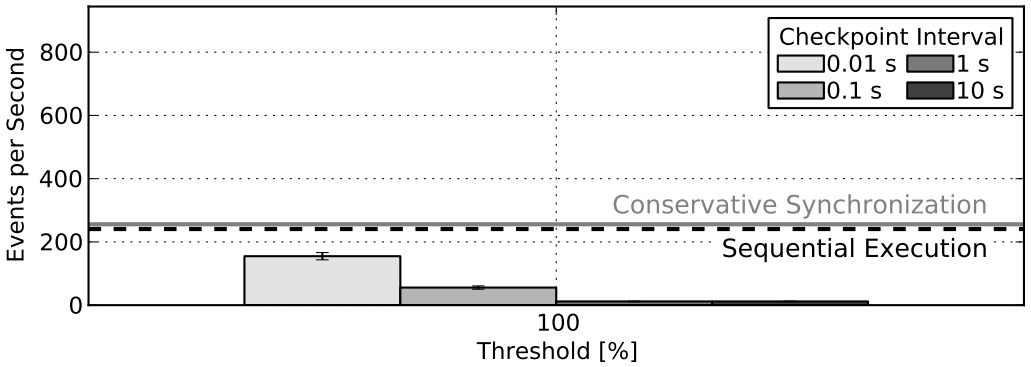
4.6.3 Case Study

To complement the synthetic benchmarks, we conduct a case study to show the user perceived performance gains in the context of a wireless multi-hop mesh-network model.

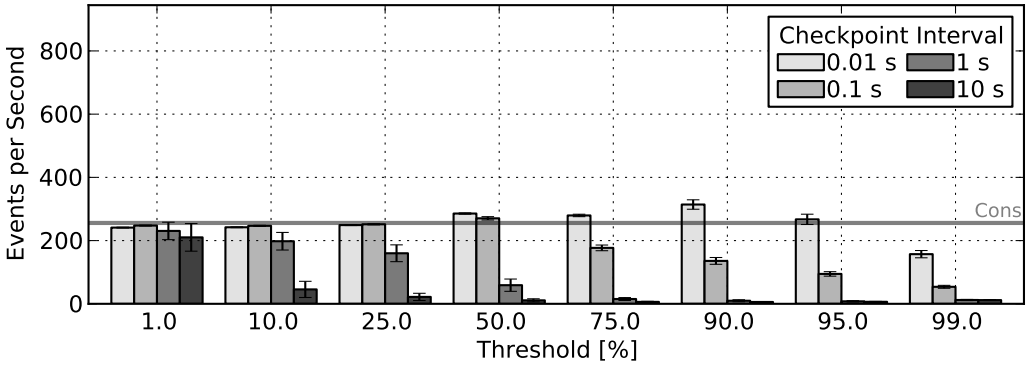
4.6.3.1 Evaluation Model and Methodology

We integrate a wireless transmission scheme with the synthetic benchmark model: While the topology and the basic traffic flows remain the same, each transmission is now received by all directly neighboring nodes due to the broadcast nature of the wireless channel. To model the wireless transmission, we furthermore extend each network node with an accurate OFDM channel model and a simple PHY/-MAC component implementing a threshold based packet error model. On each transmission, the OFDM channel component computes the fading components for each OFDM carrier while the PHY/MAC component implements a simple threshold based packet error model. For brevity, we do not discuss these models here, but refer to work by Puñal et al. [PEG11]. Furthermore, the model abstracts from a concrete MAC scheme, interference, and transport layer protocols.

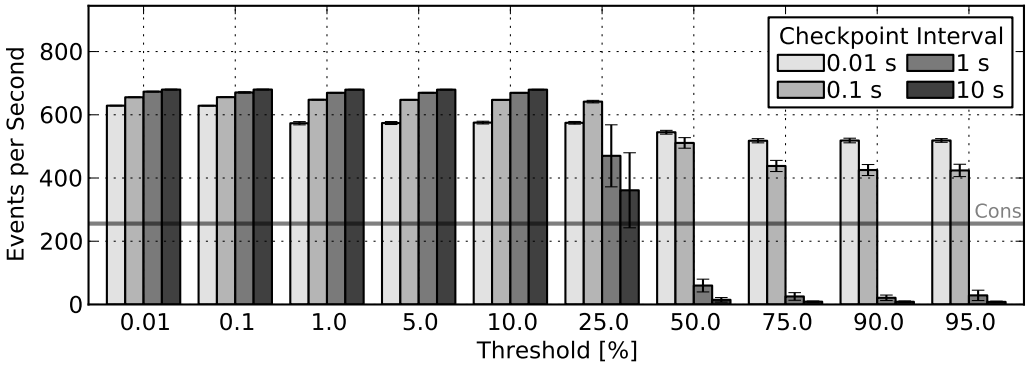
Figure 4.10(b) shows the structure of the extended network nodes, the event durations, and the delays used in the model. We measure the runtime performance in terms of the number of computed events per second during the steady state phase of the simulation while utilizing all 12 CPUs of our simulation server. In order to limit the number of results, we selected relevant thresholds based on the synthetic evaluation. Moreover, we chose a wide range of checkpoint intervals, ranging from 0.01 s to 10 s of wall-clock time to investigate the trade-off between checkpointing overhead and work preservation.



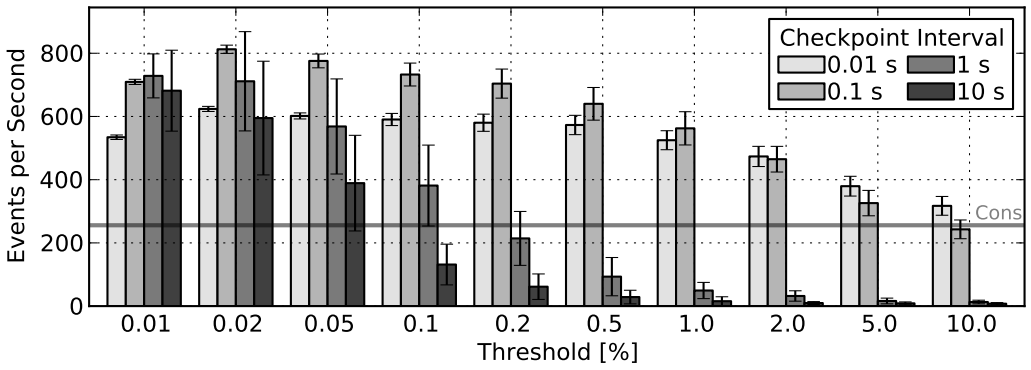
(a) Traditional optimistic and conservative synchronization.



(b) Arrival Pattern Heuristic



(c) Global Order Heuristic



(d) Local Order Heuristic

Figure 4.11 Performance of the traditional synchronization techniques and our heuristics in the case study.

4.6.3.2 Traditional Techniques

Figure 4.11(a) illustrates the performance of sequential execution (black dashed line), conservative synchronization (gray line) and purely optimistic synchronization (bars) for varying checkpoint intervals. Note that these baseline results could be potentially improved, considering the simple proof-of-concept implementation of our checkpointing mechanism. Nevertheless, even in an optimized implementation, the inherent shortcomings of both synchronization schemes remain. Hence, if our heuristics are able to prevent a rollback or a blocked waiting period, the overall performance increases also on an optimized simulation platform.

Most noticeably, conservative synchronization does not achieve any speedup at all in this model. This disappointing result is due to extremely short lookaheads given by the short propagation delays (see Figure 4.10(b)). We furthermore observe that purely optimistic synchronization is even slower than conservative synchronization over the whole range of checkpoint intervals. Hence, optimistic synchronization suffers from frequent rollbacks. This is further underlined by a tremendous difference in performance between different checkpoint intervals. For large intervals of 1 s to 10 s, the simulation makes almost no progress since a rollback is likely to occur before the next checkpoint is reached.

4.6.3.3 Arrival Pattern Heuristic

The Arrival Pattern Heuristic performs generally worse than conservative synchronization (see Figure 4.11(b)). Only for short checkpoint intervals of 0.01 s to 0.1 s it is able to gain a small speedup over the conservative scheme. In comparison to optimistic synchronization, this heuristic is nevertheless able to prevent a considerable number of rollbacks. Hence, the simulation performance exceeds purely optimistic synchronization. These results support our claim that arrival patterns do not provide enough information about event dependencies to accurately predict future events.

4.6.3.4 Global Order Heuristic

In contrast to conservative synchronization, the Global Order Heuristic achieves a 2.6-fold higher event processing rate for small thresholds and checkpoint intervals (see Figure 4.11(c)). Interestingly, up to thresholds of 10 %, the two large checkpoint intervals of 1 s to 10 s outperform the smaller intervals before showing a steep drop in performance for thresholds larger than 25 %. We ascribe this to the fact that the reduced overhead of larger checkpointing intervals allows for a higher processing rate at small thresholds. However, as soon as the threshold grows too large, rollbacks frequently occur before the next checkpoint is reached, thus preventing progress of the simulation. In this case, a higher checkpointing rate achieves better performance due to preserving more work.

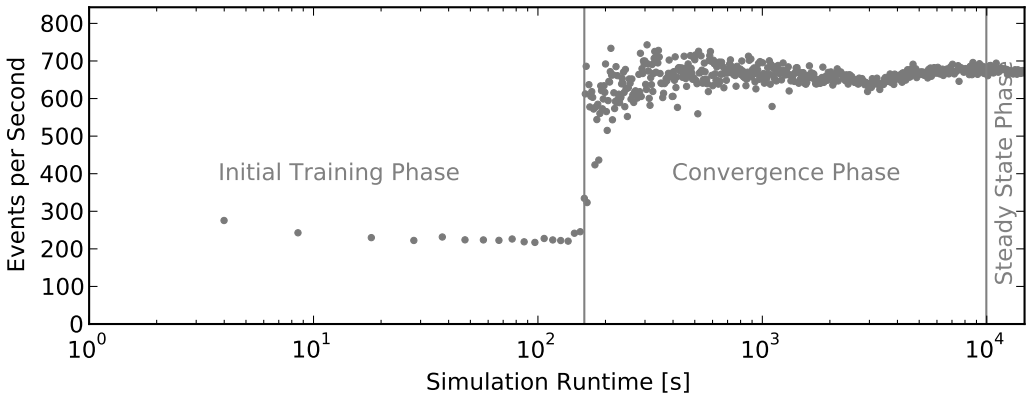


Figure 4.12 In a probabilistic synchronization scheme simulation runs consist of three phases.

4.6.3.5 Local Order Heuristic

Finally, Figure 4.11(d) shows that the Local Order Heuristic clearly outperforms all other schemes for short checkpoint intervals and very small thresholds. In particular, we observe a maximum speedup of about 3.2x over conservative synchronization. However, as expected based on the synthetic evaluation results, the performance of this heuristic rapidly declines with increasing threshold sizes.

In conclusion, the results of this case study confirm our previous assumptions and support our design decisions: A complex heuristic is indeed able to more accurately predict causal violations and hence outperform simpler heuristics, given that the simulation model is of non-trivial complexity. Nevertheless, we also observe a considerable impact of the chosen parameters, particularly threshold size and checkpointing interval, on the overall performance. Hence, in the current state of this work, a well-founded understanding of the heuristics is required to choose appropriate parameters. Future work aims at solving this issue by developing automatic calibration techniques.

4.6.4 Synchronization Phases

The heuristics need to collect a minimum set of training data before being able to predict causal violations with reasonable accuracy. Moreover, the prediction accuracy increases with the size of the training data. As a result, a probabilistic simulation run passes through three different phases as illustrated in Figure 4.12.

Initial Training Phase: While the heuristic collects the initial training data, the event scheduler employs traditional synchronization techniques. Our implementation utilizes the legacy conservative synchronization scheme of Horizon for this phase. Consequently, the runtime performance is low.

Convergence Phase: After sampling a predefined amount of training data, probabilistic synchronization commences. In the example shown in Figure 4.12, speculative event execution is enabled after at least one distribution contains more than 400 samples. At this point, approx. 3 minutes into the simulation,

runtime performance grows rapidly but shows large fluctuations due to still incomplete sample data. However, with increasing sample data size, the runtime performance converges towards a steady state. Determining the optimal size of the initial training set is left for future work.

Steady State Phase: Finally, about three hours into the simulation run, the runtime performance completely stabilizes and enters the steady state phase. In this phase probabilistic synchronization achieves its maximum performance. Hence, probabilistic synchronization primarily targets long running simulations with runtimes up to several hours or days. However, we can *immediately* skip to the steady state phase by preserving the sample data between runs if the simulation model does not change. In practice, this is indeed often the case since multiple repetitions of the same run are needed to obtain statistically significant results.

4.7 Conclusions

We presented a probabilistic synchronization scheme that learns the runtime behavior of a simulation to guide speculative event execution. Its core components are three heuristics that continuously collect event scheduling information at runtime. Based on this information, the heuristics compute the probability for inflicting a causal violation when speculatively executing events. We developed three different heuristics to trade off prediction accuracy and runtime performance. Our evaluation investigates the exact impact of this trade-off and quantifies the prediction accuracy. Furthermore, by means of a case study using a wireless mesh network, we illustrate that all three heuristics outperform conservative and optimistic synchronization.

Probabilistic synchronization successfully speeds up the runtime of a single simulation run on multi-processor systems. In practice, however, evaluating the performance of complex systems requires extensive parameter studies, i. e., repeated simulation runs with difference combinations of parameters and random seeds. The next chapter illustrates a novel parallelization approach that exploits the massively parallel processing power of GPUs to significantly speed up such parameter studies on multi-processor systems.

5

Multi-level Parallelism on GPUs

The key goal of this thesis is to foster and improve parallel simulation by exploiting the parallel processing power of modern multi-processor systems. To reach this goal, we designed and presented two contributions in the previous chapters: parallel expanded event simulation and probabilistic synchronization. While both approaches take advantage of shared-memory multi-processor systems, they do not yet consider a powerful and ubiquitously available class of parallel hardware: Graphics Processing Units (GPUs).

GPUs provide massively parallel processing capabilities and form an integral part of the majority of target platforms of this thesis, i. e., desktop and workstation computers. Hence, GPUs constitute a highly attractive substrate for our efforts towards efficient parallel simulation on multi-processor systems. However, despite an increasing support in terms of general purpose programming capabilities, GPUs are still highly specialized hardware which implement a considerably different processing model than CPUs. As a result, successfully exploiting the available processing power of GPUs is difficult. We address this challenge by designing a novel multi-level parallelization scheme that utilizes the massively parallel processing power of GPUs to achieve a cost- and time-efficient execution of large scale parameter studies.

The remainder of this chapter is structured as follows: At first, we motivate and sketch our approach to using GPUs for parallel simulation in Section 5.1. Then, Section 5.2 analyzes in the architecture of GPUs and identifies the challenges that arise from this architecture when attempting to execute parallel event simulations on GPUs. With these challenges in mind, we review state-of-the-art efforts targeting GPU-based parallel event simulation in Section 5.3 and discuss their shortcomings. Based on these two sections, we then introduce the design of our multi-level parallelization scheme in Section 5.4. Section 5.5 puts multi-level parallelization in the context of parallel expanded event simulation and discusses fundamental limitations of our approach. In Section 5.6, we then detail on the properties of our prototype implementation which we use as a basis for evaluating our scheme in Section 5.7. Finally, we conclude this chapter in Section 5.8.

5.1 Motivation

Complex technical systems, such as wireless communication networks, inherently provide a multitude of tuning parameters. In order to find an optimal configuration of a system (in terms of a specific goal), a thorough exploration of the given design space is necessary. This design space exploration is typically performed by means of detailed simulation models and elaborate parameter studies, often involving a large number of simulation runs. However, even if a single simulation run finishes quickly, the total combined runtime needed to complete a parameter study can become considerably large, thereby severely hampering the design space exploration process.

Despite existing techniques for reducing the number of relevant system parameters, e. g., factorial design [Jai91], studying only a few parameters quickly results in large amounts of simulations. For example, a study over five parameters, each with five distinct values of interest, requires simulating $5^5 = 3125$ different parameter sets. Additionally, in order to obtain statistically credible results, all parameter sets need to be repeatedly executed with varying random seeds. Considering the 3125 parameter sets and 30 repetitions, a total of nearly 100,000 distinct simulations runs are necessary.

In this chapter, we present a parallel discrete event simulation scheme that enables a cost- and time-efficient execution of large scale parameter studies on GPUs. Our approach leverages the massively parallel processing power of GPUs to *concurrently* execute all individual runs of a parameter study. The overall goal of our work is to provide a cost-efficient alternative to the traditional approach of distributing the simulations of a parameter study to multiple CPUs. We argue that conducting large scale parameters studies on a single GPU might in fact be slower than running them on a *large* number of CPUs, yet purchasing one consumer level GPU is significantly cheaper than buying and maintaining a large number of CPUs. Hence, our approach constitutes a trade-off between cost and processing power.

General Idea

In order to successfully exploit the massively parallel GPU-architecture, we need to address two particular challenges: i) GPUs implement a Single Instruction Multiple Threads (SIMT) processing paradigm in which groups of threads execute in lockstep, and ii) due to limited onboard memory, data needs to be continuously transferred between host- and GPU-memory, hence imposing large memory access latencies. Our key contribution in this chapter is a multi-level parallelization scheme that overcomes these challenges [KSGW12a, Sch11]. In particular, the scheme builds on two orthogonal levels of parallelism:

External Parallelism: We refer to the fact that individual simulations of a parameter study are inherently independent as *external parallelism*. Assuming that the simulations of a parameter study behave similarly, external parallelism enables the generation of SIMT-compatible workload by aggregating similar events from all simulations and streaming them to the GPU in a single batch.

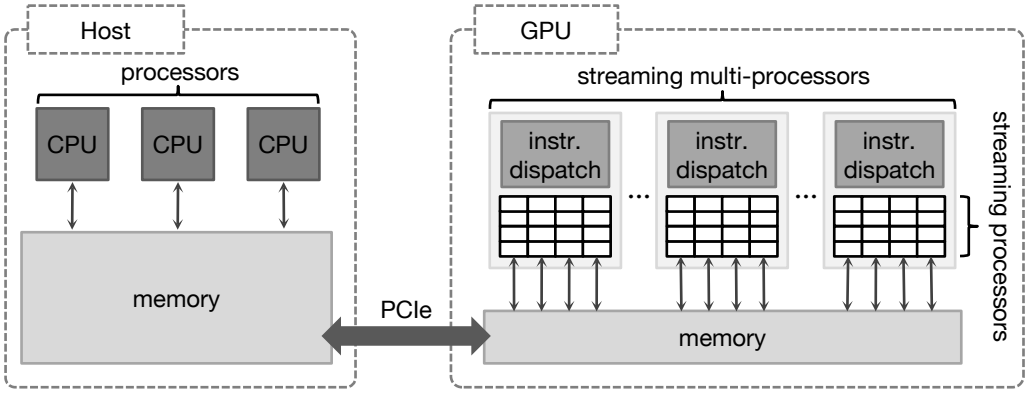


Figure 5.1 Simplified illustration of a typical GPU architecture. Each streaming multi-processor contains multiple streaming processors which share a common instruction dispatch unit. For example, a Nvidia GTX 470 GPU comprises 14 streaming multi-processors with 32 streaming processors each.

Internal Parallelism: Within the simulations comprising a parameter study, individual events are independent, denoted as *internal parallelism*. By interleaving the transfer to and from GPU-memory with the execution of independent events on the GPU, internal parallelism allows for establishing a pipelined event execution that hides memory access latencies.

Based on a proof-of-concept prototype, we analyze the performance characteristics of the proposed scheme by means of synthetic benchmarks. Moreover, we conduct a case study using a wireless network model. We show that our parallelization scheme reduces the runtime demand in this case study by a factor of 25 over an equivalent CPU-based implementation.

5.2 Challenges of Integrating GPUs with PDES

Motivated by intriguing parallel performance and fueled by general purpose programming frameworks such as NVIDIA's CUDA or OpenCL, GPUs have become an invaluable tool for high-performance computing. For instance, the NVIDIA GTX 470 graphics card used for this work provides a total of 448 processing cores, organized in 14 streaming multi-processors, which again consist of 32 streaming processors each. Supported by hardware assisted thread scheduling, GPUs allow for massively parallel processing by concurrently executing an even greater number of threads. However, in order to fully exploit the available processing power, two peculiarities of the GPU architecture need to be taken into account:

i) Threads are organized in groups which execute in lockstep, and
ii) limited onboard memory necessitates copying data to and from host memory.

We discuss both aspects in the following and show why they pose a particular challenge in the context of PDES.

```
Procedure: gpuKernel()
1: if threadId % 2 == 1 then
2:     doSomething()
3: else
4:     doSomethingElse()
5: end if
6: doSomethingTogether()
```

Algorithm 4 Example for lockstep execution on GPUs. Due to a shared instruction dispatch unit, all threads in a warp execute the two diverging code paths of the `if`-statement sequentially.

5.2.1 Lockstep Execution of Threads

In NVIDIA's Fermi architecture, a group of 32 threads forms a so called *warp* [NVI]. Since the threads in a warp execute on a common multi-processor, they share a single instruction dispatch unit (see Figure 5.1). As a result, all threads in a warp execute the same instructions in *lockstep*, implementing the Single Instruction Multiple Threads (SIMT) processing paradigm.

Nevertheless, threads can selectively mask instructions of a common instruction stream, thereby exposing different code paths embedded within this stream. However, since the instructions are part of a common instruction stream, the threads currently masking an instruction are effectively idle and need to wait for the masked instruction to finish. Algorithm 4 illustrates the resulting problem by means of a simple example. Assume `threadId` is a variable holding the unique ID of the threads in a warp. The conditional `if`-statement then causes all threads with an odd ID to process the function `doSomething` while all threads with an even ID process the function `doSomethingElse`. Since both functions comprise different sets of instructions from the common instruction stream, at first all threads with an odd ID process the instructions within `doSomething` while the threads with an even ID are idle. After finishing `doSomething`, the threads with an odd ID are idle while the threads with an even ID execute the instructions of `doSomethingElse`. As a result, the GPU executes both functions *sequentially*. Fortunately, the CUDA runtime automatically synchronizes both groups of threads after executing diverging code paths, so that all threads jointly execute the function `doSomethingTogether`.

In stark contrast to the processing paradigm of GPUs, parallel discrete event simulation executes *independent* events in parallel. In general, those events do not comprise the same code but instead model unrelated aspects of the simulated system. Consequently, when mapped to the same streaming multi-processor, independent yet unrelated events are still executed sequentially by the GPU since they involve different instructions. Concluding, in order to efficiently integrate the lockstep execution paradigm of GPUs with parallel discrete event simulations, we need to design an event processing strategy that provides SIMT workload to the streaming multiprocessors of the GPU.

5.2.2 Memory Size, Latency, and Control Overhead

In comparison to the amount of main memory provided by a typical desktop computer (∼8 GB) or server (∼32 GB), the size of GPU memory is relatively small (∼1.5 GB). Hence, data is commonly held in host memory and transferred to GPU memory on demand, modified there, and finally copied back to host memory (see Figure 5.1). Unfortunately, such memory transfers across the Peripheral Component Interconnect Express (PCIe) bus suffer from significant latencies and can easily become a performance bottleneck.

A common approach towards mitigating the adverse performance effects of these memory transfers is latency hiding [HJPM10]. The key idea is to perform three operations *concurrently*: i) Memory transfers to the GPU, ii) memory transfers from the GPU, and iii) actual GPU processing. This three-stage pipelining keeps the PCIe bus as well as the streaming processors of the GPU busy. In order to achieve a fully pipelined execution of events in parallel discrete event simulations, all events in the three stages of the pipeline need to be independent. Thus, the event scheduler has to identify independent events and synchronize their execution accordingly.

Moreover, each memory transfer over the PCIe bus involves a control overhead that favors large transfer units (1-1000 K) over small ones (1-1000 bytes) [HJPM10]. However, the memory footprint of a single event in a discrete event simulation is typically small. Hence, transferring individual events to and from GPU-memory suffers from the control overhead inherent to small transfer sizes. As a result, efficiently utilizing GPUs requires hiding the memory latency and access overhead.

5.3 Related Work

Despite the challenges outlined in the previous section, GPUs lay the foundation for improving the performance of a wide range of different types of simulations, including simulations of physical processes [YWC07], computer architectures [MCM11], vehicular networks [PAYS09], Monte Carlo simulations [PVPS09], etc. A comprehensive survey by Owens et al. [OLG+07] gives a broad overview of the subject. In this thesis, we focus exclusively on parallel discrete event simulations. Since this field comprises a large body of research efforts [Fuj90a, Per06b], this section discusses only closely related work.

5.3.1 Integrating GPUs with PDES

Parallel discrete event simulations can utilize GPUs either solely as potent co-processors or execute the entire simulation on the GPU. We discuss both variants in the following.

5.3.1.1 GPUs as Co-processors

The majority of related efforts integrate GPUs as co-processors to which complex computations are offloaded. The core logic of the simulation framework, e. g., event

schedulers and event queues, remain in host memory and run on the CPU. In this context, Bauer et al. [BMP08] investigate the applicability of GPUs to combined simulations [ZPK00] in which the discrete component of the simulation (the event scheduler) executes on CPUs while the continuous component (the event handlers) runs on GPUs. Using a synthetic workload model, the authors report considerable speedups for simulations containing computationally complex events. However, memory I/O turns out to be the primary performance bottleneck of this work which does not consider memory latency hiding techniques.

Following a similar approach, Xu et al. [XB07] identify sources of data- and task-parallelism within detailed network simulation models. In contrast to our work, this approach mainly relies on data-parallelism *within* complex events and does not explicitly attempt to achieve a high degree of task-parallelism *between* events.

SCGPSim [NPJS10] is a simulation framework focusing on SystemC simulations. Using source-to-source compilation of SystemC to CUDA-enabled code, it automatically maps sequentially executing SystemC threads to parallel threads on a GPU. This approach achieves a considerable speedup, but it inherently relies on the specific processing model of SystemC and hence cannot be applied in general to discrete event simulation.

5.3.1.2 Purely GPU-based Simulation

In contrast to limiting GPUs to mere co-processors, Perumalla [Per06a] explores the challenges of a purely GPU-based simulation framework. To account for the streaming-oriented processing model of GPUs, the traditional event scheduling loop underlying discrete event simulation is replaced by an event-streaming algorithm. Despite being a low-level GPU implementation, the approach indeed achieves a parallel speedup in a specific heat diffusion simulation. Although this work shows the feasibility of an entirely GPU-based simulation framework, the central question of general applicability remains unanswered. Moreover, this pioneering work suffers from the absence of general-purpose programming environments such as CUDA.

Park et al. [PF10, PF11] extend the previous work by developing a GPU-based event aggregation and execution scheme based on the concept of approximate time [Fuj99a]. While the proposed event aggregation scheme can indeed generate considerable performance improvements, it results in numerical errors. Although error analysis and approximation techniques allow for mitigating the amplitude of these numerical errors, this approach is as well not generally applicable.

Finally, Chatterjee et al. [CDB09] propose a fully GPU-based simulator for evaluating hardware designs on the gate level. In order to make efficient use of the GPU, the authors introduce a dedicated compilation phase in which the typically monolithic hardware model is segmented in smaller parallelizable tasks. Nevertheless, this approach heavily depends on the specifics of hardware logic simulations. In contrast, our proposed multi-level parallelization scheme is applicable to any discrete event simulation.

5.3.2 Efficient Execution of Parameter Studies

Simulation cloning [HF97, HF01, PGM08, PML08] reduces the amount of common computations across all simulations of a parameter study. Instead of executing a separate simulation for each parameter set, simulation cloning conducts only a single simulation which represents all possible execution paths within a parameter study. To branch into diverging execution paths, the simulation clones its current state at so-called decision points, i. e., events causing diverging behavior, and subsequently follows each path separately, but in parallel. As a result, the path segment up to the decision point is shared among both resulting simulation paths and thus only computed once. While simulation cloning can significantly reduce the total runtime of a parameter study, the primary drawback of this technique is its complexity and overhead due to state saving and maintenance.

5.4 Multi-level Parallelization on GPUs

Based on the previous analysis of the peculiarities of GPUs, we state two key design requirements for an efficient utilization of GPUs in parallel simulations: Specifically, the parallel simulation framework has to

i) *generate SIMT-compatible workload* to accommodate the lockstep execution paradigm of GPUs, and
ii) *hide the latency of memory transfers* between host- and GPU-memory.

In order to meet those two design requirements, our proposed solutions employ two orthogonal levels of parallelism. The first level, *external parallelism*, exploits the fact that individual simulations in a parameter study are trivially independent and can hence execute in parallel. External parallelism thus lays the foundation for an event aggregation scheme specifically designed for generating SIMT-compatible workload. The second level of parallelism, *internal parallelism*, makes use of the observation that within an individual simulation, groups of events are independent and thus allow for parallel processing. We exploit internal parallelism to hide the latencies involved in memory transfers.

Both levels of parallelism are not new in themselves but have been used in parallel simulation frameworks before. Instead, we claim that the combination of both schemes results in a novel parallelization scheme which unlocks the massively parallel processing power of GPUs for parallel discrete event simulations. The following sections introduce our approach in greater detail.

5.4.1 SIMT-compatible workload using External Parallelism

In order to meet the first design requirement, our GPU-based parallelization framework needs to generate SIMT-compatible workload to match the lockstep execution paradigm of GPUs. To this end, we design an event aggregation scheme that exploits *external parallelism*, i. e., the fact that parameter studies comprise multiple independent and self-contained simulations.

5.4.1.1 Event Aggregation Scheme

Before introducing the event aggregation scheme, we state the underlying assumptions and illustrate the idea by means of an example.

Observations and Intuition

Our event aggregation scheme rests on the following observations: In a parameter study, each individual simulation takes as input a specific combination of parameter values. Moreover, each combination of values is typically executed several times with different random seeds to obtain statistical confidence in the computed results. We argue that despite different parameterization, the individual simulations of a parameter study behave similarly since they encompass the same logic, i. e., model implementation. In particular, we expect the *order* of events in a simulation run to be similar across the individual simulations of a parameter study. In contrast, since each simulation uses a different set of parameters, the *state* of each simulation model in terms of the values of local variables differs.

To gain a better insight into this reasoning, consider an example of a parameter study investigating the throughput in a wireless network under varying channel conditions. In the underlying system model, a simple traffic generator triggers wireless transmissions of network packets according to a fixed rate. Since these transmissions are an inherent property of the simulation model, all simulations of the parameter study comprise send events and corresponding receive events which compute the received power. While computing the received power utilizes the same *algorithms*, i. e., code, at each receiver (e. g., pathloss, shadowing, and fading), the *concrete value* of the received power depends on the local state of the simulation, i. e., the distance between sender and receiver as well as the seed of the random number generators.

Hence, the fundamental idea of the event aggregation scheme is to execute the same event (handler) over a batch of different local states of each simulation.

Algorithm

Following the design of widely used simulators [HRFR06, Var01], we assume that simulation models exhibit a modular structure. Hence, we define the *event state* to be the values of the local data structures and variables of the module where the event takes place. Moreover, recall from Section 2.1.1 that the *event type* is uniquely defined by the specific event handler called when executing an event. Based on these assumptions and definitions, our approach towards generating SIMT-compatible workload for GPUs is as follows:

Given a parameter study consisting of multiple individual simulations, our scheme executes all these simulations concurrently in a round-based fashion. In each round, it first dequeues from all simulations the event with the lowest timestamp. Since the simulations supposedly exhibit the same behavior, these events are of the same event type. In contrast to the event type, the local state of each event is different. Hence, we aggregate the relevant states in a single batch (e. g., array) of event states and transfer this batch from host- to GPU-memory. Subsequently, the GPU executes the

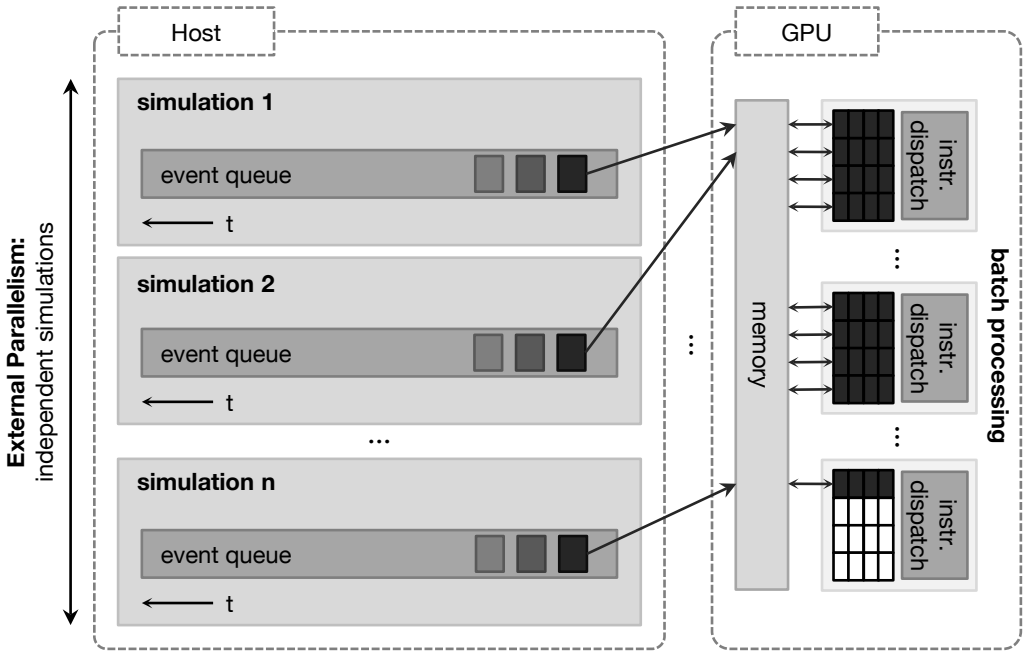


Figure 5.2 By exploiting the external parallelism between independent simulations of a parameter study, our event aggregation scheme creates SIMT-compatible workload for the streaming multi-processors of the GPU.

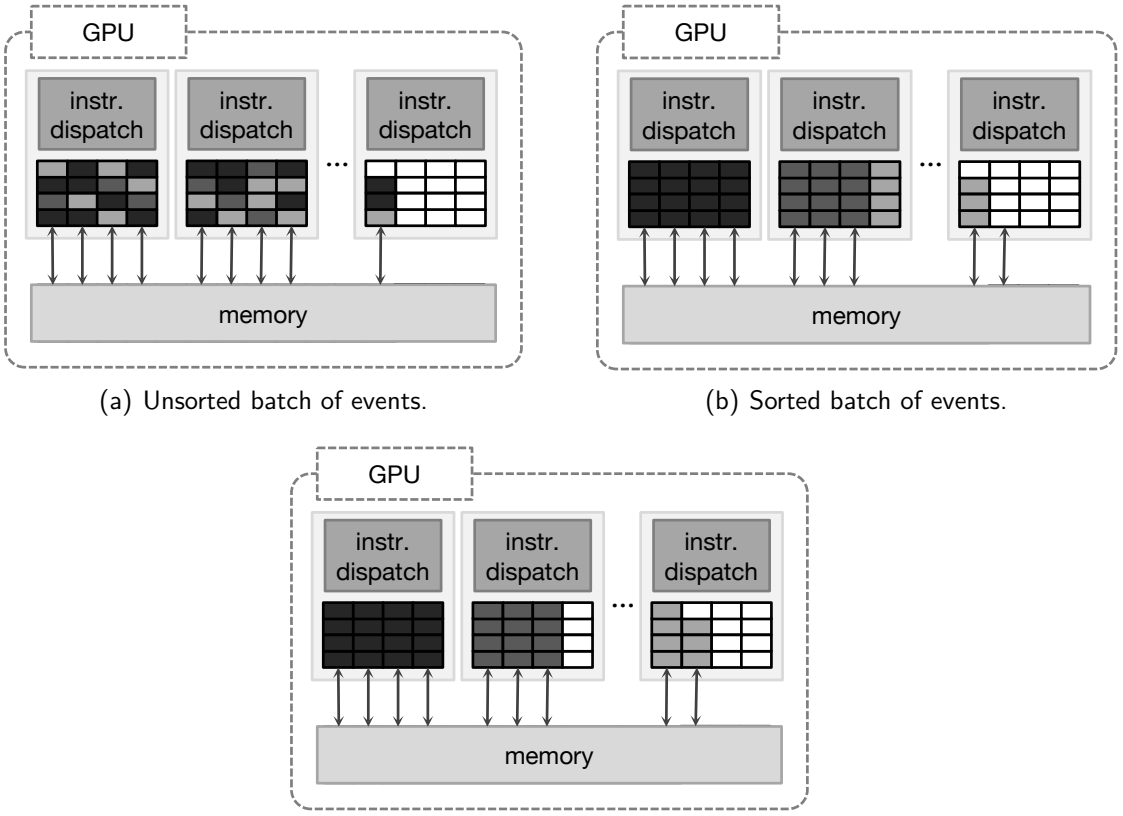
corresponding event handler on the array of event states. As a result, we generate SIMT-compatible workload by executing *one* event handler (i. e., a single set of instructions) on a batch of aggregated event states (i. e., multiple data) by means of *multiple* threads.

Figure 5.2 gives a schematic overview of the event aggregation scheme. In this simplified example, all n simulations behave identically, resulting in the same order and type of events in the corresponding event queues. Hence, by removing the first event from every event queue and aggregating the associated states in a batch, the streaming processors of the GPU can modify multiple event states while executing the same instructions.

5.4.1.2 Handling Divergent Simulations

Of course, we cannot expect all simulations of a parameter study to behave identically in all scenarios. Instead, it is more realistic to assume a divergent event ordering among simulations. Revisiting the example of a wireless transmission, a successful packet transmission triggers an ACK while an unsuccessful transmission causes a NACK. In this case, the first events of the simulations subsequently differ and the resulting event batches contain different event types in an arbitrary sequence.

The order of event states within a batch implicitly defines a mapping of events to the threads of a multi-processor, i. e., a warp. For instance, the first 32 events in the batch map to the threads of the first warp, the next 32 events to the threads of the second warp and so on. Hence, divergent simulations result in heterogeneous mappings in which the threads of a warp execute different event types (see Figure 5.3(a)). Consequently, the performance decreases due to the divergent code paths of the different event handlers.



(a) Unsorted batch of events.

(b) Sorted batch of events.

(c) Sorted and padded batch of events.

Figure 5.3 Diverging simulations cause a heterogeneous mapping of events to streaming multi-processors. By sorting and padding the batch of events, we create a homogeneous mapping that allows for truly parallel event processing.

To cope with divergent simulations, we exploit the fact that threads in *different* warps, i. e., running on separate multi-processors, can independently execute differing instructions. Thus, our approach is to identify and group events that exhibit identical runtime behavior, i. e., whose code paths do not diverge. Given such groups of equivalent events, mapping each group to independent multi-processors avoids blocking of threads and enables fully parallel execution. In the following, we group events on the basis of their type, i. e., event handler, to ensure a common code base and propose two simple mapping techniques: i) sorting and ii) padding.

Sorting

Our first mapping technique just sorts the events within a batch according to their type. The reasoning behind this simple approach is that a sorted batch of events increases the chances for assigning fewer different event types to one multi-processor. However, it cannot guarantee a clean 1-to-n mapping of event types to multi-processors and it generally performs poorly for highly heterogeneous event batches. Even in case of only few different event types, the simple sorting scheme cannot align the events in a batch to the boundaries of the multi-processors. Thus a group of events of the same type might span over two multi-processors despite actually fitting onto a single one. The light gray events shown in Figure 5.3(b) illustrate this case by mapping both to the middle and the right streaming multi-processor.

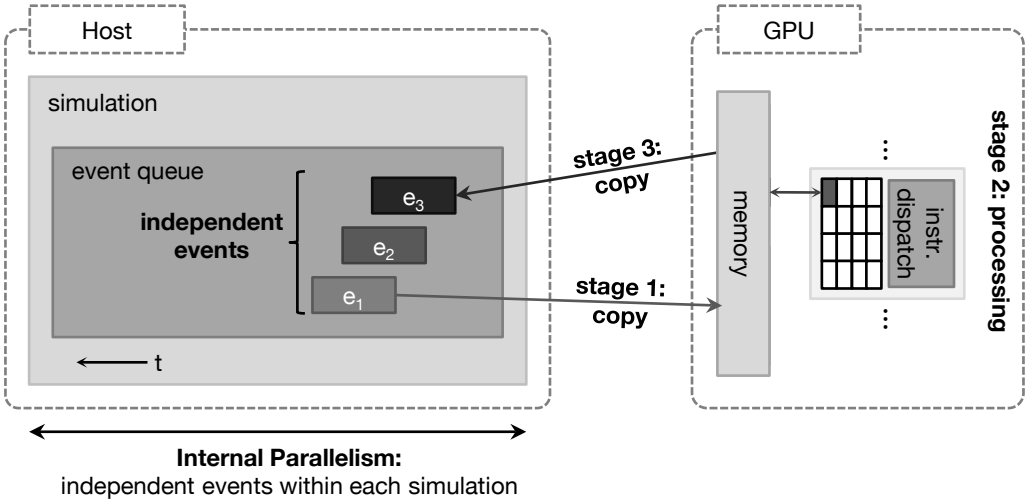


Figure 5.4 Internal parallelism utilizes independent events within a simulation to establish a three-staged pipeline for event execution: i) copy event states to GPU memory, ii) execute events, iii) copy event states back to host memory.

Padding

A straightforward extension of the sorting scheme is event padding. This scheme explicitly introduces gaps in a sorted event batch to achieve a clean 1-to-n mapping of event types to multi-processors and to align event types to the boundaries of multi-processors (see Figure 5.3(c)). However, despite achieving a homogeneous mapping, the gaps effectively decrease the utilization of GPU resources since the processors mapping to a gap cannot perform useful work. Hence, padding involves a trade-off between a decrease in resource utilization and a performance gain through homogeneous event-to-processor mappings. We analyze the performance of both simple mapping schemes in Section 5.7.1.1 and show that they indeed are able to mitigate the performance impact of divergent simulations. Future efforts nevertheless will focus on the development and implementation of more sophisticated mapping algorithms.

Both padding and sorting algorithms aim for generating SIMT-compatible workload on the level of event types. Yet, even if executing a single event type, i. e., event handler, the different states may still cause divergent code paths *inside* the corresponding event handlers (recall the conditional branch in Algorithm 4). However, since this kind of divergence depends on the parameterization, we believe that its performance impact is on average less severe than that of entirely different event types. Nevertheless, future mapping algorithms may take a more fine grained view on the event batch, for instance by incorporating the actual state of the events.

5.4.2 Hiding Memory Latencies using Internal Parallelism

In addition to external parallelism, we exploit *internal parallelism* to hide the adverse performance effects of memory transfers to and from the GPU. Internal parallelism relies on the fact that groups of events within a simulation model are independent. For instance, events computing the received power of a single wireless transmission at different receivers are independent of each other. Note that such independent

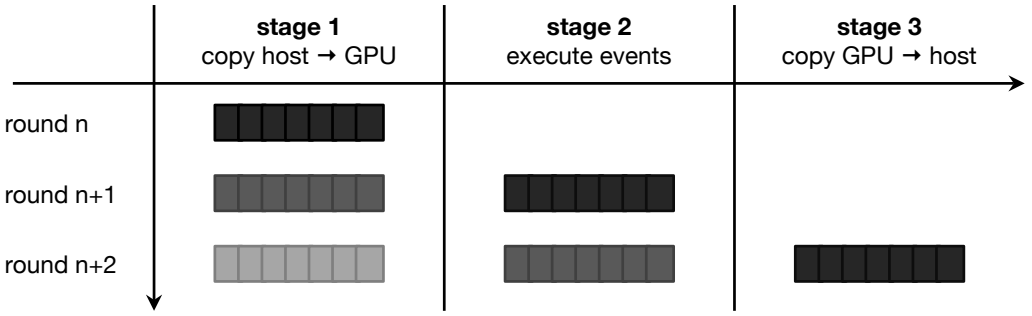


Figure 5.5 Example of three batches of independent events traversing the three pipeline stages.

events are the foundation of traditional PDES techniques [Fuj90a, Per06b], including parallel expanded event simulation and probabilistic synchronization presented before. However, in contrast to those approaches, we do not intend to just *execute* such events concurrently. Instead, we aim for a *pipelined handling* of independent events in order to hide memory transfer latencies.

Specifically, we decompose the event handling process into three steps and assign each step to a pipeline stage:

Stage 1: Copy event states from main- to GPU-memory.
Stage 2: Execute events on the GPU and update event states.
Stage 3: Copy modified event states from GPU- to main-memory.

Figure 5.4 illustrates the integration of the three pipeline stages in the event handling architecture.

To hide the memory transfer latency involved in stage 1 and 3, the pipeline executes all three stages concurrently and in a round based manner. In each round, every stage handles exactly one batch of independent events before the pipeline forwards the batches to the next stage. Figure 5.5 shows an example of this pipelined execution scheme by means of three independent batches.

In order to fully utilize the pipeline, i. e., to fill all three stages, the simulation framework needs to identify at least three batches of independent events. If fewer than three batches are available at any point in time during a simulation run, the pipeline stalls, resulting in empty stages and wasted processing resources. Hence, it is imperative to extract a maximum degree of internal parallelism from the simulation model. We thus utilize the parallel expanded event simulation paradigm for the reasons outlined in Chapter 3. Specifically, a simulation contributes an event to the batch of states collected in the current round if this particular event overlaps with all events of the two previous rounds.

5.5 Discussion

As the focus of this thesis is on workstation and server multi-processor systems, it is imperative to consider GPUs in order to fully explore the design space of multi-threaded simulation. Despite the challenges of integrating GPUs in parallel event simulation, we argue that these efforts are worth the investment: Currently,

GPUs merely act as potent co-processors, yet their architecture foreshadows the massively parallel hardware of future multi-processor systems. Hence, our analysis of the applicability of GPUs and our proposed multi-level parallelization scheme constitute a fundamental building block of this thesis and towards future massively parallel simulation.

5.5.1 Integration of Parallel Expanded Event Simulation

Internal parallelism utilizes conservative synchronization to identify independent events within each simulation. As briefly sketched in Section 5.4.2, we propose using the parallel expanded event simulation paradigm which conservatively analyzes the overlapping of expanded events. Hence, multi-level parallelization integrates seamlessly into this thesis.

Conceptually, nevertheless, multi-level parallelization can utilize any conservative event synchronization scheme. In fact, we can hide the actual synchronization scheme behind a simplistic *getNextEvent*-interface: Via this interface, the multi-level event scheduler queries the synchronization scheme for the next available independent event at the beginning of each new round. If no such event exists, the interface returns an empty event. Moreover, by keeping track of the number of queries and the events returned upon the three previous requests, the interface notifies the synchronization algorithm about the completion of offloaded events. The latter is important since the synchronization scheme needs to know when offloaded events that block dependent events from parallel execution are complete. Only after completely executing such blocking events, formerly dependent events become eligible for parallel processing.

Summarizing, multi-level parallelization is orthogonal to parallel expanded event simulation. Yet, both schemes complement each other, thereby integrating seamlessly within the context of this thesis.

5.5.2 Restrictions of the Programming Environment

In contrast to parallel expanded event simulation, we do not integrate GPU-based multi-level parallelization with Horizon. This design decision is rooted in restrictions of the programming environment supported by GPUs. In order to foster general purpose programming on GPUs, vendor-specific programming environments (e. g., Nvidia Compute Unified Device Architecture (CUDA) and AMD Accelerated Parallel Processing (APP)) and vendor-independent programming environments (e. g., OpenCL) abstract from low-level shader programming. Instead, they strive to achieve a (transparent) mapping of GPU functionality to general-purpose programming languages such as C (OpenCL) and C++ (Nvidia CUDA and AMD APP). However, due to differences in the processing model and performance considerations, only a limited set of the features provided by the host programming languages are available on the GPU.

OMNeT++ and Horizon make heavy use of C++ polymorphism, i. e., virtual functions, to incorporate user defined simulation models in a general simulation engine.

However, none of the currently available programming environments fully support C++ polymorphism. As a result, combining the proposed multi-level parallelization scheme with HORIZON would require extensive re-engineering of HORIZON and existing simulation models. Instead, we designed and implemented a proof-of-concept prototype that is heavily inspired by the modeling concepts of HORIZON, but is otherwise kept simple and compatible with GPU capabilities. Section 5.6 discusses the implementation of this prototype in further detail.

5.5.3 Limited GPU-Memory

A further limitation of our approach is the relatively small size of onboard memory which restricts the number and the size of event states that can reside in GPU-memory. For instance, assuming a parameter study with 500 simulations and a 3-staged pipeline, a maximum of 1500 events states need to be stored in GPU-memory. Assuming furthermore a typical consumer GPU with 1.5 GB of memory, each event state may not be larger than 1 MB, if the GPU does not reserve memory for the graphical user interface. However, we argue that this limitation does not impose severe restrictions on model developers in practice: First, the growing popularity of general purpose computing on GPUs will likely foster a strong increase in the size of GPU memory. Second, in our prototype, the event state contains only data which is transferred between the modules of a simulation model and which is typically limited in size. Third, vendor specific memory management features, such as NVIDIA's Unified Virtual Addressing (UVA), expose the entire host memory to the GPU at the price of increased access latency. We detail on the last two aspects in the next section.

5.6 Implementation

We developed a proof-of-concept prototype to investigate the viability of our multi-level parallelization scheme. Due to the reasons outlined in Section 5.5.2, the prototype does not directly build on HORIZON, yet its architecture and the modular structure of its simulation models are inspired by OMNeT++ [Var01] and HORIZON. We select Nvidia's CUDA as GPU programming and execution environment because it belongs to the most sophisticated frameworks available at present. The next sections briefly cover the fundamental properties of our prototype implementation.

5.6.1 Programming Interface

The prototype consists of a CPU-bound simulation core and GPU-located simulation models, i. e., event handlers implemented as GPU kernels. Despite being a prototype, ease of use from a modeler's perspective is an explicit design goal of our implementation. This includes particularly the programming interface and modeling API. Hence, our framework exports a typical discrete event simulation interface that abstracts from GPU-programming and even CUDA. It primarily provides access

to random number generators and allows for creating and scheduling new events. Hence, the model implementation effort is comparable to typical CPU-based simulators such as OMNeT++ or ns-3 [HRFR06] as the interface hides the complexities of GPU programming. For instance, to avoid the considerable overhead of dynamic memory allocation on the GPU, the framework creates new events in a specific buffer provided by each event state. After copying the event state and the buffer back to host memory, newly created events are removed from the buffer and enqueued in the event queues of the corresponding simulations.

5.6.2 Memory Management

The prototype implementation utilizes NVIDIA's Unified Virtual Addressing (UVA). UVA provides a single virtual address space spanning host- and GPU-memory, thereby significantly increasing the amount of memory available to the GPU. The drawback of UVA however is a severe performance penalty when accessing data in unified memory from the GPU as it needs to be fetched from host memory. Hence, to achieve a compromise between fast yet limited GPU memory and slow yet large virtual memory, we restrict the event state to comprise solely the "payload" of events:

Analogously to OMNeT++ and Horizon, simulation models for our prototype attach payload data to events to exchange information between modules. For instance, an event modeling the reception of a wireless transmission might carry as payload the size of the received data, the modulation, and the ID of the sender. In contrast, data structures local to the module handling this event, e. g., a list of received transmissions, remain in UVA and can hence become (arbitrarily) large.

Since modules typically transfer only a fraction of their local state, the event payload is much smaller than the local state of a module. As a result, this approach decreases memory utilization on the GPU and minimizes the size of memory transfers between host- and GPU-memory. Moreover, it gives model designers explicit control over the event state size and the corresponding memory transfer overhead.

5.6.3 Pipelined Execution

We utilize multiple CUDA streams to implement pipelined event execution. Each stream is part of a *simulation driver* which performs four tasks in a round based fashion. In each round, a simulation driver

i) collects one event from each simulation,
ii) writes the corresponding event states to its CUDA stream,
iii) launches the event handling kernel, and
iv) reads modified event states from the CUDA stream.

By interleaving the rounds of multiple simulation drivers, our implementation establishes a pipelined event execution. In the current prototype, these CPU-based tasks execute sequentially in a single thread. We leave the extension of the prototype to a multi-threaded architecture for future work and focus instead on the GPU-related challenges.

Parameter	Value(s)
Simulations (external parallelism)	$2^i, i \in \{0, 1, 3, 5, 7, 9, 10, 11, 12, 13, 14\}$
Modules (internal parallelism)	1, 2, 5, 10
Events per simulation	300
Event state size	72 bytes
LCG iterations per event	20000

Table 5.1 Synthetic benchmark parameters.

5.7 Evaluation

This section evaluates our multi-level parallelization scheme based on our proof-of-concept implementation. We analyze the performance properties of the parallelization scheme in terms of the performance impact of divergent simulations and the event handling overhead. In order to precisely control these parameters, the evaluation employs a set of synthetic benchmarks we introduce separately in the following sections. Moreover, to get an impression of the potential performance gain in a real-world scenario, we complement the synthetic benchmarks with a case study based on an abstract wireless mesh-network model.

The benchmarking platform is a workstation PC providing an AMD Phenom II X4 945 4-core CPU with 8 GB of main memory and one NVIDIA GeForce GTX 470 GPU accommodating 1.28 GB of memory. The simulation framework runs on a 64 bit version of Ubuntu 10.10 with NVIDIA's proprietary drivers in version 290.10. Finally, we enable full optimizations using the NVIDIA CUDA compiler, nvcc, in version 4.0 and g++ in version 4.4. Furthermore, each data point shows the mean and the 99 % confidence intervals computed over 30 independent repetitions. Nevertheless, the confidence intervals are barely visible due to highly consistent performance results.

5.7.1 Synthetic Benchmarks

We utilize two different synthetic benchmarks to investigate i) the impact of divergent simulations on parallelization performance, and ii) the event handling overhead of our framework. The following sections introduce both benchmarks in detail and analyze their respective results.

5.7.1.1 Divergent Simulations

The first synthetic benchmark analyzes the performance impact of divergent simulations. It particularly investigates the effectiveness of the simple sorting and padding algorithms outlined in Section 5.4.1.

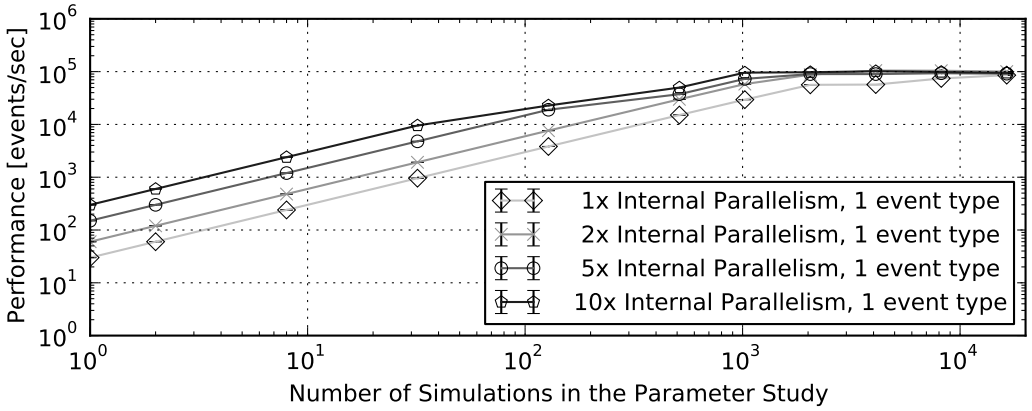


Figure 5.6 Optimal (baseline) simulation performance using non-divergent and computationally complex simulations. The performance increases linearly along both levels of parallelism until the GPU is saturated for large numbers of simulations.

Methodology

The performance impact of divergent simulations is most pronounced for computationally complex events which hide all other performance effects and overheads. Hence, the benchmark uses Linear Congruential Generators (LCGs) for generating one pseudo random number per event. All modules of the benchmark model encapsulate one LCG and continuously re-schedule a single local event in fixed intervals. To create divergent behavior, the benchmark employs different LCGs per module, each one resulting in a different code path and hence event type. Internal parallelism is controlled via the number of modules in the model as we consider events with equal timestamps on different modules to be independent. The size of the event state is 72 bytes which includes the minimum set of meta-data required by the framework for handling events. Moreover, each simulation comprises a static workload by processing a fixed number of events. We measure the performance in terms of the average number of events processed per second. Table 5.1 shows an overview of the set of parameters and their values.

Convergent Simulations

In order to quantify the performance impact of divergent simulations, we first need to determine the maximal achievable performance in a non-divergent scenario. Hence, Figure 5.6 shows the performance over varying degrees of internal and external parallelism if only one event type is present in the simulations. Focusing on an internal parallelism of 1, we observe a perfect linear performance increase up to 2048 simulations (note both logarithmic scales). Hence, this computationally dense benchmark indeed constitutes an ideal case for the massively parallel processing power of the GPU. Beyond 2048 simulations however, the GPU is fully saturated, achieving no additional speedup.

In addition, the performance increases linearly with the level of internal parallelism from a 2-fold speedup up to a 10-fold speedup. Considering the three stages of the event state transfer pipeline, one would expect a maximum performance increase by a factor of 3. However, the reason for the additional speedup is that our benchmarks

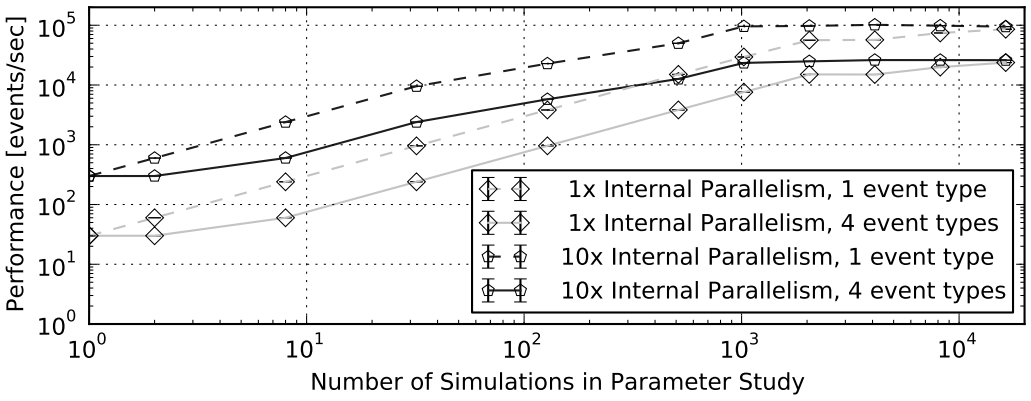


Figure 5.7 Comparison of non-divergent (dashed lines) with 4-way divergent (solid lines) simulations w/o sorting and padding of events. The simulation performance drops by a factor of 4 due to divergent code paths.

utilize ten concurrent simulation drivers, each copying a batch of event states, executing the events on the GPU, and copying the event states back. Hence, according to the level of internal parallelism, the benchmark keeps 1, 2, 5, or 10 of the simulation drivers busy. A larger number of simulation drivers in turn handle more concurrent events, thereby achieving a better utilization of the hardware resources, i. e., the PCIe bus and the GPU. In particular the GPU employs hardware based thread scheduling that schedules multiple threads (sequentially) onto one streaming processor in order to hide I/O and instruction fetch latencies. As a result, the GPU performs best if the number of threads, i. e., the total number of events, exceeds the number of physical streaming processors.

Divergent Simulations

After establishing the optimal performance as baseline, we now consider 4-times divergent simulations. To this end, the modules of each individual simulation select one out of four different LCGs such that each batch of event states comprises exactly four different event types. Figure 5.7 compares the baseline performance (dashed lines) with the performance of a 4-times divergent configuration (solid lines) without applying sorting or padding.

We clearly observe a consistent 4-fold performance decrease over all values of internal and external parallelism larger than 2. For two simulations, the batch of events comprises only two events for parallel processing. Hence, the drop in performance is limited to a factor of 2. This confirms the expected adverse effects of divergent simulations. Moreover, the fact that the performance drops exactly by a factor of 4 for more than 2048 simulations shows that in this benchmark the GPU is the performance limiting component and not the PCIe bus.

Divergent Simulations using Sorting

Sorting the events before execution yields the performance results shown in Figure 5.8. Similarly to the non-sorted and non-padded case discussed before, a 4-fold decrease in simulation performance remains for up to 64 simulations. However, starting with 128 simulations, the performance of the divergent benchmark matches the

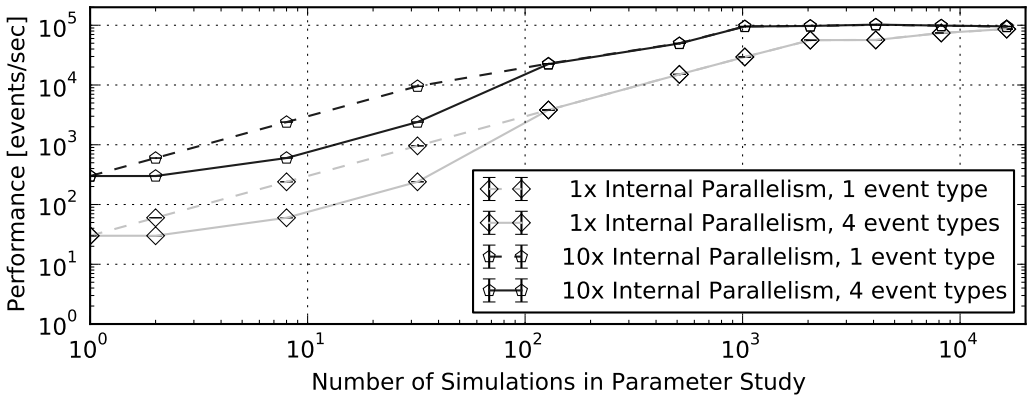


Figure 5.8 Comparison of non-divergent (dashed lines) with 4-times divergent (solid lines) simulations with sorting but w/o padding of events. For more than 128 simulations, sorting achieves a perfect mapping of events to multi-processors, resulting in a perfect performance recovery.

ideal performance of the convergent one. We ascribe this to the fact that the particular GPU used in the benchmarks utilizes a warp size of 32 threads. Therefore, in case of 4 different event types and 128 simulations, the simple sorting scheme *coincidentally* achieves an ideal mapping of exactly 32 identical event types to each warp. The same also holds true for the remaining data points as the number of simulations is always divisible by 32 without remainder. However, we deliberately chose these particular measurement points to demonstrate this characteristic of the sorting algorithm. In real scenarios, such a perfect mapping constitutes an ideal case which occurs only rarely. Hence, sorting is not sufficient to mitigate the performance loss inflicted by divergent simulations.

Divergent Simulations using Padding

A more promising approach to realizing an ideal mapping of event types to warps is to pad the sorted batch of event states as sketched in Section 5.4.1. Applying this technique to the 4-times divergent benchmark yields the results presented in Figure 5.9. The figure clearly illustrates that for an internal parallelism of 1, divergent and convergent simulations achieve the same performance. Furthermore, for an internal parallelism of 10, the results show a significant improvement over sorting, however, the performance still remains slightly below the non-divergent case. This is due to the additional computational overhead caused by the padding algorithm as well as a less efficient utilization of the multi-processors due to gaps in the event batch. We particularly blame the latter for the performance drop at 8 and 32 simulations.

Summary

This benchmark confirms the negative influence of divergent simulations on parallel simulation performance. It shows moreover that padding the batch of event states successfully mitigates the performance impact while the effectiveness of sorting highly depends on the particular scenario.

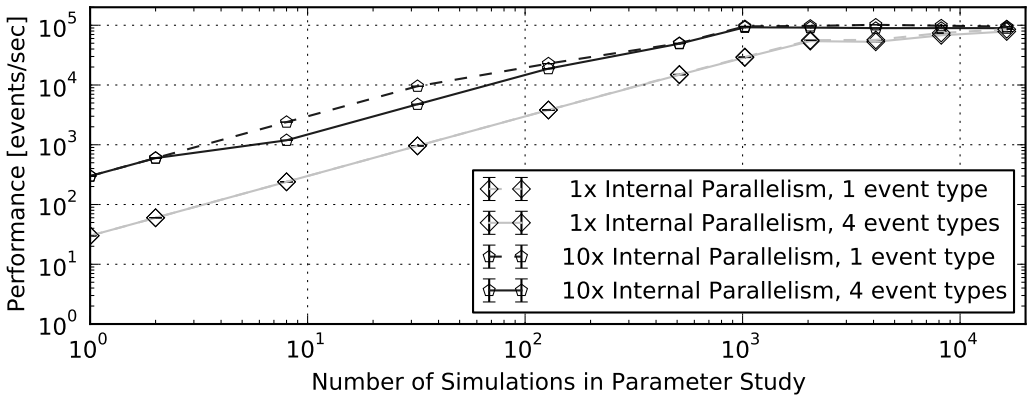


Figure 5.9 Comparison of non-divergent (dashed lines) with 4-times divergent (solid lines) simulations with sorting and padding of events. Padding achieves a significant performance recovery but causes a noticeable runtime overhead.

5.7.1.2 Overhead

The overhead of our prototype implementation comprises three distinct components:

i) The *event handling overhead* of all event management operations in the framework, such as creating, deleting, enqueueing, and dequeueing of events,
ii) the *event mapping overhead* imposed by the sorting or padding algorithms, and
iii) the *memory transfer overhead* caused by copying event states between host- and GPU-memory over the PCIe bus.

All three kinds of overhead are tightly coupled and influence each other. In this section, we analyze their interaction and their performance impact in detail.

Methodology

To measure the runtime overhead, we use a synthetic benchmark model which does not perform any computations in the event handlers apart from continuously rescheduling new events. As a result, the runtime of this simulation model constitutes a direct measure for the total overhead. In contrast to the benchmark model used in the previous section, this model employs a fixed workload for the *whole* parameter study. Since we use the runtime as a measure for the overhead, the workload has to remain constant when changing the benchmark parameters of interest. Thus, each parameter study executes a fixed number of events, which are equally distributed across all simulations and modules within the simulations.

Based on this model, we analyze the overhead under consideration of the degree of internal and external parallelism, the event mapping algorithm, and the event state size.

Event Handling Overhead

This benchmark measures the event handling overhead for different degrees of internal and external parallelism. As in the previous benchmark, we vary the level of

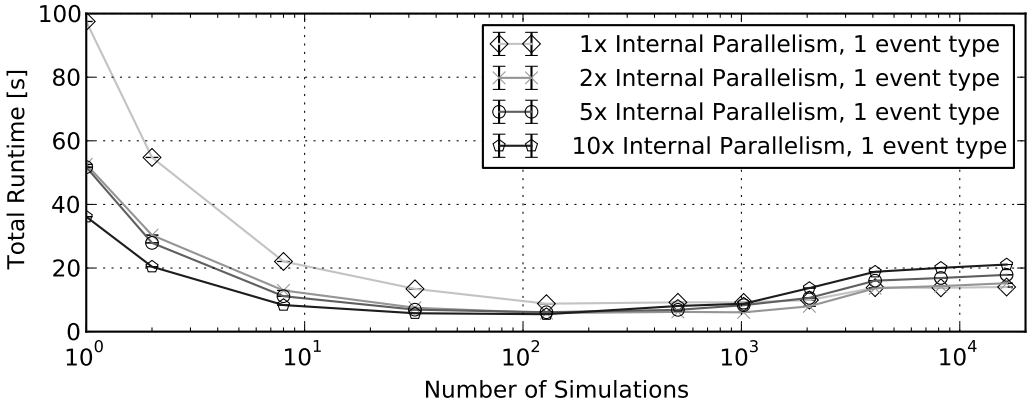


Figure 5.10 Runtime of the synthetic overhead benchmark indicating that the event handling overhead is effectively parallelized with an increasing number of simulations. Up to 4096 simulations, a higher degree of internal parallelism furthermore reduces the overhead due to memory latency hiding. Beyond 4096 simulations the GPU is overloaded.

external parallelism between 1 and 16384 and set the level of internal parallelism to 1, 2, 5 and 10. We furthermore fix the event state size to 72 bytes and use the padding-based event mapping algorithm since it delivers the best mapping results. Nevertheless, since the events in this benchmark only re-schedule themselves using the same API function call, all events are of the same type. Consequently, the benchmark exhibits perfectly convergent simulations.

Figure 5.10 shows the resulting total runtimes of the benchmarks. Focusing on the results obtained for an internal parallelism of 1, we observe a nearly linear decrease in the total runtime when increasing the number of simulations from 1 to 128 (note the logarithmic scale of the x-axis). From these results we conclude that the GPU is actually under-utilized in this scenario: By increasing the size of the event batch (due to increasing external parallelism) the overhead is distributed across more events, thereby improving the GPU utilization and hence the efficiency. For 128 to 1024 simulations, the runtime converges towards a stable value which constitutes the best compromise between the number of events and the overhead. Finally, the runtime increases again for more than 2048 simulations, indicating that the GPU is overloaded.

Analyzing the results for the other values of internal parallelism indicates a similar behavior, yet even lower runtimes up to 1024 simulations. This demonstrates that the pipelining approach using multiple simulation drivers successfully hides memory transfer latencies and improves the utilization of the GPU. Nevertheless, the figure also reveals that the runtime increases again for more than 2048 simulations. Specifically, the larger the internal parallelism, the longer it takes to complete the benchmark, thus resulting in an inversion of the performance results. In these benchmark scenarios the GPU is in fact over-saturated due to contention on the CUDA streams.

Event Mapping Overhead

A type of overhead specific to our framework is caused by the event mapping algorithms. Figure 5.11 compares the runtimes of the benchmark model using i) neither

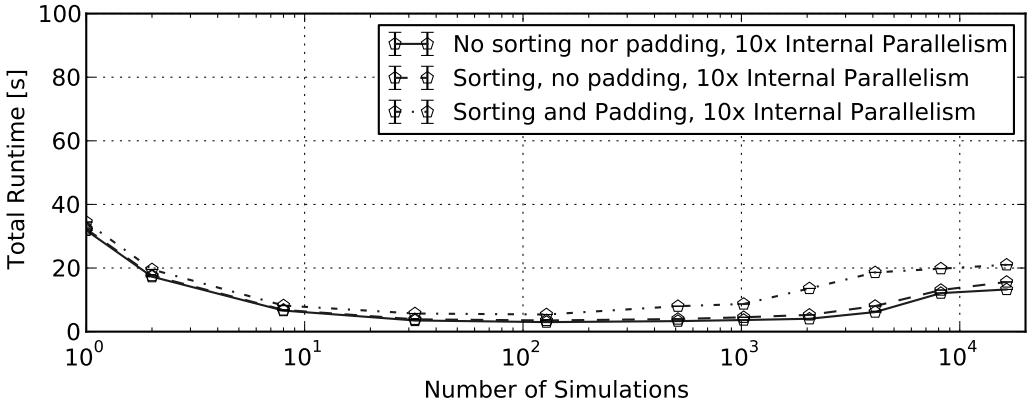


Figure 5.11 Comparison of the overhead introduced by sorting and padding of events. Padding adds a considerable overhead over sorting which in turn adds only a negligible overhead.

sorting nor padding, ii) only sorting, iii) sorting and padding. We again vary the degree of external parallelism between 1 and 16384, yet we use a static level of internal parallelism of 10 and an event state size of 72 bytes.

As expected, applying neither sorting nor padding results in the lowest runtimes, i. e., overhead. In comparison, sorting adds only slightly more overhead. This overhead, however, grows with an increasing number of simulations as the sorting operation becomes more complex. In contrast, the simple padding algorithm adds a considerable overhead, in particular for more than 1024 simulations. Nevertheless, as shown in Section 5.7.1.1, the performance gain of the padding scheme clearly outweighs the additional overhead.

Memory Transfer Overhead

Lastly, we investigate the influence of the event state size on the event handling overhead. To this end, we fix the number of simulations to 512 and stepwise increase the event state size from the minimum of 72 bytes to 26000 bytes. Figure 5.12 plots the resulting total runtimes for different values of internal parallelism. In general, we notice that the runtimes increase roughly linearly with the size of the event state. Hence, the event state size constitutes a crucial performance factor, urging model developers to keep the event state small.

Analyzing the lower two curves of the figure in detail, we observe that 2-fold internal parallelism performs slightly better than 1x internal parallelism. In contrast to this expected result, the figure illustrates that an internal parallelism of 10 causes a longer runtime than 5-fold internal parallelism which in turns takes more runtime than 1x and 2x internal parallelism. We ascribe this to the fact that with an increasing degree of internal parallelism, more CUDA streams concurrently read from and write to GPU memory, therefore causing more load and more contention on the PCIe bus. Hence, 2-fold internal parallelism achieves the best resource utilization in this benchmark, whereas 1x internal parallel under-utilizes the PCIe bus and 5x as well as 10x internal parallelization over-utilize the bus.

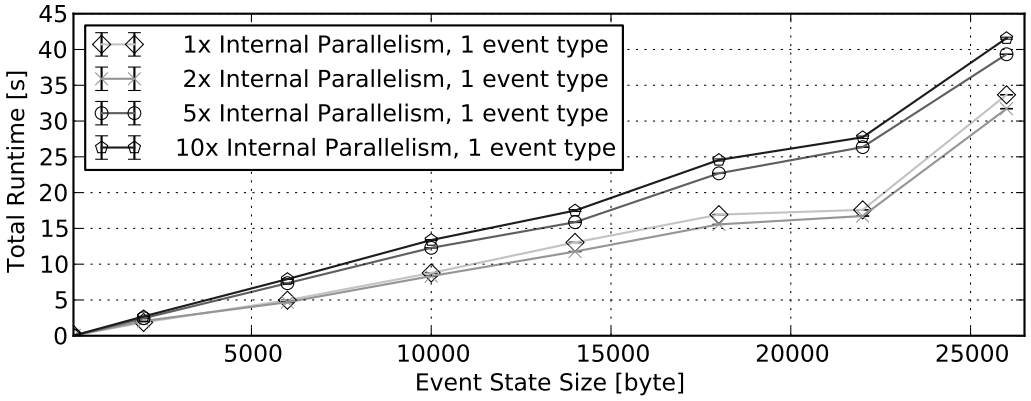


Figure 5.12 Comparison of the overhead of different event sizes and levels of internal parallelism. The overhead grows with the size of the events as well as the internal parallelism due to an increased load on the PCIe bus. The number of simulations is fixed to 512 in this benchmark.

Note however that the events of this benchmark model are by design computational insignificant. Hence, they do not consume enough runtime to outweigh the memory transfer overhead inherent to large event states.

Summary

Our evaluation shows that with increasing internal and external parallelism the event handling overhead per event decreases due to better resource utilization. Regarding the event mapping overhead, our benchmark furthermore confirms that padding induces a larger overhead than sorting, which in turn adds only a small overhead. Considering the results of the previous section, however, the performance gain due to padding outweighs the additional overhead. Finally, we observe that the size of the event states significantly influences performance, in particular due to contention on the PCIe bus for large event sizes.

5.7.2 Case Study

In addition to the synthetic benchmarks, we conduct a case study by means of an abstract wireless mesh-network model to get an impression of the user-perceived performance gain.

Methodology

The model simulates wireless transmissions based on an accurate and hence computationally complex channel and error model. For the sake of brevity, we do not introduce these models here but refer to Puñal et al. [PEG11] for detailed information. Moreover, the simulated network comprises 5 nodes connected in a fully meshed topology, i. e., every transmission is received by all nodes in the network as shown in Figure 5.13.

Each node consists of three separate modules: i) An application module which broadcasts packets at a fixed sending rate. ii) A MAC module which implements

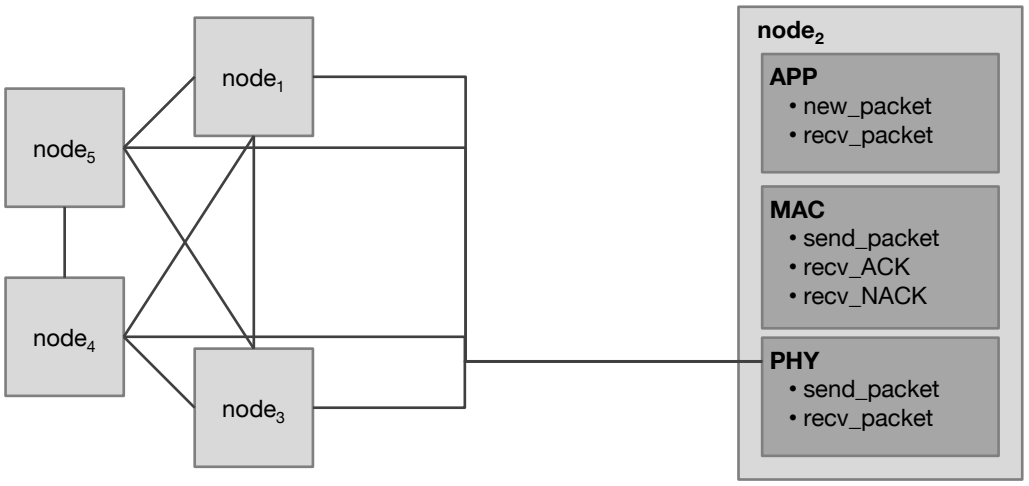


Figure 5.13 Illustration of the fully meshed topology of the abstract wireless network model. Furthermore, the right part shows the three components of a network node and the seven distinct event types occurring at those components.

the error model and a rudimentary MAC protocol: Each receiver sends an ACK or a NACK, depending on whether or not a transmission was successfully received. Note that sending different replies is a source of divergent behavior. iii) A PHY module which models the effects of the wireless channel. The model furthermore abstracts from the network and transport layer and disregards interference. Since events traverse the modules up and down the protocol stack, a total of 7 different event types occur in the model (see Figure 5.13). Furthermore, due to the fully meshed topology, every transmission is received by 4 nodes, resulting an internal parallelism of at least 4. Finally, the event size is 268 bytes to accommodate all meta-data needed for each transmission.

In order to create a simple parameter study, the model provides two independent parameters. First, the application module allows for configuring the packet generation rate. Second, the fading model of the wireless channel considers different movement speeds in order to model a dynamic environment. We vary the former between 1, 5, and 10 packets/s and the latter between 1, 5, and 10 m/s. In addition, every combination of parameter values is repeated 30 times with different seeds to obtain statistical confidence. Altogether, the parameter study comprises 270 simulation runs.

To allow for an accurate comparison between classic CPU-bound simulation and the multi-level parallelization scheme, the case study utilizes two versions of the wireless network model: A CPU-based version which builds on OMNeT++ and a GPU-based variant using our prototype simulation framework. We explicitly kept the implementation of both models as similar as possible despite the architectural differences of the underlying simulation frameworks. Hence, this case study aims to provide a rough impression of the potential performance gain while keeping the limitations of this particular comparison in mind. Moreover, the CPU benchmark does not utilize internal parallelization but solely employs external parallelism. We argue that any internal parallelization scheme using n CPUs is inherently slower than n independent sequential simulations because of additional synchronization overhead. Given the 4-core CPU of our benchmarking machine, we execute the CPU-bound parameter study in groups of 4 parallel simulations.

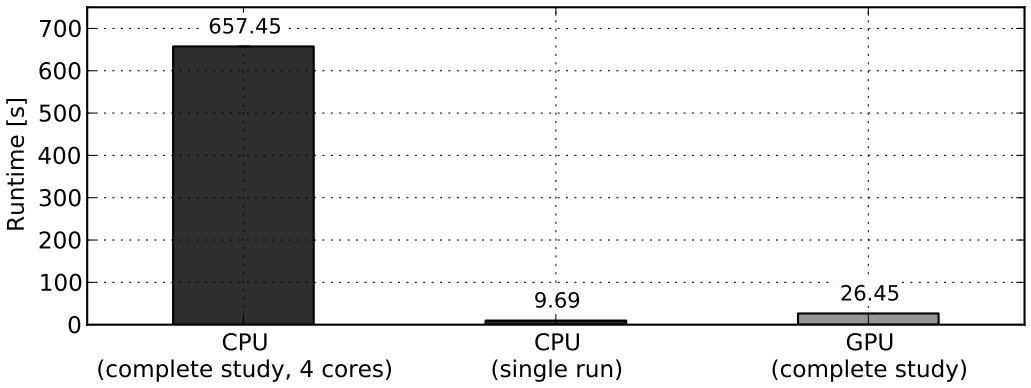


Figure 5.14 Runtime of the complete parameter study on a 4-core CPU (left) and on a GPU (right), as well as the runtime of a single CPU-bound parameter run (middle). GPU-assisted multi-level parallelization achieves a 25-fold performance improvement over CPU-based execution.

Results

Figure 5.14 compares the resulting total runtime of the CPU-based parameter study (left bar) with the total runtime of the GPU-based parameter study (right bar). The figure shows that our GPU-assisted approach significantly outperforms the traditional CPU-bound implementation by a factor of more than 25. This confirms the ability of our approach to enable a time-efficient execution of parameter studies. Moreover, the bar in the middle of Figure 5.14 illustrates the runtime of a single simulation on one CPU-core. It hence represents a distributed execution of the parameter study on 270 individual CPUs. This large-scale setup achieves an additional 2.7-fold performance increase over our single GPU-based implementation. However, the cost of purchasing and maintaining computers with a total of 270 CPUs by far exceed the cost of a single computer providing one consumer level graphics card. Thus, our approach constitutes a truly cost-efficient alternative to purely CPU-oriented large scale parallelization.

5.8 Conclusions

This chapter addresses our overarching goal to enable efficient parallel simulation on multi-processor systems by utilizing GPUs as massively parallel co-processors. Based on an analysis of the architecture of GPUs and the resulting challenges with regard to parallel event simulation, we presented a novel parallel discrete event simulation scheme that enables a cost- and time-efficient execution of large scale parameter studies. In order to efficiently bridge the substantially different processing paradigms of GPUs and parallel event simulations, our scheme exploits two levels of parallelism: i) External parallelism between the inherently independent simulations of a parameter study, and ii) internal parallelism among the events within each simulation. Building on these types of parallelism, we define an event aggregation scheme that generates SIMT-compatible workload, and a pipelined event copy scheme that hides memory transfer latencies.

Multi-level parallelization seamlessly integrates with expanded parallel event simulation. By utilizing the modeling paradigm underlying parallel expanded event

simulation, the multi-level parallelization scheme can efficiently identify independent events within a simulation, i. e., extract internal parallelism. As a result, both schemes successfully complement each other. Based on a proof-of-concept implementation, we obtained early performance results that underline the viability of the proposed multi-level parallelization scheme.

Until now, this thesis focused on novel parallelization techniques: Parallel expanded event simulation constitutes a new modeling paradigm while probabilistic synchronization and multi-level parallelization are event execution and synchronization schemes. Given a simulation model, all three techniques aim for extracting maximum performance from the model. However, we have not yet covered the question whether or not the structure of a particular simulation model is beneficial for parallel execution. To answer this question, model developers need to gain a deep understanding of the execution behavior of a simulation model. Hence, the next chapter changes the focus from parallelization techniques to a performance analysis and prediction methodology that provides an insight into the behavior of a parallel expanded event simulation model.

6

Performance Analysis of Parallel Expanded Event Simulations

Our work presented in the previous sections as well as the large body of related efforts [BFBC06, CS05, PVM09] illustrate that parallel simulation can significantly improve simulation runtimes. Hence, the research community is investigating the fundamentals of parallel simulation for over two decades [Fuj90a, Nic96, Per06b] by now. However, parallel simulation is not generally used as evaluation tool in the networking research community on a day-to-day basis. We ascribe this to the fact that developing and handling parallel simulations is noticeably more complex than sequential ones.

Based on this observation, this thesis aims at developing means that foster the widespread use of parallel simulation in the networking research community. In order to reach this goal, we proposed three contributions in the previous sections:

i) Parallel Expanded Event Simulation as an *intuitive* modeling paradigm,
ii) Probabilistic Synchronization as *self-adapting* synchronization scheme, and
iii) Horizon, a simulation framework employing a centralized *partitioning- and loadbalancing-free* parallelization architecture.

These approaches intent to reduce the efforts imposed on model developers and simulation users who want to apply parallel simulation. However, they do not actively support developers in the process of *analyzing and optimizing* the parallel performance of a simulation model or a parallel simulation framework. Hence, this chapter presents a performance analysis and performance prediction methodology. Our methodology provides developers of parallel simulation models and frameworks with an insight into the properties of a parallel simulation in terms of event dependencies and CPU utilization. Analyzing this information allows model developers to identify and eventually eliminate performance bottlenecks.

The remainder of this chapter first outlines the benefits of analyzing the performance of parallel simulations in Section 6.1. We then investigate the challenges of finding an

optimal event-to-CPU mapping in Section 6.2. Based on these challenges, Section 6.3 reviews related efforts targeting performance analysis of parallel simulations. After reviewing the state-of-the art, Section 6.4 presents a formal problem definition of our optimization problem and the resulting MILP. To mitigate the computational complexity of this MILP, Section 6.5 describes a trace splitting scheme, proves its correctness, and outlines relaxations of the MILP which trade off accuracy for scalability. We then evaluate the accuracy and performance of our scheme and its optimizations in Section 6.6. Finally, we discuss limitations of our approach in Section 6.7 and conclude in Section 6.8.

6.1 Motivation

Achieving satisfying parallel performance is a demanding task for both, users and developers of parallel simulation models and frameworks. We claim that disappointing parallel performance results from a lack of insight into the behavior and runtime properties of parallel simulations. Hence, developers and users of parallel simulation need an accurate and simple-to-use means of identifying, understanding, and eliminating performance issues. By developing a performance analysis methodology, we intend to provide such as means and thereby foster the wide-spread use of parallel simulation. In the following, we illustrate three use cases in which the performance analysis of a parallel simulation model proves beneficial.

Model Development Support

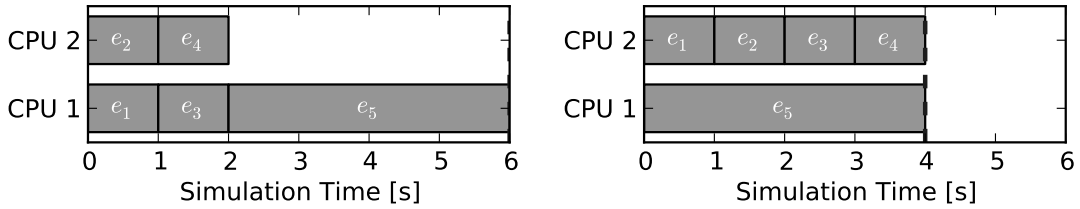
The performance of a simulation model under parallel execution needs to be considered from the very beginning of the model development process. In general, developing a parallel simulation model is considerably more complex than creating an equivalent sequential one. The key contributing factor in this regard is the structure of a simulation model. In order to provide a parallel workload, i. e., independent events, a simulation model must be structured in independent components where such events can take place simultaneously.

This puts a significant burden on developers of parallel simulation models: In addition to accurately and correctly modeling a complex system – a task which is already demanding in itself – developers also have to design a model structure that allows for efficient parallel event execution. The challenge in this process lies in creating a structure that reflects the simulated system, complies with best practices in software engineering, and generates adequate CPU utilization.

Moreover, changing the structure of an existing model that delivers poor performance is tedious and time consuming due to re-validation. As a result, model developers need to analyze the performance of a model as early as possible in the development process in order to avoid or identify and eliminate performance bottlenecks.

Efficient Resource Utilization

Analyzing the performance of a parallel simulation model allows for making efficient use of the available computing resources. Parallel simulation models exhibit a certain



(a) FIFO event schedule of HORIZON. (b) Optimal event schedule.

Figure 6.1 Example illustrating the potential differences in the simulation runtime between HORIZON and an optimal event schedule when executing five independent events. The event scheduler of HORIZON does not consider the complexity of events and assigns events to CPUs in FIFO order, resulting in suboptimal CPU utilization.

degree of parallelism given by the maximum number of events that can be processed in parallel. Hence, the degree of parallelism is a natural limit for the number of CPUs that can effectively speed up a parallel simulation run. Increasing the number of CPUs beyond this threshold only results in the surplus CPUs being idle at any point in time during a simulation run.

Thus, accurate knowledge of the degree of parallelism allows for maximizing the utilization of the available CPUs by assigning only as many CPUs to a simulation run as actually needed. Surplus CPUs can instead be utilized by executing multiple parallel simulations concurrently. This is particularly useful when conducting a large number of independent simulation runs as part of a parameter study.

Synchronization and Event Scheduling Performance

Developers of parallel simulation frameworks such as HORIZON need to analyze the performance of the event synchronization and event scheduling algorithms employed by the framework. Recall that in the context of our work, event *synchronization* algorithms identify independent events for parallel processing while event *scheduling* algorithms assign independent events to CPUs. However, the knowledge of these algorithms is limited to the information available to them at runtime, e. g., the events in the FES, the size of the lookahead, and the current utilization of the CPUs. They hence do not posses a global view on *all* events over the *entire* simulation run, resulting in suboptimal performance.

To obtain a better understanding of this issue, we illustrate a simple example in Figure 6.1. Assume the synchronization algorithm has identified five independent events $e_1, \ldots, e_5$. Assume furthermore that the event scheduling algorithm assigns these events to the two available CPUs simply in the order of their IDs, i. e., first e_1, then e_2 and so on. Figure 6.1(a) shows the resulting CPU utilization which leaves CPU 1 idle for 4 time units. The optimal schedule shown in Figure 6.1(b) instead achieves a better CPU utilization. Considering this result, framework developer need to know if a different assignment strategy, e. g., earliest-end-time-first (see Section 7.3.1), achieves a performance closer to the optimum.

Thus, in order to assess the efficiency of synchronization or event scheduling algorithms, framework developers need to know the *optimal* event schedule under consideration of event inter-dependencies and the number of available CPUs. The

performance difference between actual synchronization and scheduling algorithms and the theoretical optimum indicates the remaining potential for optimization.

General Idea

In this chapter, we present a performance analysis methodology [KTGW11, Ten10] that calculates a lower bound on the simulation runtime for our parallel simulation framework HORIZON [KLG+10]. Given a simulation model implementing parallel expanded event simulation and an arbitrary number of CPUs, the performance analysis methodology finds an optimal event-to-CPU mapping that minimizes the simulation runtime.

This mapping problem is NP-complete [Che90, LK78] and is thus not generally considered by existing performance analysis tools [BT02, JTKG03, Lin92, WSR+92]. We address this complexity issue by modeling the parallelization scheme underlying parallel expanded event simulation as a Mixed Integer Linear Program (MILP). As a result, we leave the actual problem of finding an optimal schedule to the efficient heuristics and algorithms of modern MILP solvers. To further mitigate the complexity problem, we develop a trace splitting scheme. By applying a divide-and-conquer strategy, trace splitting significantly reduces the complexity of the MILP while maintaining the accuracy of the results. Additionally, we introduce relaxations of the MILP that trade accuracy for scalability.

Our methodology is split in three steps: First, a sequential run of the simulation model traces the event execution and records the timestamps and processing times of all events. Second, we feed this trace to the MILP which calculates an optimal event schedule that minimizes the overall simulation runtime. Third, we are able to analyze the runtime behavior of the simulation by visualizing the event schedule.

In summary, we make the following contributions in this chapter:

i) We introduce a methodology for *analyzing and predicting* the runtime performance of a parallel simulation model based on *integer linear programming*.
ii) We present a *trace splitting* scheme that significantly improves the scalability of our methodology while maintaining its accuracy. We furthermore prove the correctness of this optimization.
iii) We discuss relaxations of the MILP which trade accuracy for scalability.

6.2 Problem Analysis

Our goal is to design a performance analysis and prediction tool that provides model developers with an optimal event schedule that minimizes the simulation runtime on a given set of CPUs. To this end, the tool needs to precisely reflect the event execution and synchronization scheme of parallel expanded event simulation in order to compute a *valid* schedule. Specifically, it must take into account event dependencies, the structure of the model, and the available processing resources.

This mapping problem is a special case of the NP-complete *parallel machine scheduling problem* [Che90, LK78]: Given l (identical) machines and n different jobs, each

with processing times p_j, $j \in \{1, \ldots, n\}$, the task is to assign each job to a machine, such that the maximum completion times c_j of the events is minimal:

$$\text{minimize} \max_{j \in \{1 \ldots n\}} c_j$$

Such an assignment is valid if it complies with restrictions on the order of jobs, e. g., job i must precede job j, and ensures that each machine processes only one job at a time.

In our variant of the scheduling problem, the goal is to assign a number of *expanded events* with corresponding *event durations* to a set of CPUs such that the simulation runtime is minimal. Furthermore, we have to ensure a valid event sequence, i. e., only overlapping events may be executed in parallel while non-overlapping events have to be handled in the order of their starting times. Hence, in contrast to the classic parallel machine scheduling problem our optimization problem has to consider two time domains per event:

i) the simulated time which defines dependencies among events and
ii) the simulation time which specifies the utilization of the CPUs.

The solution of this scheduling problem yields an upper bound on the speedup one can expect when executing the given model on l CPUs in comparison to sequential execution. This bound is tighter and more specific to the model than the trivial linear speedup bound which predicts an l-fold speedup when utilizing l CPUs. It is important to note that the resulting minimal runtime is a bound for conservative synchronization strategies only. An optimistic strategy may successfully parallelize events that do not overlap and therefore achieve an even better performance [JR91, SR95].

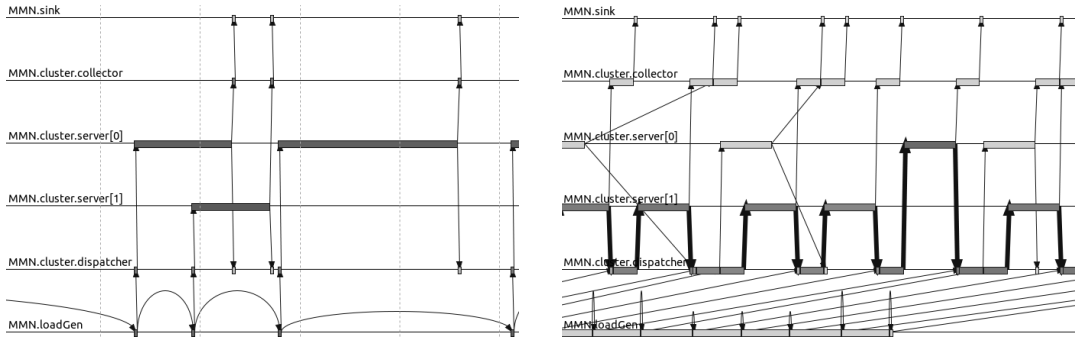
A widely used technique to solve such optimization problems is *integer linear programming*. The major advantage of applying linear programming is that we can express a complex optimization problem in a concise mathematical formulation. By feeding this formulation to an MILP solver, we furthermore leave the problem of efficiently finding a valid solution to a dedicated tool. This is particularly beneficial considering the multitude of algorithms and heuristics for solving MILPs proposed in the literature [Tod02].

6.3 Related Work

This section discusses closely related research efforts regarding performance analysis of parallel simulations.

6.3.1 Critical Path Analysis

Pioneered by Berry et al. [BJ85], *critical path analysis* [SR93, YM89] is among the most widely used performance analysis techniques for parallel simulations. Based on an event trace of a simulation run, it first constructs the dependency graph across all events. In this directed acyclic graph, every event is annotated with its



(a) The extended sequence chart shows the successor relationship and the duration of expanded events.

(b) The bottleneck analysis of the extended sequence chart visualizes the critical path through a simulation run.

Figure 6.2 The extended sequence chart illustrates the runtime behavior of an expanded event simulation and supports model developers in identifying performance bottlenecks.

processing time and connected to another event if a dependency relationship exists among them. Then, in a second step, critical path analysis calculates the path(s) with the largest sum of processing times through the graph. These paths determine the minimum processing time of a parallel simulation run, assuming an unbounded number of processing units. In contrast, our methodology explicitly considers the available processing resources and thus allows predicting the simulation performance for a given set of CPUs.

Wieland et al. [WSR+92] introduce an alternative construction algorithm for the critical path that does not require the construction of a dependency graph. Instead, the critical path is determined recursively based on the notion of the earliest possible completion time (called Earliest Processing Time (EPT)) of an event e which is again given by the EPT of all predecessors of e plus the processing time of e. However, this revised algorithm also does not consider a specific set of processing units, but again relies on an unbounded number of CPUs.

In order to apply critical path analysis to a limited set of CPUs, Lin et al. [Lin92] combine critical path analysis with three selected event scheduling policies. For a given set of virtual CPUs, each policy defines a specific event-to-CPU assignment strategy and allows predicting the performance of a parallel simulation when executed under the selected scheduling policy. Although this approach allows for a much more realistic performance prediction, it relies on online event scheduling algorithms. In general, those algorithms cannot determine the optimal event schedule due to their limited scope at runtime and thus are not able to find the true lower bound on the simulation runtime as achieved by our methodology. As a result, our approach provides a reference against which online event scheduling algorithms can be compared.

Finally, in the context of our research efforts, we developed an extension of the OMNeT++ sequence chart tool based on critical path analysis [KTGW12]. The sequence chart tool [VH08] is a part of OMNeT++'s Integrated Development Environment (IDE) and visualizes the scheduling relations among events using an event trace. We extend the sequence chart to i) consider expanded events (see Figure 6.2(a)), and ii) visualize performance bottlenecks (see Figure 6.2(b)). For

the latter, the tool first computes the critical path through the traced simulation run. Based on the critical path, it determines for each event on the path a "bottleneck factor" that incorporates the number of overlapping events and their processing complexities. The extended sequence chart finally visualizes the bottleneck factor for each event on the critical path by means of a color scheme. However, while our tool provides valuable information about performance bottlenecks in a simulation model, it does not take the CPU utilization into account.

6.3.2 Synchronization Overhead Estimation

In contrast to critical path analysis, OSim [SF87] by Swope et al. utilizes an event trace to optimally synchronize an actual parallel simulation. The goal of OSim is to reconstruct event dependencies from an event trace at runtime and thereby eliminate the need for actual synchronization protocols such as the Null Message Algorithm (NMA) [CM79]. As a result, the parallel simulation blocks solely on data dependencies instead of synchronization related overhead, such as handling of null-messages. Hence, OSim allows to identify the overhead of a particular synchronization protocol.

Bagrodia et al. refine the approach of OSim by defining an efficiency metric for synchronization protocols on top of the Ideal Simulation Protocol (ISP) [BT02]. Based on this metric, the authors analyze the overhead of four selected conservative synchronization protocols. However, since OSim and ISP require to actually execute the simulation in parallel, their performance prediction is restricted to available hardware and hence cannot predict the performance for an arbitrary number of CPUs.

6.3.3 Resource-based Performance Analyzers

Juhasz et al. [JTKG03] present a trace-based performance analyzer for distributed simulations that explicitly takes the characteristics of the simulation hardware into account. In addition to the relative performance of the CPUs, it considers the latency and the topology of the underlying computing network as well as the overhead of selected synchronization protocols. Moreover, the performance analyzer calculates event-to-CPU mappings according to specific assignment policies such as "random", "modulo", and "optimal load-balancing". However, since the analyzer targets partition-based parallel simulations instead of centralized approaches, an event can only be assigned to the one CPU onto which its "parent" logical process was mapped. Consequently, the analyzer can only exploit parallelism across logical processes and not across all events as in a centralized parallelization framework such as Horizon.

Finally, Liu et al. [LNPP99] illustrate an alternative approach for predicting the performance of a parallel simulator. Based on detailed overhead measurements of the core components of the parallel simulator and a selected model, the authors conduct an educated "back-of-the-envelope" estimation of the parallel simulation runtime. Despite the simplicity of the methodology, the authors report surprisingly accurate results. However, their prediction relies on an equally distributed workload

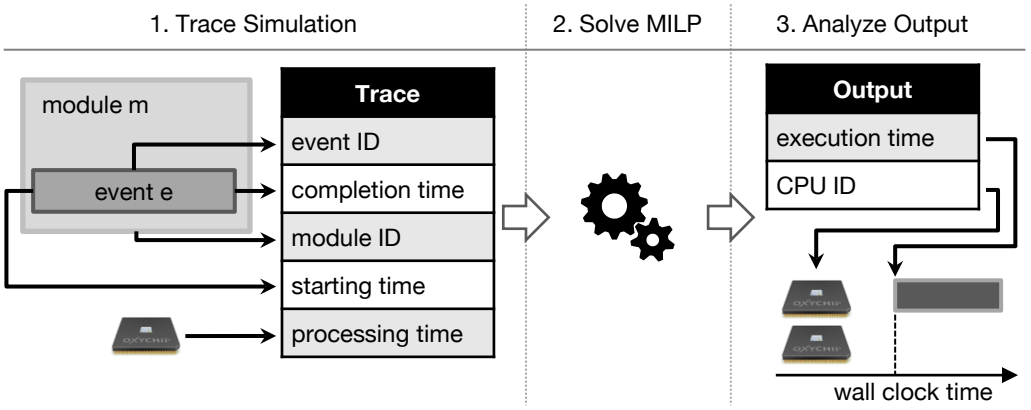


Figure 6.3 Overview of the three steps involved in the performance analysis methodology: 1. tracing of a simulation run, 2. solving the MILP to obtain an optimal event schedule, and 3. analyzing the output which consists of a CPU assignment and wall-clock starting time.

across the CPUs and hence cannot handle load asymmetries which naturally exist in complex simulation models.

6.4 Performance Analysis Methodology

In this section, we introduce the general concept of our performance analysis methodology and present the formal definition of the MILP. Deriving an optimal event schedule is a three-step process as illustrated in Figure 6.3: The first step involves executing a given simulation model sequentially while tracing runtime performance information. In a second step, the MILP takes this information as input and subsequently computes an optimal event schedule. The final third step comprises analyzing the output of the MILP, for instance by visualizing the CPU utilization.

6.4.1 Tracing Simulation Runtime Data

In order to compute an optimal event schedule, the MILP requires detailed knowledge of the runtime behavior of a parallel simulation. To this end, the simulation framework collects the following information for each event during a sequential simulation run:

- the unique *event ID*,
- the *starting time* in simulated time,
- the *completion time* in simulated time,
- the *processing time* in simulation time, and
- the ID of the *module* where the event is executed.

While the purpose of the event ID and the processing time is straightforward, the MILP uses the starting and completion times to determine independent events according to parallel expanded event simulation. Furthermore, to ensure data consistency within the simulation model, Horizon permits only one active worker per

module at a time. Hence, the module ID is required to distinguish truly independent events from those which need to be executed sequentially on the same module.

Due to the centralized architecture of HORIZON, the event scheduler can easily extract this information at runtime and periodically write it to a trace file on disk. Still, special care needs to be taken to prevent undesired side effects of the tracing process on the accuracy of the measurements such as overestimated event processing times due to an increased event handling overhead.

6.4.2 Problem Definition

In this section we give a formal definition of the event scheduling problem. First, we define the input parameters of the MILP and introduce the nomenclature used in the following sections. For completeness and convenience, we also replicate notations already defined in Section 3. We furthermore characterize a valid solution of the scheduling problem and analyze its properties.

Definition 15 (Input to Scheduling Problem)
The *input* of an event scheduling problem is a 6-tuple $(E, C, t_\mathsf{s}, t_\mathsf{c}, t_\mathsf{p}, m)$ with:

- $E = \{e_1 \dots e_n\} \subset \mathbb{N}$ represents the set of *events*.
- $C = \{c_1 \dots c_l\} \subset \mathbb{N}$ represents the set of *CPUs*.
- $t_\mathsf{s} : E \to \mathbb{R}^+,\ e \mapsto$ starting time of e.
- $t_\mathsf{c} : E \to \mathbb{R}^+,\ e \mapsto$ completion time of e.
- $t_\mathsf{p} : E \to \mathbb{R}^+,\ e \mapsto$ event processing time of e.
- $m : E \to \mathbb{N},\ e \mapsto$module ID of e.

We assume without loss of generality that events are ordered with respect to increasing starting times: For $e_1, e_e \in E$ with $e_1 < e_2$ it holds $t_\mathrm{s}(e_1) \leq t_\mathrm{s}(e_2)$. Further, two events $e_1, e_2 \in E$ overlap, denoted by $e_1 \parallel e_2$, if and only if the duration intervals intersect:

$$e_1 \parallel e_2 \Leftrightarrow [t_\mathrm{s}(e_1); t_\mathrm{c}(e_1)] \cap [t_\mathrm{s}(e_2); t_\mathrm{c}(e_2)] \neq \emptyset.$$

Finally, it is important to differentiate between the *duration* $t_\mathrm{d}(e) = t_\mathrm{c}(e) - t_\mathrm{s}(e) \geq 0$ of e in simulated time and the *processing time* $t_\mathrm{p}(e)$ of e in simulation time. The event duration specifies the interval an event spans in the simulation whereas the processing time is the wall-clock time it takes to handle the event on a CPU.

After defining the input of the scheduling problem, we can now specify its output. The solution to a given input of the scheduling problem is a schedule which i) assigns events to CPUs, and ii) specifies a valid order of events on all CPUs under consideration of the dependencies between events. We formally define a schedule as follows:

Definition 16 (Event Schedule)
For an input $(E, C, t_\mathsf{s}, t_\mathsf{c}, t_\mathsf{p}, m)$, a *schedule* $\mathcal{S}$ is a tuple of mappings (x, y) with:

- $x : E \to C,\ e \mapsto c$.
  $x(e)$ assigns event e to CPU c for execution.
- $y : E \to \mathbb{R}^{\geq 0},\ e \mapsto t$.
  $y(e)$ denotes the point t in simulation time (wall clock time) at which the execution of e starts on the CPU assigned by $x(e)$.

The output of the scheduling problem includes the execution starting time $y(e)$ for all events e to model delays in the event execution on a CPU. Such delays occur due to dependencies between events and result in "gaps" in the CPU utilization, i. e., periods in which a CPU is idle and does not process an event (see Figure 6.8 in Section 6.6.2).

Furthermore, we define a feasible schedule as follows:

Definition 17 (Feasible Event Schedule)
For an input $(E, C, t_\mathsf{s}, t_\mathsf{c}, t_\mathsf{p}, m)$, a schedule $\mathcal{S} = (x, y)$ is *feasible* if and only if:

a) For all $d, e \in E$ with $x(d) = x(e)$ or $m(d) = m(e)$:

$$y(d) \geq y(e) + t_\mathsf{p}(e) \text{ or } y(e) \geq y(d) + t_\mathsf{p}(d)$$

"Events mapped to the same CPU and events of the same module are processed sequentially."

b) For all $d, e \in E$ with $t_\mathsf{c}(d) < t_\mathsf{s}(e)$:

$$y(d) + t_\mathsf{p}(d) \leq y(e)$$

"Non-overlapping events are processed sequentially."

A trivial feasible schedule assigns all events to a single CPU for sequential processing according to increasing starting times. Our goal, however, is to find an optimal feasible schedule that minimizes the simulation runtime under consideration of multiple CPUs. Hence, we define an optimal schedule as follows:

Definition 18 (Optimal Event Schedule)
For an input $(E, C, t_\mathsf{s}, t_\mathsf{c}, t_\mathsf{p}, m)$ a schedule $\mathcal{S} = (x, y)$ is *optimal* if and only if:

a) $\mathcal{S}$ is feasible
b) For all feasible schedules $\mathcal{S}' = (x', y')$:

$$R := \max_{e \in E}(y(e) + t_\mathsf{p}(e)) \leq \max_{e' \in E}(y'(e') + t_\mathsf{p}(e'))$$

Definition 18 states that an optimal schedule is feasible and its overall runtime R is less than or equal to the runtime of any other feasible schedule for the given input.

6.4.3 Mixed Integer Linear Program Formulation

We now formulate a MILP that takes an input $(E, C, t_\mathrm{s}, t_\mathrm{c}, t_\mathrm{p}, m)$ and computes an optimal schedule $\mathcal{S}$ for that input. In order to model the schedule $\mathcal{S} = (x, y)$, we define three sets of variables:

- $x_{e,c} \in \{0, 1\},\ e \in E, c \in C$, with $x_{e,c} = 1$ if event e is assigned to CPU c and $x_{e,c} = 0$ otherwise.
- $y_e \in \mathbb{R}^+,\ e \in E$, representing the starting time of the execution of event e in simulation time.

- $z_{d,e} \in \{0,1\},\ d,e \in E, d < e, d \parallel e$, with $z_{d,e} = 1$ if the execution of event d starts before event e in wall clock time and $z_{d,e} = 0$ if the execution of e starts before or equal to d. The purpose of these variables is to model the fact that two independent events may begin execution in any order. To find a solution however, the MILP has to decide on an arbitrary order represented by the value of these variables.

Additionally, we define the variable r which holds the total runtime of the schedule. Furthermore, M is a large positive constant used to conditionally enable or disable constraints depending on the randomly selected assignment of the $z_{d,e}$ variables. Based on these variables, the MILP is defined as follows:

Objective function:

$$\text{minimize } r$$

subject to the following constraints:

$\forall e \in E$:

$$\sum_{c \in C} x_{e,c} = 1 \tag{6.1}$$

$$y_e + t_\mathrm{p}(e) \leq r \tag{6.2}$$

$\forall c \in C$, $\forall d, e \in E$ with $d < e$ and $d \parallel e$ and $m(d) \neq m(e)$:

$$y_e - y_d + (1 - z_{d,e}) \cdot M \geq (x_{d,c} + x_{e,c} - 1) \cdot t_\mathrm{p}(d) \tag{6.3}$$

$$y_d - y_e + z_{d,e} \cdot M \geq (x_{d,c} + x_{e,c} - 1) \cdot t_\mathrm{p}(e) \tag{6.4}$$

$\forall c \in C$, $\forall d, e \in E$ with $d < e$ and $d \parallel e$ and $m(d) = m(e)$:

$$y_e - y_d + (1 - z_{d,e}) \cdot M \geq t_\mathrm{p}(d) \tag{6.5}$$

$$y_d - y_e + z_{d,e} \cdot M \geq t_\mathrm{p}(e) \tag{6.6}$$

$\forall d, e \in E$ with $t_\mathrm{c}(d) < t_\mathrm{s}(e)$:

$$y_d + t_\mathrm{p}(d) \leq y_e \tag{6.7}$$

Constraint 6.1 ensures that each event is assigned to exactly one CPU. Additionally, Constraint 6.2 guarantees that r is an upper bound on the runtime of the schedule. Constraints 6.3 to 6.6 enforce the first condition of a feasible schedule: When two events are mapped to the same CPU, they are executed sequentially with the order depending on the value of the variables $z_{d,e}$ (Constraints 6.3 and 6.4). Furthermore, events on the same module are executed sequentially, again with the order depending on the value of $z_{d,e}$ (Constraints 6.5 and 6.6). Specifically, if $z_{d,e}$ is randomly set to 0 by the MILP solver, the terms including the large constant M in Constraint 6.3 and Constraint 6.5 lift the left part of the constraint to a very large value. As a result, these constraints are trivially true and hence have no impact on the solution of the MILP, i. e., they are disabled. The converse is consequently true for Constraints 6.4

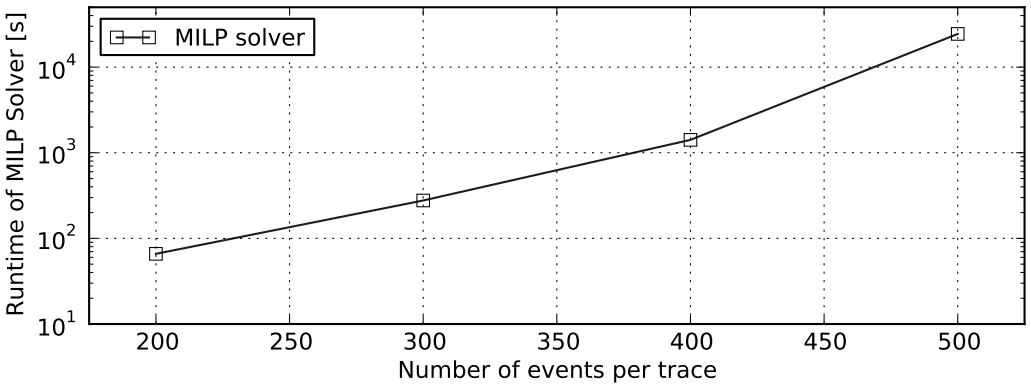


Figure 6.4 The event scheduling problem exhibits an exponential runtime complexity.

and 6.6. Finally, Constraint 6.7 models the second condition for feasibility: Events that do not overlap are executed sequentially.

In combination with the minimization goal of the objective function, the aforementioned constraints enforce the optimality of the schedule. The values of $x_{e,c}$ and y_e in the optimal solution define the optimal schedule $\mathcal{S} = (x, y)$ in the canonic way: $y(e) := y_e$, $x(e) := c$ if $x_{e,c} = 1$, and $R := r$

6.5 Scalability Improvements

We rely on the heuristics and algorithms [Tod02] of modern MILP solvers to compute a (nearly) optimal solution for the MILP. However, since the scheduling problem is NP-complete, solving the MILP typically exhibits exponential complexity in the size of the input, i. e., the length of the event trace. Consequently, the scheduling problem becomes computationally intractable already for short event trances which comprise just a few hundred events.

Figure 6.4 illustrates this fact visually. Without going into details, the figure shows the runtime of the MILP solver for event traces of a simple queueing network model used in our evaluation (see Section 6.6.1). The (super-)linearly increasing curve over a logarithmic scale confirms the exponential runtime complexity of the MILP for the given input, hence rendering the methodology infeasible in practice. In this section, we present approaches to counteract the complexity problem with *and* without sacrificing the accuracy, i. e., optimality, of the results.

6.5.1 Splitting Schedules

The first approach towards increasing the scalability of the performance analysis methodology aims at reducing the input size of the MILP while at the same time retaining the correctness of the resulting schedule. It is based on the observation that the sequence of expanded events generally contains regions of non-overlapping events as indicated by the dashed vertical lines in Figure 6.5. Since parallel expanded event simulation executes only overlapping events in parallel, the event scheduler blocks until all events preceding a non-overlapping region have been processed. Hence, these regions act as natural synchronization points in a parallel simulation run.

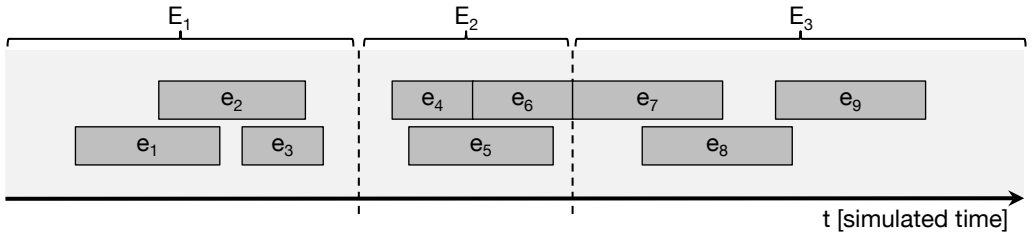


Figure 6.5 The sequence of expanded events in a simulation might contain regions of non-overlapping events which allow for splitting the event trace.

It thus suffices to compute an optimal schedule for the event sequence preceding the synchronization point and the event sequence succeeding it. We exploit this property by dividing the full event trace into a set of significantly smaller sub-traces before feeding those to the MILP. We then iteratively reconstruct a valid schedule for the full trace from the set of sub-schedules. Specifically, the total predicted runtime of the simulation is given by the sum of the runtimes calculated for each sub-trace without loss of accuracy. Due to the smaller input size and the exponential complexity of the scheduling problem, finding solutions for the sum of sub-traces is considerably less complex than for the full event trace.

In the following, we prove that the combined schedule is indeed an optimal schedule for the full scheduling problem. To this end, we first define the notion of a split.

Definition 19 (Split of an Event Trace)
For an input $(E, C, t_\mathsf{s}, t_\mathsf{c}, t_\mathsf{p}, m)$ the tuple (E_1, E_2) is called a *split* of $E = \{e_1 \dots e_n\}$ if the following holds:

a) $E_1 = \{e_1 \dots e_i\}$ and $E_2 = \{e_{i+1} \dots e_n\}$ for some i with $1 \leq i < n$
b) For all $d \in E_1$ and $e \in E_2 : t_\mathsf{c}(d) < t_\mathsf{s}(e)$

Based on this definition, we now formulate the key theorem.

Theorem 2
Given an input $(E, C, t_\mathsf{s}, t_\mathsf{c}, t_\mathsf{p}, m)$, with a split (E_1, E_2) of E and optimal schedules $\mathcal{S}_j = (x_j, y_j)$ on $(E_j, C, t_\mathsf{s}|_{E_j}, t_\mathsf{c}|_{E_j}, t_\mathsf{p}|_{E_j}, m|_{E_j})$, $j \in \{1, 2\}$, then $\mathcal{S} = (x, y)$ with

$$
\begin{aligned}
x(e) &:= x_j(e) \text{ for } e \in E_j \\
y(e) &:= \begin{cases} y_1(e) & \text{for } e \in E_1 \\ \max\limits_{d \in E_1} y_1(d) + t_\mathsf{p}(d) + y_2(e) & \text{for } e \in E_2 \end{cases}
\end{aligned}
$$

is an optimal schedule on $(E, C, t_\mathsf{s}, t_\mathsf{c}, t_\mathsf{p}, m)$.

By iterative application of this theorem, we can compose an optimal schedule for the entire trace from the individual splits. In order to prove the optimality of the combined schedule, we first need to show its feasibility.

Lemma 4
Given an input $(E, C, t_\mathsf{s}, t_\mathsf{c}, t_\mathsf{p}, m)$, with a split (E_1, E_2) of E and feasible schedules $\mathcal{S}_j = (x_j, y_j)$ on $(E_j, C, t_\mathsf{s}|_{E_j}, t_\mathsf{c}|_{E_j}, t_\mathsf{p}|_{E_j}, m|_{E_j})$, $j \in \{1, 2\}$, then $\mathcal{S} = (x, y)$ as defined in Theorem 2 is a feasible schedule for $(E, C, t_\mathsf{s}, t_\mathsf{c}, t_\mathsf{p}, m)$.

Proof. We prove the two conditions of feasibility for the combined schedule $\mathcal{S}$.
Definition 17 a): For $d, e \in E$, let $x(d) = x(e)$ or $m(d) = m(e)$. Since $\mathcal{S}_1$ and $\mathcal{S}_2$ are feasible, the cases $d, e \in E_1$ and $d, e \in E_2$ are fulfilled by definition. Hence, we only have to consider $d \in E_1, e \in E_2$:

$$\begin{aligned} y(d) + t_\mathrm{p}(d) &= y_1(d) + t_\mathrm{p}(d) \\ &\leq \max_{e' \in E_1} y_1(e') + t_\mathrm{p}(e') \\ &\leq \max_{e' \in E_1} y_1(e') + t_\mathrm{p}(e') + y_2(e) \\ &= y(e). \end{aligned} \tag{*}$$

Definition 17 b): Again, we only have to consider $d \in E_1, e \in E_2$ since $\mathcal{S}_1$ and $\mathcal{S}_2$ are feasible. Because (E_1, E_2) is a split of E, it follows $t_\mathrm{c}(d) < t_\mathrm{s}(e)$. Finally, from (*) follows Definition 17 b). □

We can now prove Theorem 2:

Proof. We show that the combined schedule $\mathcal{S}$ fulfills the two conditions of optimality.
Definition 18 a): Follows directly from Lemma 4.
Definition 18 b): The runtime of the combined schedule is

$$\begin{aligned} R &= \max_{e \in E} y(e) + t_\mathrm{p}(e) \\ &\overset{(*)}{=} \max_{e \in E_2} y(e) + t_\mathrm{p}(e) \\ &= \max_{e \in E_2} \max_{d \in E_1} y_1(d) + t_\mathrm{p}(d) + y_2(e) + t_\mathrm{p}(e) \\ &= \left(\max_{d \in E_1} y_1(d) + t_\mathrm{p}(d) \right) + \left(\max_{e \in E_2} y_2(e) + t_\mathrm{p}(e) \right) \\ &= R_1 + R_2 \end{aligned}$$

Proof by contradiction: Assume there exists a feasible schedule $\mathcal{S}' = (x', y')$ with $R' = \max_{e \in E} y'(e) + t_\mathrm{p}(e) < R$. Because $\mathcal{S}'$ has to first compute all events from E_1 before starting to compute events from E_2 (or else $\mathcal{S}'$ would not be feasible (Definition 17 b)), it takes at least R_1 to finish the computation of E_1 since $\mathcal{S}_1$ is optimal. After that, $\mathcal{S}'$ needs at least R_2 to finish the computation of E_2 since $\mathcal{S}_2$ is optimal as well. Therefore, it follows $R' \geq R_1 + R_2 = R$. Contradiction. □

Concluding, splitting an event trace and calculating the total runtime from the resulting sub-problems constitutes a valid means of improving the scalability of our performance analysis scheme. We demonstrate the effectiveness of this optimization in Section 6.6 and discuss limitations in Section 6.7.

6.5.2 Eliminating Events with Insignificant Processing Times

A further approach to reducing the input size of the MILP focuses on eliminating events of very low computational complexity from the event trace. It bases on the observation that the event processing times in a simulation model may span several

orders of magnitude as shown in Figure 3.13 in Section 3.4.4.4. As a result, long running events can completely dominate short running events in terms of the total simulation runtime. In contrast, every event in the input to the MILP increases the complexity of finding a solution, independent of its processing time. We thus conclude that events with very short processing times have only a marginal impact on the overall simulation runtime while at the same time contributing equally to the complexity of the MILP.

By deleting all events with processing times below a threshold from the event trace, the complexity of solving the MILP decreases. At the same time, the error introduced by ignoring events remains within predictable bounds. Specifically, we can derive the maximum error of the computed runtime from the set $D \subset E$ of dropped events, the number of CPUs $|C|$, and the runtime $r_d \in \mathbb{R}^{\geq 0}$ calculated over the remaining events:

Assuming that all dropped events can be equally distributed among all CPUs, the lower bound $\tilde{r}$ for the actual runtime r is

$$\tilde{r} = r_d + \frac{1}{|C|} \cdot \sum_{d \in D} t_\mathrm{p}(d).$$

If instead all dropped events need to be executed sequentially because of mutual dependencies, the maximum error $\epsilon_{\tilde{r}}$ of $\tilde{r}$ is

$$\epsilon_{\tilde{r}} = \frac{|C| - 1}{|C|} \cdot \sum_{d \in D} t_\mathrm{p}(d).$$

6.5.3 Relaxations

In addition to reducing the size of the input to the MILP, we can additionally improve its scalability by relaxing its constraints at the price of less accurate results. In the following, we discuss two different relaxations.

6.5.3.1 Relaxation 1: Overloading CPUs

The first relaxation replaces Constraints 6.3 and 6.4, which enforce that each CPU executes only one event at a time, with a single less strict formulation:

$$\forall c \in C : \sum_{e \in E} (x_{e,c} \cdot t_\mathrm{p}(e)) \leq r$$

> "The sum of the processing times of all events e assigned to CPU c is a lower bound for the runtime r."

In contrast to requiring that a CPU must not be overloaded at any point in time, i. e., not concurrently handling more than one event, the relaxed constraint allows such an overloading of CPUs. This relaxation thus computes runtimes which are too low. However, such overload situations occur only if more independent events than CPUs are available. If, in contrast, the number of independent events is smaller than the number of CPUs, some CPUs remain idle (see Figure 6.8), thereby compensating

for a potential previous overloading. Hence, this relaxation effectively specifies that a CPU must not be overloaded on *average* over the entire runtime r.

This introduces an interesting side effect in combination with the trace splitting scheme. When applying the relaxed MILP to a split trace, the average load assignment is enforced on each sub-trace instead of the entire trace. As a result, r becomes more precise with an increasing number of splits. In terms of complexity, the relaxation defines only one constraint for each CPU instead of one for each pair of overlapping events and each CPU. It thus considerably reduces the size of the integer linear program while still retaining a reasonable lower bound on the overall simulation runtime.

6.5.3.2 Relaxation 2: Disregarding CPUs

The second relaxation removes Constraints 6.1, 6.3 and 6.4 from the MILP. It thus does not consider CPUs anymore, but focuses solely on event dependencies. As a result, the predicted simulation runtime is a lower bound on the simulation runtime given an unbounded number of CPUs. Hence, this relaxation corresponds to a mathematical formulation of the critical path analysis [BJ85].

The results of the relaxed linear program provide information on the maximal degree of parallelism in the model. To this end, we extract from the output of the MILP solver the maximum number of events that execute in parallel, i. e., whose intervals of execution time intersect:

$$p_{\max} := \max\left(|\{e, e' \in E | [y(e); y(e) + t_{\mathrm{p}}(e)] \cap [y(e'); y(e') + t_{\mathrm{p}}(e')] \neq \emptyset\}|\right)$$

From $p_{\max}$, we can derive an upper bound on the number of useful CPUs. In fact, extracting the maximal degree of parallelism from an event trace does not necessarily require solving a integer linear program. Instead, one can easily extract the maximum number of overlapping events from the trace in linear time by traversing the trace sequentially. However, the complexity of the relaxed linear program is small (see Section 6.6.3) and the declarative way of specifying a linear program is convenient.

Concluding, by means of trace splitting, event dropping, and by applying relaxations, the complexity of the MILP reduces significantly – with and without loss of accuracy. We evaluate the exact impact of our scalability optimizations on the accuracy of the results in the next section.

6.6 Evaluation

In the following sections, we evaluate accuracy, complexity, and performance of the runtime analysis scheme and its optimizations. To this end, we first introduce the evaluation methodology before presenting the actual evaluation results.

6.6.1 Methodology

Our evaluation comprises two components: i) An evaluation model, and ii) an implementation of the MILP. We now describe each component in detail.

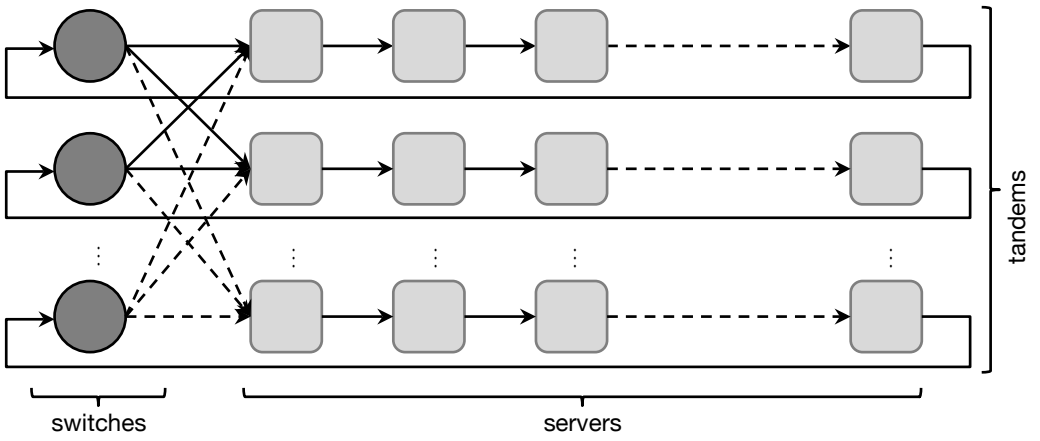


Figure 6.6 Structure of the closed queueing network utilized in the evaluation. The servers continuously process and forward incoming tasks. Each switch distributes incoming tasks randomly to the tandem queues.

6.6.1.1 Evaluation Model

For simplicity and controllability of the workload, we base the evaluation of our performance analysis methodology on a model of a closed queueing network as depicted in Figure 6.6. The network consists of tandem queues, each composed of a chain of servers and a switch. Initially, each server creates a task with a Time To Live (TTL) value and sends it to its neighboring server. Subsequently, the servers continuously process incoming tasks with an exponentially distributed service time, decrement the TTL of the task, and forward the task again. Once the TTL of a task reaches 0, it is discarded. Furthermore, a switch dispatches incoming tasks to one of the tandem queues in a uniformly distributed manner. All the links in the network exhibit a static propagation delay. In order to confront the integer linear program with a wide range of event processing times, handling a task involves a dummy computation of uniformly distributed length. The simulation ends when no tasks remain in the network. Table 6.1 summarizes the exact parameters used in the evaluation.

6.6.1.2 Implementation and Setup

We use an extended implementation of Horizon based on OMNeT++ 3.3 [Var01]. The modifications of Horizon mainly target the central event scheduler and comprise the necessary functionality to create an event trace during sequential execution.

The mixed integer linear program is implemented using Zimpl v3.1.0 (Zuse Institute Mathematical Programming Language) [Koc11]. The Zimpl compiler translates the description of the MILP and a given event trace into a format suitable for the IBM ILOG CPLEX v12.4 MILP solver used in this evaluation. All measurements were conducted on two 6-core AMD Opteron 2431 CPUs using 32GB of RAM and running a 64-bit Ubuntu 9.10.

Parameter	Value(s)
Number of tandems	10
Number of servers per tandem	7
Link delay	1.5 s
Service mean time / event duration	exponential, mean 0.1 s
Event processing time	uniform, 0 - 0.3 s
Time-to-live per task	23

Table 6.1 Parameterization of the closed queueing network.

6.6.2 Accuracy

We assess the prediction accuracy of the MILP and its relaxations by comparing the estimated runtimes against measurements of the actual simulation runtimes. To this end, we first establish ground truth by executing the evaluation model in parallel on different numbers of CPUs, ranging from 2 up to 6. Since we are only interested in analyzing the parallel runtime performance, we omit sequential execution. For each number of CPUs, we then apply the MILP, Relaxation 1, and Relaxation 2 to a previously collected event trace. Furthermore, because of the regular behavior of the simulation model, we do not expect changes in the runtime behavior of the simulation over time. Hence, we restrict the TTL of each task to 23, resulting in a relatively short trace of about 1600 events. For all three versions of the linear program, the MILP solver computes a solution within at least 0.1 % optimality. See Section 6.6.3 for a detailed analysis of the runtime of the MILP solver.

Figure 6.7 compares the measured runtimes of Horizon to the runtimes predicted by the MILP and its relaxations. The figure clearly illustrates that the runtimes of Horizon exceed all predicted runtimes. At first, we focus on the differences between Horizon and the MILP.

6.6.2.1 MILP

The MILP accurately models the parallel expanded event simulation scheme and allows for computing an optimal event schedule that minimizes the simulation runtime. As expected, Horizon does not match the minimal runtime, but requires roughly 8 % longer runtimes. We argue that two factors contribute to this difference in performance: i) The performance prediction methodology assumes an optimal simulation framework without event handling overhead, and ii) Horizon is unable to achieve an optimal event schedule.

Event Handling Overhead: The event trace contains only pure event processing times and disregards the event handling overhead of the event scheduler, i. e., dequeueing an event, checking its overlapping, and offloading it to a worker thread. As a result, the predicted runtime is optimal in the sense that an optimal simulation framework imposes no such event handling overhead. In contrast, since the tracing component measures the execution time of the event

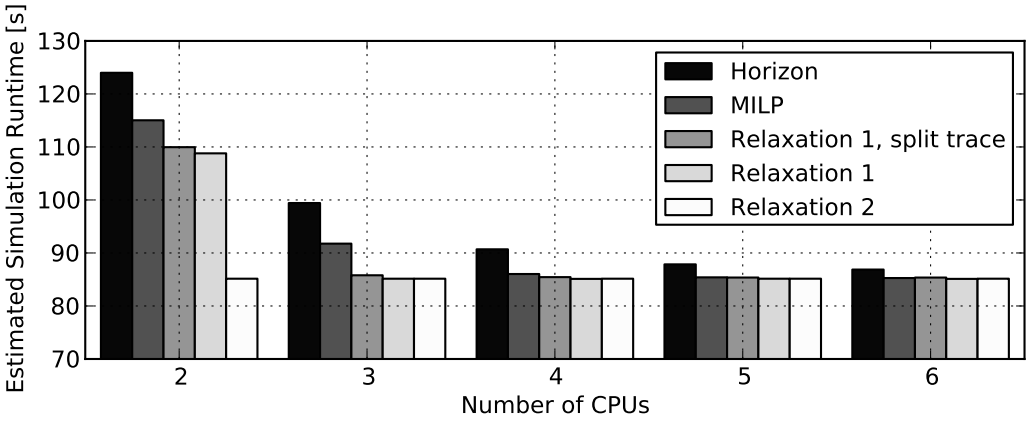


Figure 6.7 Comparison of the predicted simulation runtimes computed by the MILP and its relaxations with the actual runtime of HORIZON. We only focus on parallel runtime performance and hence omit sequential execution.

handlers, the event handling overhead encountered by workers, e. g., creation and deletion of event, enqueueing events in the FES, is included in the measurements. This enables framework developers to asses the performance impact of the core of the simulation framework, i. e., the event scheduler and its synchronization algorithm.

Recalling the evaluation results presented in Section 3.5.5.3, the optimized event handling overhead of HORIZON is about 1.5 µs per event. We deduce from this that the combined event handling overhead for processing 1600 events is at maximum 2-3 ms. This overhead, however, is negligible in comparison to the event processing times which average at 0.15 s and the total runtime of the simulation which ranges between 85 s and 125 s. Thus, we conclude that the event handling overhead of HORIZON certainly contributes to the difference in performance, yet its impact is insignificant.

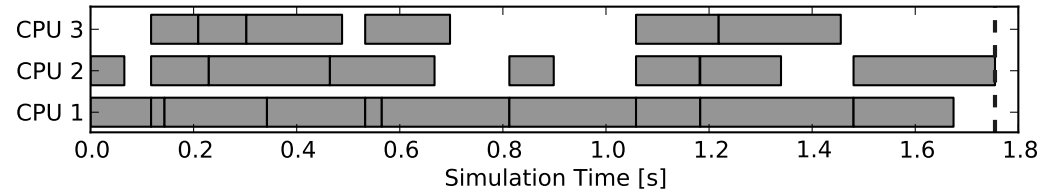
Event Schedule: The second contributing factor to the performance difference between HORIZON and the predicted runtime is that HORIZON is not able to distribute parallel events optimally to CPUs. To back up this claim, we compare the first 24 events of the event schedule generated by HORIZON to the optimal schedule computed by the MILP in Figure 6.8. The figures show that the optimal schedule indeed achieves a better CPU utilization, resulting in a shorter runtime.

We identify three primary reasons for this result:

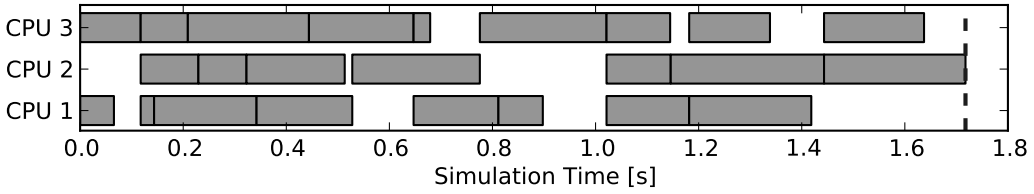
i) First, the event scheduler does not consider the processing times of the events but merely identifies independent events based on their duration. The processing time, however, determines the workload an event imposes on a CPU, and hence the CPU utilization.

ii) In its current implementation, the centralized event scheduler offloads events in First In First Out (FIFO) order. It may instead achieve a better performance by first collecting the set of parallelizable events, i. e., all overlapping events, and then analyzing this sets, e. g., in terms of the processing times, in order to achieve a better CPU utilization.

iii) Even if considering the set of parallelizable events and the corresponding processing times, the event scheduler of HORIZON has to make its



(a) Schedule used by the scheduler of HORIZON.



(b) Optimal schedule.

Figure 6.8 Comparison of the schedule used by HORIZON with an optimal schedule for the first 24 events of the evaluation trace.

decisions at runtime. Hence, it has no global knowledge of the entire simulation run as opposed to the MILP which computes an optimal schedule for the complete event trace. In particular, at any point in time during a simulation run, the event scheduler of HORIZON can only incorporate those events in its scheduling decision which are in the future event set of the simulation. As a result, a runtime event scheduler can only approximate an optimal event schedule.

Motivated by this analysis, a next logical step is to enhance the event scheduler of HORIZON. Hence, we outline a scheduling algorithm that considers the completion time of expanded events in Section 7.3.1 as future work.

6.6.2.2 Relaxations

Revisiting Figure 6.7, we observe that both relaxations compute runtime bounds which deviate noticeably from the optimal schedule. Since these relaxations allow overloading of CPUs (Relaxation 1) or do not at all take CPUs into account (Relaxation 2), the resulting runtime bounds are too low. However, as pointed out in Section 6.5.3, Relaxation 1 increases in accuracy when used in conjunction with the trace splitting scheme. Figure 6.7 indicates that in the evaluation scenario at hand, it indeed achieves better results which are closer to the optimal schedule when applied to split traces.

Finally, because the solution of Relaxation 2 is independent of the number of CPUs, it only provides information on the maximum degree of parallelism in the simulation under investigation. In terms of accuracy it is hence equivalent to the critical path analysis.

6.6.3 Scalability

In the following, we evaluate the performance improvements gained by the trace splitting scheme and the relaxations of the MILP.

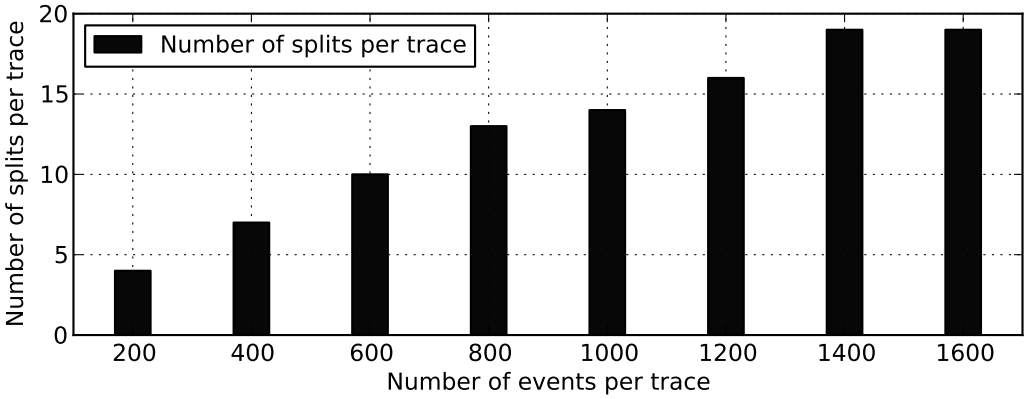


Figure 6.9 Number of splits generated from input traces of specific length. The regular behavior of the queueing network allows for evenly distributed splits.

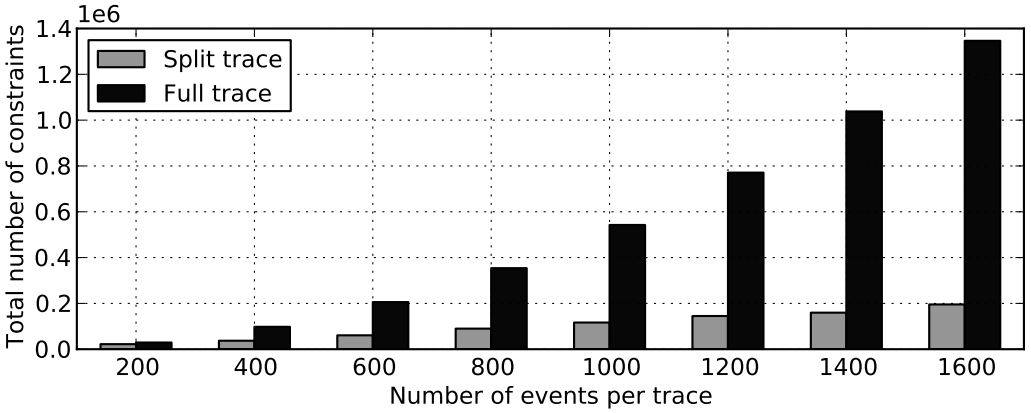
6.6.3.1 Input Complexity

The primary goal of the trace splitting scheme is to reduce the input size of the MILP-solver by dividing the full scheduling problem into a set of sub-problems. We measure the input size of the MILP-solver in terms of the number of constraints and the number of variables.

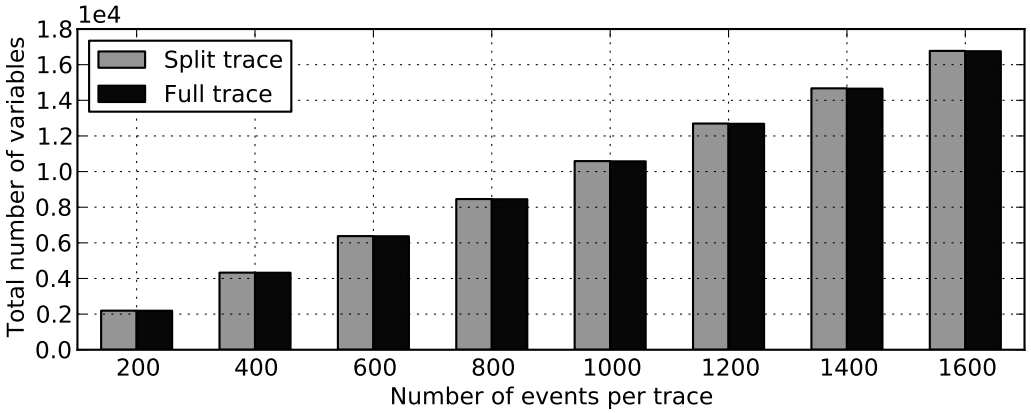
As described in Section 6.6.1.2, we use Zimpl to specify the mixed integer linear program and compile it into an lp-file which CPLEX accepts as input. During the compilation process, Zimpl reads the input trace and generates separate constraints for all events that match a constraint in the Zimpl-specification. For instance, for two overlapping events e and f in the input trace, Zimpl creates a set of specific constraints in the lp-file according to all constraints in the Zimpl-specification that match overlapping events. Hence, the number of constrains in the resulting lp-file is dependent on the input data and is thus a direct measure of the input complexity of CPLEX.

Our evaluation bases on event traces ranging from 200 to 1600 events and a fixed number of 5 CPUs for the scheduling problem. Due to the regular structure of the queueing network model, the traces are evenly splittable into sub-traces as illustrated in Figure 6.9. Please note that in the following, results regarding split traces always refer to the sum over all splits.

Figure 6.10(a) shows the total number of constraints in the resulting lp-files. It clearly illustrates that the number of constraints increases polynomially for full inputs as opposed to a roughly linear growth for split inputs. In general, the number of constraints in the lp-files is polynomial with respect to the input size due to constraints which quantify over all combinations of events and CPUs, such as Constraints 6.3-6.7. However, the small and equal size of the splits prevents this polynomial characteristic from gaining considerable impact per split. As a result, the splitting scheme effectively transforms the polynomial growth into a linear growth over the number of splits. In contrast, Figure 6.10(b) indicates that splitting has no influence on the number of variables in the lp-files. The reason for this behavior is that variables are tied to events and CPUs which remain constant over the sum of all splits (cf. Definition 19).



(a) Number of constraints in the mixed integer linear program after running Zimpl.



(b) Number of variables in the mixed integer linear program after running Zimpl.

Figure 6.10 Size of the mixed integer linear program in terms of the number of constraints and the number of variables after running Zimpl on full and split input traces of varying length. The number of CPUs in the scheduling problem is fixed to 5. Results for split traces are the sum over all sub-traces.

6.6.3.2 Runtime

We analyze the performance improvements of the splitting scheme and the relaxations by investigating the runtime of the performance analysis scheme. Since our implementation builds upon Zimpl and CPLEX, the total runtime of the analysis scheme is the sum of the runtimes of both tools. We measure the runtime of Zimpl by means of the UNIX tool *"time"* while CPLEX itself provides detailed information on the time to solve a given problem.

Figure 6.11 illustrates the runtimes of Zimpl for the MILP, Relaxation 1 (both with and without splitting), and Relaxation 2 for traces ranging from 200 to 1600 events. For full event traces, the runtimes for the mixed integer linear program and its relaxations increase polynomially. This is in line with the results of the previous section which show a polynomial growth in the number of generated constraints. In contrast, the runtimes grow roughly linearly for split inputs due to similarly sized sub-traces.

Furthermore, Figure 6.12 depicts the runtimes of CPLEX. Specifically, as shown in Figure 6.12(a), the runtimes of CPLEX for solving the MILP dominate the time demand of all other schemes. This exponential growth confirms the computational

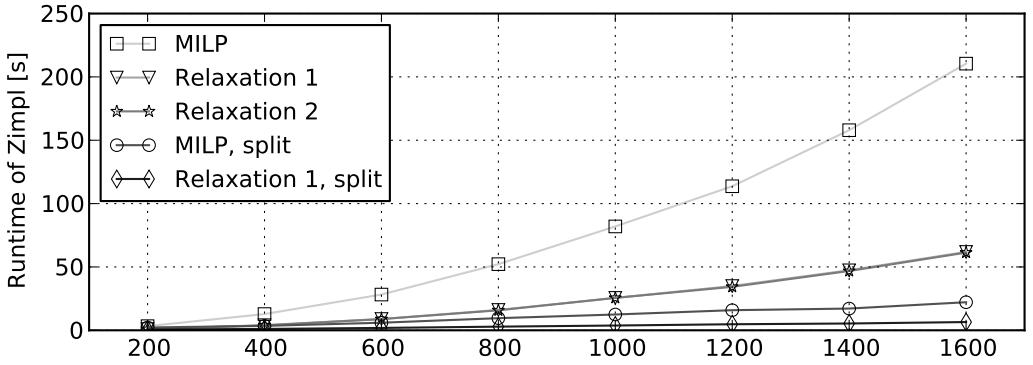
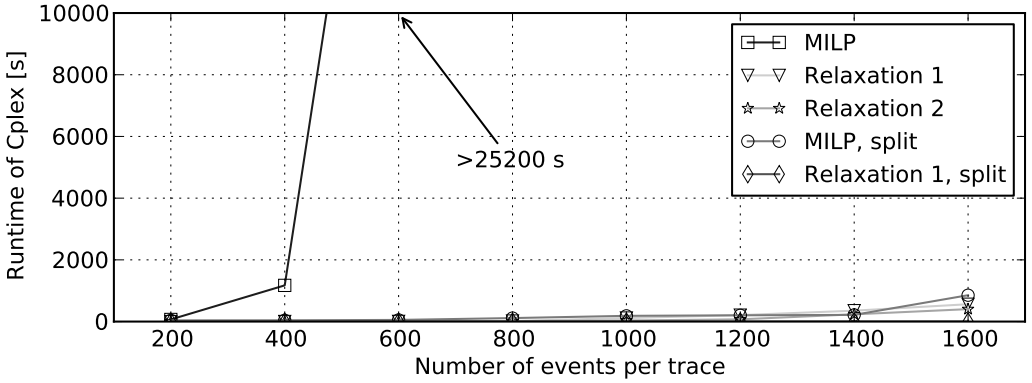
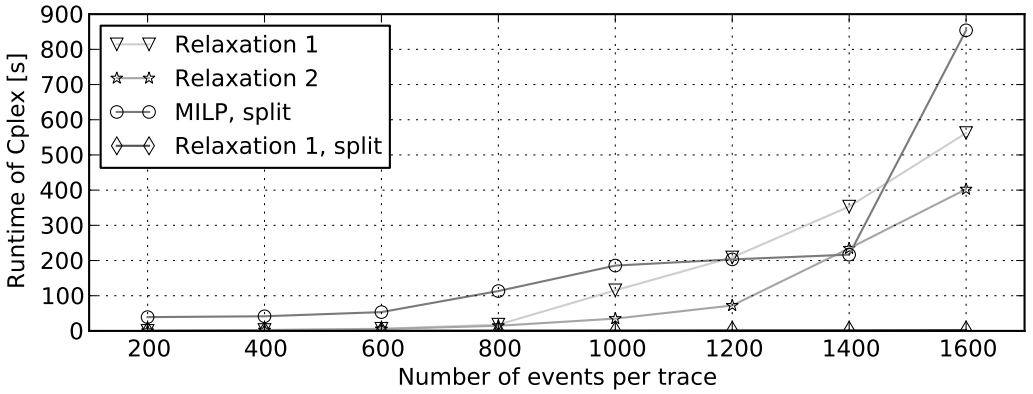


Figure 6.11 Runtimes of Zimpl for traces of ranging from 200 to 1600 events. The MILP and Relaxations 1 and 2 show a polynomial growth in runtime while the splitting scheme reduces the growth to a roughly linear one.



(a) Runtimes of CPLEX including the mixed integer linear program.

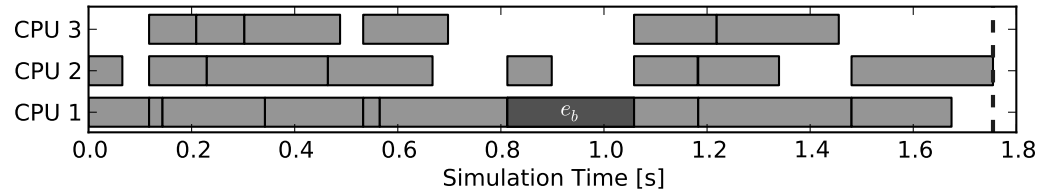


(b) Runtimes of CPLEX without the runtimes of the mixed integer linear program.

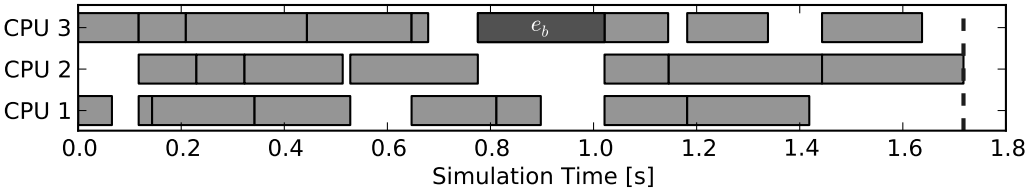
Figure 6.12 Runtimes of the performance analysis schemes for traces of specific length. The runtimes of CPLEX for the mixed integer linear program without splitting are omitted for traces of more than 600 events due to excessive runtimes.

complexity of the scheduling problem. In fact, the scheduling problem becomes computationally intractable for our purposes when the input exceeds 600 events. Hence, we do not present results for the MILP and such traces.

To allow for a more detailed analysis of the remaining schemes, Figure 6.12(b) zooms in on their runtimes. The runtimes of Relaxation 1 and 2 follow a similarly shaped slow polynomial growth. Thus, these relaxed scheduling problems are indeed of significantly reduced complexity in comparison to the strict MILP. In the case of



(a) Schedule used by the scheduler of HORIZON.



(b) Optimal schedule.

Figure 6.13 Comparison of the schedule used by HORIZON with an optimal schedule for the first 24 events of the evaluation trace.

the MILP with split inputs, the graph shows alternating regions of no increase in runtime and regions of considerable increases. We ascribe this to the fact that the runtimes of the splitting scheme depend on the size of the splits. For a sequence of equally sized splits, the runtime demand grows linearly whereas varying sizes introduce fluctuations as visible in the figure: Analyzing Figure 6.9 reveals that no splittable regions exist between event 1400 and 1600. As a result, the size of the last split of the event trace increases. This in turn increases the complexity of finding a solution for the scheduling problem, resulting in a longer runtime of the solver. Moreover, due to the exponential complexity of the scheduling problem, an increase in the input size has a larger impact on the runtime than the polynomial complexity of Zimpl. Thus, the results for Zimpl in Figure 6.11 do not show similar fluctuations. Lastly, we observe in Figure 6.12 that the runtime demand of Relaxation 1 for split traces is barely noticeable at the bottom of the figure. Hence, the combination of the splitting scheme and Relaxation 1 achieves a scalable runtime performance.

6.6.4 Analyzing Event Schedules for Performance Optimization

Finally, this section briefly illustrates how visualizing event schedules enables developers to identify performance bottlenecks. To this end, we again compare the first 24 events of the optimal event schedule with the schedule generated by HORIZON as shown in Figure 6.8. For the reader's convenience, we reproduce the original figure here as Figure 6.13.

We observe that in both schedules the highlighted event e_b blocks progression of the simulation, i. e., all later events depend on e_b according to parallel expanded event simulation. The event scheduler can thus offload further events only after the completion of e_b. As a result, two of the three CPUs remain idle, rendering e_b a severe performance bottleneck. Hence, visualizing event schedules indicates to model developers which events constitute performance bottlenecks.

To eliminate such bottlenecks, model developers need to obtain an understanding of how these events interact with the model. By means of the unique event ID,

developers can identify these events in the trace and determine their occurrence in the model, i. e., the module they take place at. Given this insight, analyzing the event interaction in the context of the model allows for deriving optimizations which eliminate the apparent performance bottleneck. Such optimizations include:

Restructuring Event Dependencies: e_b blocks parallel execution of later events according to parallel expanded event simulation if the corresponding event durations do not overlap. Still, non-overlapping events might still be independent if they do not influence each other as shown in Figure 4.4(b). If this is the case, carefully adapting the durations of the involved events such that they overlap enables parallel execution.

Sub-dividing the Event Workload: The visualized event schedules show that e_b is of non-trivial complexity. Based on this observation, developers may investigate if the event handler contains parallelizable computations, e. g., independent loop iterations. If this is the case, distributing the workload of the formerly monolithic event across multiple parallel events on separate modules allows for creating a better CPU utilization. Examples include splitting up a common channel module into separate modules for each network node or even channel effect such as fading, pathloss, and shadowing.

Whether or not these optimizations are applicable is of course highly dependent on the particular simulation model. Thus, visualizing event schedules is a powerful means of identifying performance bottlenecks. Yet, eliminating these bottlenecks remains a key challenge for model developers, involving thorough analysis and a profound understanding of the model at hand.

6.7 Discussion and Limitations

The evaluation of our performance analysis methodology illustrates that the trace splitting scheme and the relaxations of the MILP achieve a significant performance improvement over the non-relaxed MILP. However, despite these optimizations, only relatively short event traces can be handled by our methodology in a reasonable amount of time. Since a typical simulation run comprises hundreds of thousands to millions of events, it is certainly impractical to predict the runtime of an entire simulation run. Nevertheless, we argue that the presented performance analysis methodology is useful particularly in the early phases of the model development process. Early evaluation scenarios are still of a small scale while at the same time the structure of the model can still easily be changed in order to optimize parallel performance. Furthermore, it is often simply not necessary to trace and analyze the whole simulation run. Instead, one might be interested in analyzing just a specific part of the simulation model or a specific sequence of events, e. g., after extending a model with new functionality. In this case, selectively tracing the model under investigation is sufficient and yields traces of acceptable size.

We showed that the trace splitting scheme is a correct and effective means to reduce the overall runtime of the performance analysis methodology. However, there

is no guarantee that a given event trace is indeed splittable. In fact, we observe an interesting relationship between the efficiency of parallel expanded event simulation and the splittability of a trace: The more events overlap in a simulation model, the more events can be processed in parallel at runtime, resulting in a good speedup. Yet, a large number of overlapping events increases the computational complexity of the MILP and reduces the chance of finding splits of short length. In this situation, however, good simulation performance reduces the need for conducting a performance analysis to find bottlenecks.

Due to the trace-based approach of our performance analysis methodology, the estimated runtimes are tightly coupled to the performance of the underlying hardware. For instance, tracing a simulation run on a desktop machine yields event processing times which are different from the event runtimes on a simulation server. However, we do not expect asymmetric performance differences in terms of the event processing times between different hardware platforms. Instead, the processing times of all events scale by a linear factor on machines of different performance. Hence, the relations between events remain valid and thus also the event schedule calculated by the linear program.

Until now, we assumed that the processing time of an event is given by the time span needed to actually execute an event by a worker thread. Hence, any differences between the calculated lower bound and actual measurements are due to a combination of a potentially non-optimal schedule and the overhead of the event scheduler (cf. Section 6.6.2.1). Despite showing that the impact of the overhead is negligible in the context of events of non-trivial complexity, we can further mitigate its influences. To this end, we assume that the event scheduler imposes a static overhead for de-queueing the next event, checking its overlapping, and assigning it to a worker. Based on this assumption, the tracing component may simply add this approximated overhead to the measured event processing times in the trace file.

An apparent limitation of the trace-based approach is that each trace represents only one simulation run and its parameterization. Consequently, the event schedule computed by our scheme is only valid for one particular set of parameters. Nevertheless, we apply the same argumentation as for the multi-level parallelization scheme (cf. Chapter 5). We hence assume that the order of events in different simulation runs remains similar across varying parameter sets while the main differences are in the local state of the simulation model.

6.8 Conclusions

In this chapter, we presented a performance analysis methodology which provides simulation model developers with an insight into the execution behavior of parallel expanded event simulations. Specifically, our methodology calculates an optimal event schedule and thus the lower bound on the runtime of a given parallel simulation model in our Horizon simulation framework. This insight supports developers in assessing the efficiency of parallel event execution and aids the development of performance optimizations for simulation models, frameworks, and evaluation setups. Thus, by actively supporting the development process of parallel simulations,

we hope to foster the wide spread application of parallel simulation in the network research community.

7

Summary and Conclusions

Discrete event simulation constitutes a fundamental tool for developing, evaluating, and analyzing communication systems. Despite abstracting from irrelevant technical details in simulation, the complexity of a communication system under investigation often translates into *computational* complexity of the corresponding simulation model. To mitigate the excessive simulation runtimes resulting from high computational complexity, the research community developed parallel simulation techniques that distribute the computational workload across multiple processing units. Yet, in this thesis, we identified two major developments that challenge the established state-of-the-art in parallel simulation:

i) Multi-processor computers have become the de facto standard hardware platform for desktop and server systems, providing parallel processing capabilities to researchers. However, the majority of existing parallelization techniques is rooted in distributed simulation on computing clusters whose hardware properties fundamentally differ from multi-processor systems. Thus, existing parallelization techniques do not fully exploit the processing power of multi-processor systems.

ii) The focus of interest in the networking research community shifted from wired to wireless communication systems due to the proliferation of IEEE 802.11 and IEEE 802.14.5. In contrast to wired systems, network nodes in wireless networks are tightly coupled and interact frequently due to the broadcast nature of the wireless channel. This tight coupling in turn constitutes a challenge for parallel event execution.

Consequently, the overarching goal of this thesis is to develop paradigms, algorithms, and tools that enable efficient parallel simulations on multi-processor systems. To achieve this goal, we identify and tackle three distinct research questions:

Q1: How to achieve efficient parallel simulation of tightly coupled systems?
Tightly coupled systems, such as wireless networks, constitute a particular challenge for parallel simulation. Specifically, the tight coupling of components

hinders the identification and safe parallel execution of independent events. As a result, this thesis investigates how to achieve efficient parallel execution of such tightly coupled systems.

Q2: How to exploit multi-processor systems?
Parallel discrete event simulation traditionally focused on distributed computing clusters. In contrast to such clusters, multi-processor systems provide a single globally-shared memory space and fast thread synchronization primitives. Hence, the second research question is how to design a parallelization framework and corresponding synchronization algorithms that exploit the specific properties of multi-processor system in order to achieve i) efficient and ii) simple-to-use parallel event execution.

Q3: How to Support Developers of Parallel Simulations?
Despite over two decades of research, parallel discrete event simulation is still not widely employed in the networking research community. The primary reason hindering general adoption of parallel simulation is the increased effort in developing simulation models that deliver satisfying performance under parallel execution. Consequently, this thesis analyses how to provide support to model developers in order to foster the adoption of parallel simulation.

7.1 Contributions and Achievements

This thesis makes four contributions that address the aforementioned questions. We briefly summarize these four contributions and their key concepts in the following.

7.1.1 Parallel Expanded Event Simulation

Our first contribution is *parallel expanded event simulation*, a novel modeling approach to parallelization. Targeting an efficient parallel execution of tightly coupled system models, its goal is to augment simulation models with event dependency information to aid event synchronization. It is based on the observation that physical processes in a real system take time to complete. Hence, we extend discrete events with a *duration* such that the resulting *expanded events* span a period of simulated time, representing the (processing) duration of a physical process. Building on top of the concept of expanded events, we furthermore define a conservative parallelization scheme stating that overlapping expanded events are safe for parallel processing.

We show the viability of parallel expanded event simulation by implementing a parallel simulation framework, named Horizon. Aiming specifically for ease-of-use, Horizon avoids the need for explicit partitioning and load-balancing by utilizing a simple master-worker architecture. Still, our evaluation shows that Horizon achieves considerable speedups using synthetic and real-world simulation models. In particular, Horizon outperforms traditional distributed parallel discrete event simulation techniques implemented in the state-of-the-art OMNeT++ simulation framework.

This contribution addresses two of the research questions formulated above, namely Q1 and Q2. First, the parallelization scheme defined by parallel expanded event simulation and the master-worker architecture of HORIZON exploit the globally shared memory of multi-processor systems. This enables fast thread synchronization and relieves users from partitioning simulation models and applying load-balancing schemes. Hence, parallel expanded event simulation constitutes an answer to question Q1. Second, the duration of expanded events improve the timing information available in conservative event synchronization. It thus gives an answer to question Q2.

7.1.2 Probabilistic Synchronization

Our second contribution is a probabilistic synchronization scheme that aims at relaxing conservative synchronization while limiting the aggressiveness of optimistic synchronization. To this end, probabilistic synchronization dynamically derives event dependencies at runtime in order to guide parallel event execution. At the core of our contribution are three different *heuristics* which continuously collect event scheduling information, such as successor relations and arrival times, to *learn* event dependencies. Given an event which is not eligible for parallel processing according to conservative synchronization, i. e., beyond the current lookahead, the heuristics estimate if executing this events will result in a causal violation. If the probability for a causal violation is below a user defined threshold, the synchronization scheme speculatively executes the event. Our scheme outperforms both traditional synchronization schemes as well as state-of-the-art approaches in probabilistic synchronization.

Similar to parallel expanded event simulation, probabilistic synchronization provides answers to the first two research questions of this thesis. For one, our implementation of probabilistic synchronization and the three heuristics fundamentally rely on shared memory. Hence, our novel approach to event synchronization demonstrates how to successfully exploit the properties of multi-processor systems, giving an answer to question Q1. Additionally, by determining and analyzing the dependencies between events, probabilistic synchronization is able to efficiently identify independent events in simulation models of tightly coupled systems. Probabilistic synchronization thus provides an answer to question Q2.

7.1.3 Multi-level Parallelism using GPUs

The third contribution of this thesis constitutes a multi-level parallelization scheme that utilizes GPUs to achieve a time and cost efficient parallel execution of large scale parameter studies. The basic idea is to exploit the massively parallel processing power of GPUs to execute the individual simulation runs of a parameter study in parallel. However, the streaming-based processing model of GPUs differs significantly from classic (discrete) event simulation. Hence, our scheme uses two levels of parallelism, *internal* and *external parallelism*, to define an *event aggregation scheme* and a *memory transfer pipeline* in order to map parallel discrete event simulation onto the streaming-based processing model of GPUs. We evaluate our contribution

by means of a prototype implementation. In comparison to a traditional CPU-bound implementation, our scheme achieves considerably better performance at a fraction of the hardware costs.

We consider GPUs to be specialized multi-processor in themselves, particularly due to the fact that they feature a shared memory space across GPU-threads and, though subject to restrictions, also between GPU and CPU memory. As a result, the multi-level parallelism scheme shows how to exploit multi-processor systems for efficient parallel simulation, thereby contributing an answer to question Q1.

7.1.4 Performance Prediction and Analysis

Our fourth and final contribution is a performance analysis and prediction methodology. Its goal is to aid simulation developers in understanding and optimizing the performance of a parallel expanded event simulation. To this end, the methodology computes an optimal event schedule, i. e., assignment of events to CPUs, under consideration of event dependencies and CPU resources by means of a Mixed Integer Linear Program (MILP). By visualizing and analyzing the resulting event schedule, simulation model developers can identify performance bottlenecks, e. g., regions of low parallelism, and attempt to improve performance by restructuring the simulation model.

To mitigate the computational complexity of calculating an optimal event schedule, we moreover develop complexity reduction techniques such as relaxations of the MILP and an input splitting scheme. We demonstrate the accuracy and usability of our performance analysis methodology as well as the effectiveness of our complexity reduction techniques by means of a queueing network model. By aiming at providing support to model developers, our methodology contributes an answer to research question Q4.

7.2 Application of our Work

This thesis was born out of the necessity for reducing the runtime demand of accurate and complex network simulation models in a simple-to-use fashion. Since its first implementation, the Horizon simulation framework has been heavily used on a day-to-day basis to investigate resource allocation and packet scheduling strategies in LTE networks. In the context of these efforts, Horizon was employed by colleagues [PG13] as well as students as part of two master and diploma theses [Gry12, Wey11]. This underlines the real-world applicability of Horizon and shows that we successfully reached our goal of creating a simple-to-use and efficient parallel simulation framework.

In addition, the experience gained in academia during the genesis of this thesis proved valuable in a cooperation project with industry. Analogous to our motivation for this thesis, our industry partner (a leading manufacturer of telecommunication equipment) experienced excessive simulation runtimes while struggling with a reluctant adoption of parallelization techniques by simulation users. We provided development support for an in-house parallel network simulator and supplied a user

support tool similar to our extended sequence chart [KTGW12]. Concluding, the motivation for this project as well as the challenges encountered during its realization closely match the core problems addressed by this thesis, thereby highlighting the relevance of our work.

7.3 Future Directions

Building on top of the status established by our contributions, we identify future directions of this research work in the following and briefly sketch the envisioned approaches.

7.3.1 Earliest-Completion-Time-First Scheduling

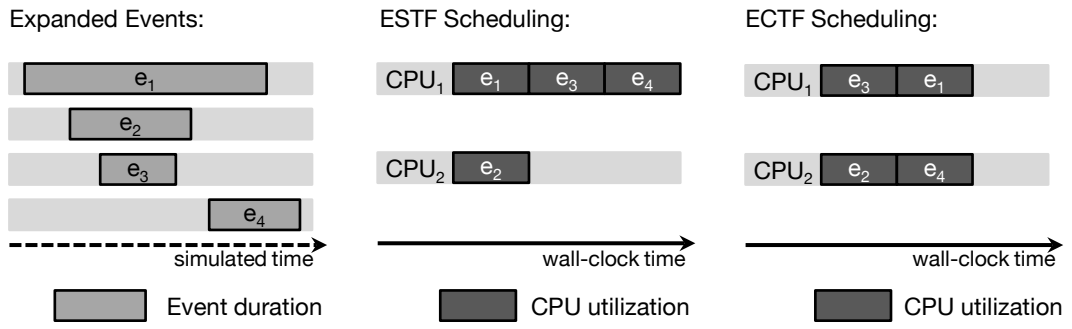
A future effort in the context of this thesis aims at improving the efficiency of the centralized event scheduler. In particular, we envision an alternative scheduling scheme, denoted *Earliest Completion Time First (ECTF) scheduling*, that offloads events such that the barrier t_b advances as soon as possible to maximize the parallelization window.

In the current state of HORIZON, the event scheduler continuously dequeues the event e with the earliest *starting* time from the FES, checks if $t_\mathrm{s}(e) \leq t_\mathrm{b}$ holds true and if so, offloads e to the workers. This strategy guarantees causal correctness, i. e., the central scheduler offloads non-overlapping expanded events in increasing starting time order (cf. Chapter 3). As a result, given a set of independent events, the scheduler offloads the events within this set according to their starting time order as well, denoted as *Earliest Starting Time First (ESTF) scheduling*. In general, however, independent events can be offloaded in any order.

The goal of earliest-completion-time-first scheduling is to offload events such that the barrier t_b advances as soon as possible. By moving the barrier forward, further events might become eligible for parallel processing, thereby limiting the risk of blocking while at the same time improving the resource utilization. The reasoning underlying earliest-completion-time-first scheduling is as follows:

Consider a set of *mutually overlapping* independent events $I \subseteq F$ which have not yet been offloaded, i. e., $I \cap O = \emptyset$. Then, earliest-completion-time-first scheduling offloads the event $e' \in I$ with the smallest completion time, i. e., $e' = \arg\min\{t_\mathrm{c}(e) | e \in I\}$. Recall furthermore that t_b is defined by the minimal completion time of all offloaded events $e \in O$. Hence, offloading e' will likely update the barrier to $t_\mathrm{b} = t_\mathrm{c}(e')$. Yet processing e' first will also very likely result in e' in finishing early, thereby enabling the barrier to move to the next event.

Figure 7.1 illustrates the problem and our proposed solution graphically. Given are two CPUs and a set of expanded events as shown in Figure 7.1(a). In this situation the current scheduling strategy of HORIZON performs considerably worse (see Figure 7.1(b)) than the proposed strategy (see Figure 7.1(c)) in terms of resource utilization.



(a) Expanded events in simulated time.

(b) The scheduler first offloads e_1 and e_2. Afterwards, the non-overlapping events e_3 and e_4 need to execute sequentially.

(c) The scheduler first offloads e_3 and e_2. After their completion it can offload both overlapping events e_1 and e_2 in parallel.

Figure 7.1 The earliest-completion-time-first scheduler selects the event with the smallest completion time. It thereby immediately advances the barrier to maximize the number of parallelizable events and hence the CPU utilization.

The primary challenge of this work is to study and understand the model properties which enable a performance improvement of ECTF over ESTF scheduling. From these properties, we can then derive metrics which aid developers in selecting the best performing strategy for a given simulation model. Moreover, we investigate how to further improve the resource utilization by additionally taking the computational complexity of events into account. Given knowledge of the processing time of an event on a CPU, the event scheduler can approximate the optimal event schedule as computed by our performance analysis methodology.

7.3.2 Automatic Configuration of Probabilistic Synchronization

Future efforts regarding probabilistic synchronization focus primarily on extending the synchronization scheme with automatic configuration and performance tuning capabilities. This is motivated by the observation that our evaluation of probabilistic synchronization shows large differences in performance between i) the individual heuristics and ii) different configurations of the heuristics. As a result, users need to have a profound understanding of the properties of the three heuristics and good knowledge of the behavior of the simulation model in order to find a well performing configuration. These requirements, however, hinder efficient use of our probabilistic synchronization scheme, especially by novice users.

Given a simulation model, the task of the envisioned configuration component is to i) select an appropriate heuristic, and ii) fine-tune the configuration of the selected heuristic. To this end, we propose a two-staged approach consisting of an initial offline selection of a heuristic, followed by a continuous online re-configuration of the selected heuristic at runtime:

Offline Selection of Heuristic: In order to select an appropriate heuristic for the given simulation model, we first need to obtain information about the runtime properties of the model which influence the performance of the heuristics.

Hence, we conduct an initial simulation run to measure model properties such as event complexities, event arrival patterns, or the length of the simulation run. The configuration component then matches the measurements against a previously established decision tree to select a heuristic. It should be sufficient to perform this step only once for each simulation model.

Online Re-configuration: At runtime, the configuration component dynamically adapts the parameterization of the selected heuristic to maximize simulation performance. To determine the impact of these adaptations, the configuration component measures the current simulation performance, e. g., in processed events per seconds, and the accuracy of the heuristics, e. g., the average number of rollbacks per event. Depending on the dynamics of the simulation model it suffices to perform the adaptation only once and re-use the resulting parameterization for all succeeding simulation runs, or it might be necessary to continuously adapt the configuration.

The challenge of this work is to identify accurate performance metrics and understand how changes of the configuration parameters impact simulation performance. Moreover, the adaptation algorithm needs to avoid local maxima, but instead quickly converge to a globally optimal parameterization.

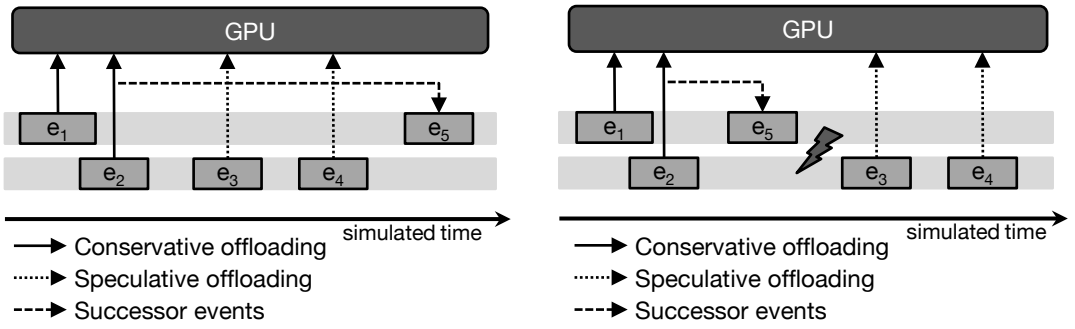
7.3.3 Multi-level Parallelism on GPUs

In the context of our multi-level parallelization scheme, we identify two directions for future work. The first direction envisions a speculative event pre-processing scheme that integrates tightly with internal parallelism and the event transfer pipeline. The second line of work addresses advanced algorithms for mapping events to streaming processors on GPUs.

7.3.3.1 Speculative Pre-processing of Events

We envision a speculative event execution scheme which improves simulation performance by i) pre-processing events which are not yet eligible for parallel processing according to conservative synchronization, and by ii) increasing the resource utilization of the PCIe bus and the GPU. Specifically, we observe that the effects of executing an event are two-fold, consisting of a modified event state and newly generated successor events. However, both effects influence the global simulation state only *after* copying the modified event state and the successor events back from GPU- to CPU-memory.

The anticipated scheme exploits this fact by speculatively dispatching events to the first pipeline stage (“copy state to GPU-memory”) and the second stage (“execute event on GPU”), but skipping the third stage (“copy state to CPU-memory”). Instead, it temporarily stores the modified event states and successor events in GPU memory. If the speculatively executed events do not create a causal violation with respect to conservatively offloaded events, the execution scheme copies the corresponding event states back to CPU memory and enqueues newly generated events in the future event set (see Figure 7.2(a)). If, however, a speculatively executed



(a) No causal violation: The successor event e_5 of a previously offloaded event e_2 succeeds the speculatively offloaded events e_3 and e_4. The scheme can hence commit the pre-computed results of e_3 and e_4.

(b) Causal violation: The successor event e_5 of a previously offloaded event e_2 precedes the speculatively offloaded events e_3 and e_4. The scheme discards the pre-computed results of e_3 and e_4.

Figure 7.2 Speculative pre-processing of events on the GPU.

event triggers a causal violation, the execution scheme simply *discards* all modified event states and successor events (see Figure 7.2(b)).

Since this scheme only commits the results of correctly executed events, i. e., copies them to CPU memory, it is *not* entirely optimistic in nature. Instead, it *pre-computes* events utilizing resources that would otherwise be left unused. Hence, the key challenge of this work is managing the memory resources on the GPU and the utilization of the PCIe bus such that speculative event execution does not interfere with normal event execution.

7.3.3.2 Advanced Event Mapping Algorithms

Lastly, we target the development of advanced event mapping algorithms that mitigate the performance impact of divergent code paths which occur *within* event handlers of the same type. This is necessary since the event mapping algorithms presented in Chapter 5 apply sorting and padding of events only on the basis of different event types. Hence, these algorithms cannot handle divergent code paths which result from conditional branches in the code of events of the same type.

Since the target of conditional branches often depends on the value of local state variables, we intend to develop advanced mapping algorithms that explicitly take the state of an event into account. We anticipate to employ automated source code instrumentation techniques to identify relevant state variables and expose their values to the mapping algorithms. The key challenge of this work is to transparently integrate such techniques in the development and execution process of a simulation model while limiting the runtime overhead.

7.4 Final Remarks

A fundamental concept of our contributions is a centralized event handling and scheduling architecture. In the context of parallelization, this constitutes a rather

unconventional approach as it contradicts the intuitive notion that good parallel performance requires a distributed, i. e., parallel, architecture. However, targeting small and medium scale multi-processor systems, we explicitly favor a centralized architecture for two reasons:

i) Distributed architectures are often too complex for typical simulation users since these architectures require additional efforts such as explicit partitioning or load balancing. Based on our experience from work in academia [Pei12] as well as in industry, simulation users are seldom willing to learn and effectively configure these schemes. Instead, our simple centralized parallelization scheme was well accepted in practice.
ii) A centralized architecture provides a prolific foundation for truly novel approaches to parallelization. We demonstrated this property by developing new heuristics for probabilistic synchronization and a multi-level parallelization scheme.

Taking these two aspects into account, we regard our approach to parallelization successful and expect that it will find further application in practice in the future.

Glossary

ACK	Acknowledgment
API	Application Programming Interface
APP	Accelerated Parallel Processing
CDF	Cumulative Distribution Function
CDMA	Code Division Multiple Access
NUMA	Non-Uniform Memory Access
CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit
CSI	Channel State Information
DES	Discrete Event Simulation
ECTF	Earliest Completion Time First
EIT	Earliest Input Time
eNodeB	Evolved NodeB
EOT	Earliest Output Time
EPT	Earliest Processing Time
ESTF	Earliest Starting Time First
FES	Future Event Set
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPR	False Positive Rate
GPU	Graphics Processing Unit
GSM	Global System for Mobile Communications
GVT	Global Virtual Time
HLA	High Level Architecture
IDE	Integrated Development Environment
ISP	Ideal Simulation Protocol
LBTS	Lower Bound on incoming Time Stamps
LCG	Linear Congruential Generator
LP	Logical Process
LTE	3GPP Long Term Evolution
MANET	Mobile Ad-hoc Network
MILP	Mixed Integer Linear Program
MPI	Message Passing Interface
NMA	Null Message Algorithm
OFDMA	Orthogonal Frequency-Division Multiple Access
OS	Operating System
PCIe	Peripheral Component Interconnect Express
PDES	Parallel Discrete Event Simulation
PDF	Probability Density Function
pdns	Parallel/Distributed ns
PR	Positive Rate
SIFS	Short Interframe Space

SIMT	Single Instruction Multiple Threads
SINR	Signal-to-Interference-plus-Noise Ratio
SISO	Single-Input Single-Output
SSF	Scalable Simulation Framework
TDD	Time Division Duplex
TTI	Transmission Time Interval
TTL	Time To Live
UE	User Equipment
UMTS	Universal Mobile Telecommunications System
UVA	Unified Virtual Addressing
WMN	Wireless Mesh Network
WSN	Wireless Sensor Network

Index

Bibliography

[AKLW10] M. H. Alizai, G. Kunz, O. Landsiedel, and K. Wehrle. Promoting Power to a First Class Metric in Network Simulations. In *Proceedings of the Workshop on Energy Aware Systems and Methods, in conjunction with GI/ITG ARCS 2010*, 2010.

[Amd67] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the Spring Joint Computer Conference*, 1967.

[ASW05] G. W. Allen, P. Swieskowski, and M. Welsh. MoteLab: A Wireless Sensor Network Testbed. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, 2005.

[AWK+11] M. H. Alizai, H. Wirtz, G. Kunz, B. Grap, and K. Wehrle. Efficient Online Estimation of Bursty Wireless Links. In *Proceedings of the 16th IEEE Symposium on Computers and Communications*, 2011.

[BBC+12] P. D. Barnes, J. M. Brase, T. W. Canale, M. M. Damante, M. A. Horsley, D. R. Jefferson, and R. A. Soltz. A Benchmark Model for Parallel ns-3. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, 2012.

[BCF+95] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-second Local Area Network. *Micro*, 15(1):29–36, February 1995.

[BD87] G. E. P. Box and N. R. Draper. *Empirical Model-building and Response Surfaces*. John Wiley & Sons, 1987.

[BEF+00] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Ya Xu, and Haobo Yu. Advances in Network Simulation. *Computer*, 33(5):59–67, May 2000.

[Bel05] F. Bellard. QEMU, a fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, 2005.

[BFBC06] L. Bononi, M. Di Felice, M. Bertini, and E. Croci. Parallel and Distributed Simulation of Wireless Vehicular Ad hoc Networks. In *Proceedings of the 9th International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2006.

[BGMW07] M. Bohge, J. Gross, M. Meyer, and A. Wolisz. Dynamic Resource Allocation in OFDM Systems: An Overview of Cross-Layer Optimization Principles and Techniques. *IEEE Network Magazine, Special Issue: "Evolution toward 4G Wireless Networking"*, 21(1):53 – 59, 2007.

[BGT93] C. Berrou, A. Glavieux, and P. Thitimajshima. Near Shannon limit Error-correcting Coding and Decoding: Turbo-Codes. In *IEEE International Conference on Communications*, 1993.

[BJ85] O. Berry and D. Jefferson. Critical Path Analysis of Distributed Simulation. In *Proceedings of the SCS Conference on Distributed Simulation*, 1985.

[BJK+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[BL94] R. L. Bagrodia and W.-T. Liao. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Transactions on Software Engineering*, 20(4):225–238, 1994.

[BLKW08] A. Becher, O. Landsiedel, G. Kunz, and K. Wehrle. Towards Short-Term Wireless Link Quality Estimation. In *Proceedings of Fifth Workshop on Embedded Networked Sensors*, 2008.

[BMP08] D. W. Bauer, Matthew McMahon, and E. H. P. An Approach for the Effective Utilization of GP-GPUs in Parallel Combined Simulation. In *Proceedings of the 40th Winter Simulation Conference*, 2008.

[Box76] G. E. P. Box. Science and Statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976.

[BRM12] R. Birke, G. Rodriguez, and C. Minkenberg. Towards Massively Parallel Simulations of Massively Parallel High-Performance Computing Systems. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, 2012.

[Bro86] E. D. Brooks. The Butterfly Barrier. *International Journal of Parallel Programming*, 15(4):295–307, August 1986.

[BT02] R. L. Bagrodia and M. Takai. Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages. *IEEE Transactions on Parallel and Distributed Systems*, 11(4):395–411, 2002.

[BTC+98] R. Bagrodia, M. Takai, Y. Chen, X. Zeng, and J. Martin. Parsec: A Parallel Simulation Environment for Complex Systems. *IEEE Computer*, 31(10):77–85, 1998.

[BZvR04] R. Barr, H. Zygmunt, and R. van Renesse. JiST: Embedding Simulation Time Into a Virtual Machine. In *Proceedings of EuroSim Congress on Modelling and Simulation*, 2004.

[CDB09] D. Chatterjee, A. DeOrio, and V. Bertacco. Event-driven Gate-level Simulation with GP-GPUs. In *Proceeding of the 46th ACM/IEEE Design Automation Conference*, 2009.

[Cha99] X. Chang. Network Simulations with OPNET. In *Proceedings of the Winter Simulation Conference*, 1999.

[Che90] T. Cheng. A State-of-the-art Review of Parallel-machine Scheduling Research. *European Journal of Operational Research*, 47(3):271–292, August 1990.

[CK06] M.-K. Chung and C.-M. Kyung. Improving Lookahead in Parallel Multiprocessor Simulation Using Dynamic Execution Path Prediction. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, 2006.

[CM79] K. M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[CNO99] J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling the Global Internet. *Computing in Science & Engineering*, 1(1):42–50, January 1999.

[CON02] J. Cowie, A. Ogielski, and D. M. Nicol. The SSFNet Network Simulator. Software on-line: `http://www.ssfnet.org/homePage.html`, 2002.

[CPF99] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient Optimistic Parallel Simulations using Reverse Computation. *ACM Transactions on Modeling and Computer Simulations*, 9(3):224–253, July 1999.

[CS90] B. Cota and R. Sargent. A Framework for Automatic Lookahead Computation in Conservative Distributed Simulations. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 1990.

[CS05] G. Chen and B. K. Szymanski. DSIM: Scaling Time Warp to 1,033 Processors. In *Proceedings of the 37th Winter Simulation Conference*, 2005.

[DM98] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January 1998.

[Fal99] K. Fall. Network Emulation in the Vint/NS Simulator. In *Proceedings of the 4th IEEE Symposium on Computers and Communications*, 1999.

[FC94] A. Ferscha and G. Chiola. Self-adaptive Logical Processes: The Probabilistic Distributed Simulation Protocol. In *Proceedings of the 27th Annual Simulation Symposium*, 1994.

[Fer95] A. Ferscha. Probabilistic Adaptive Direct Optimism Control in Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, 1995.

[FH98] R. M. Fujimoto and P. Hoare. HLA RTI Performance in High Speed LAN Environments. In *Proceedings of the Fall Simulation Interoperability Workshop*, 1998.

[FPP+03] R. M. Fujimoto, K. S. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley. Large-Scale Network Simulation: How Big? How Fast? In *Proceedings of 11th International IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2003.

[FTG92] R. M. Fujimoto, J. J. Tsai, and G. C. Gopalakrishnan. Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp. *IEEE Transactions on Computers*, 41(1):68–82, January 1992.

[Fuj90a] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[Fuj90b] R. M. Fujimoto. Performance of Time Warp under Synthetic Workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 1990.

[Fuj98] R. M. Fujimoto. Time Management in the High Level Architecture. *Transactions of The Society for Modeling and Simulation International*, 71(6):388–400, 1998.

[Fuj99a] R. M. Fujimoto. Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, 1999.

[Fuj99b] R. M. Fujimoto. Parallel and Distributed Simulation. In *Proceedings of the Winter Simulation Conference*, 1999.

[GG10] J. Gross and M. Güneş. Introduction. In K. Wehrle, M. Günes, and J. Gross, editors, *Modeling and Tools for Network Simulation*, chapter 1, pages 1–11. Springer, Berlin, Germany, April 2010.

[GK06] P. Gepner and M. F. Kowalik. Multi-Core Processors: New Way to Achieve High System Performance. In *Proceedings of the International Symposium on Parallel Computing in Electrical Engineering*, 2006.

[Gry12] M. Grysla. Inter-cell Interference Optimization for Dynamic Scheduling in LTE. Master's Thesis, RWTH Aachen University, 2012.

[GSS+03] D. Gesbert, M. Shafi, D. Shiu, P. J. Smith, and A. Naguib. From Theory to Practice: An Overview of MIMO Space-time Coded Wireless Systems. *IEEE Journal on Selected Areas in Communications*, 21(3):281–302, April 2003.

[HAW08] D. Halperin, T. Anderson, and D. Wetherall. Taking the Sting out of Carrier Sense: Interference Cancellation for Wireless LANs. In *Proceedings of the 14th ACM International Conference on Mobile Computing and Networking*, 2008.

[HBE+01] J. Heidemann, N. Bulusu, J. Elson, C. Intanagonwiwat, K. Chan Lan, Y. Xu, W. Ye, D. Estrin, and R. Govindan. Effects of Detail in Wireless Network Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 2001.

[HF97] M. Hybinette and R. M. Fujimoto. Cloning: A Novel Method for Interactive Parallel Simulation. In *Proceedings of the 29th Winter Simulation Conference*, 1997.

[HF01] M. Hybinette and R. M. Fujimoto. Cloning Parallel Simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):378–407, October 2001.

[HJPM10] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-Accelerated Software Router. In *Proceedings of the ACM SIGCOMM Conference*, 2010.

[HRFR06] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley. ns-3 Project Goals. In *Proceedings of the 2006 Workshop on ns-2: The IP Network Simulator*, 2006.

[iee] IEEE 802.3 Ethernet Working Group. online [last accessed 26 September 2012].

[inf] InfiniBand Trade Association. online [last accessed 26 September 2012].

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis*, volume 182. John Wiley & Sons New York, 1991.

[JB94] V. Jha and R. L. Bagrodia. A Unified Framework for Conservative and Optimistic Distributed Simulation. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, 1994.

[Jef85] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[JM05] S. Jansen and A. McGregor. Simulation with Real World Network Stacks. In *Proceedings of the 37th Winter Simulation Conference*, 2005.

[JR91] D. Jefferson and P. Reiher. Supercritical Speedup. In *Proceedings of the 24th Annual Simulation Symposium*, 1991.

[JTKG03] Z. Juhasz, S. Turner, K. Kuntner, and M. Gerzson. A Performance Analyser and Prediction Tool for Parallel Discrete Event Simulation. *International Journal of Simulation*, 4(1):7–22, May 2003.

[JZT+04] Z. Ji, J. Zhou, M. Takai, J. Martin, and R. L. Bagrodia. Optimizing Parallel Execution of Detailed Wireless Network Simulation. In *In Proceedings of 18th Workshop on Parallel and Distributed Simulation*, 2004.

[JZTB06] Z. Ji, J. Zhou, M. Takai, and R. L. Bagrodia. Improving Scalability of Wireless Network Simulation with Bounded Inaccuracies. *ACM Transactions on Modeling Computer Simulation*, 16(4):329–356, 2006.

[KBV09] M. Kozlovszky, A. Balasko, and A. Varga. Enabling OMNeT++-based Simulations on Grid Systems. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, 2009.

[KDL+05] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, and B. Vanthournout. A Modular Simulation Framework for Spatial and Temporal Task Mapping onto Multi-processor SoC Platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2005.

[Kie05] T. Kiesling. Using Approximation with Time-Parallel Simulation. *Simulation*, 81(4):255–266, April 2005.

[KLG+10] G. Kunz, O. Landsiedel, J. Gross, S. Götz, F. Naghibi, and K. Wehrle. Expanding the Event Horizon in Parallelized Network Simulations. In *Proceedings of the 18th International IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.

[KLGW09] G. Kunz, O. Landsiedel, S. Götz, and K. Wehrle. Poster Abstract: Protocol Factory: Reuse for Network Experimentation, 2009. 6th USENIX Symposium on Networked Systems Design and Implementation.

[KLW09] G. Kunz, O. Landsiedel, and K. Wehrle. Poster Abstract: Horizon - Exploiting Timing Information for Parallel Network Simulation. In *Proceedings of the 17th International IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2009.

[KLW10] G. Kunz, O. Landsiedel, and G. Wittenburg. From Simulations to Deployments. In K. Wehrle, M. Günes, and J. Gross, editors, *Modeling and Tools for Network Simulation*, chapter 6, pages 83–97. Springer, Berlin, Germany, April 2010.

[Koc11] T. Koch. Zimpl User Guide, 2011.

[KRT02] I. Koffman, V. Roman, and R. Technol. Broadband Wireless Access Solutions based on OFDM Access in IEEE 802.16. *IEEE Communications Magazine*, 40(4):96–103, 2002.

[KSGW11] G. Kunz, M. Stoffers, J. Gross, and K. Wehrle. Runtime Efficient Event Scheduling in Muti-threaded Network Simulation. In *Proceedings of the 4th International Workshop on OMNeT++*, 2011.

[KSGW12a] G. Kunz, D. Schemmel, J. Gross, and K. Wehrle. Multi-level Parallelism for Time- and Cost-efficient Parallel Discrete Event Simulation on GPUs. In *Proceedings of the 26th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, 2012.

[KSGW12b] G. Kunz, M. Stoffers, J. Gross, and K. Wehrle. Know Thy Simulation Model: Analyzing Event Interactions for Probabilistic Synchronization in Parallel Simulations. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, 2012.

[KSW+08] A. Köpke, M. Swigulski, K. Wessel, D. Willkomm, PT Haneveld, TEV Parker, OW Visser, HS Lichte, and S. Valentin. Simulating Wireless and Mobile Networks in OMNeT++ – The MiXiM Vision. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, 2008.

[KTGW11] G. Kunz, S. Tenbusch, J. Gross, and K. Wehrle. Predicting Runtime Performance Bounds of Expanded Parallel Discrete Event Simulations. In *Proceedings of the 19th International IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2011.

[KTGW12] G. Kunz, S. Tenbusch, J. Gross, and K. Wehrle. Poster Abstract: Extending the OMNeT++ Sequence Chart for Supporting Parallel Simulations in Horizon, 2012.

[Kun10] G. Kunz. Parallel Discrete Event Simulation. In K. Wehrle, M. Günes, and J. Gross, editors, *Modeling and Tools for Network Simulation*, chapter 8, pages 121–131. Springer, Berlin, Germany, April 2010.

[LA96] M. Liljenstam and R. Ayani. A Model for Parallel Simulation of Mobile Telecommunication Systems. In *Proceedings of 4th International IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1996.

[LA97] M. Liljenstam and R. Ayani. Partitioning PCS for Parallel Simulation. In *Proceedings 5th International IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1997.

[Law96] K. P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996(29):7, 1996.

[LAW08] O. Landsiedel, H. Alizai, and K. Wehrle. When Timing Matters: Enabling Time Accurate and Scalable Simulation of Sensor Network Applications. In *Proceedings of the 2008 International Conference on Information Processing in Sensor Networks*, 2008.

[LF00] M. Loper and R. M. Fujimoto. Pre-sampling as an Approach for Exploiting Temporal Uncertainty. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, 2000.

[LF04] M. Loper and R. M. Fujimoto. A Case Study in Exploiting Temporal Uncertainty in Parallel Simulations. In *Proceedings of the 2004 International Conference on Parallel Processing*, 2004.

[LGW06] O. Landsiedel, S. Gotz, and K. Wehrle. Towards Scalable Mobility in Distributed Hash Tables. In *Proceedings of the International Conference on Peer-to-Peer Computing*, 2006.

[Lin92] Y. B. Lin. Parallelism Analyzers for Parallel Discrete Event Simulation. *Transactions on Modeling and Computer Simulation*, 2(3):239–264, July 1992.

[Liu09] J. Liu. *Parallel Discrete-Event Simulation*. John Wiley & Sons, 2009.

[LK78] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of Scheduling under Precedence Constraints. *Operations Research*, 26(1):22–35, 1978.

[LKGW09] O. Landsiedel, G. Kunz, S. Götz, and K. Wehrle. A Virtual Platform for Network Experimentation. In *Proceedings of the 1st ACM SIGCOMM Workshop on Virtualized Infastructure Systems and Architectures*, 2009.

[LL91] Y. B. Lin and E. D. Lazowska. A time-division Algorithm for Parallel Simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 1(1):73–83, January 1991.

[LLH09] J. Liu, Y. Li, and Y. He. A Large-scale Real-time Network Simulation Study Using PRIME. In *Proceedings of the 2009 Winter Simulation Conference*, 2009.

[LN01] J. Liu and D. Nicol. Learning Not to Share. In *Proceedings 15th Workshop on Parallel and Distributed Simulation*, 2001.

[LN02] J. Liu and D. M. Nicol. Lookahead Revisited in Wireless Network Simulations. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, 2002.

[LNPP99] J. Liu, D. M. Nicol, B. J. Premore, and A. L. Poplawski. Performance Prediction of a Parallel Simulator. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, 1999.

[LNT01] J. Liu, D. M. Nicol, and K. Tan. Lock-free Scheduling of Logical Processes in Parallel Simulation. In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation*, 2001.

[LR09] E. W. Lynch and G. F. Riley. Hardware Supported Time Synchronization in Multi-core Architectures. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, 2009.

[LTE06] 3rd Generation Partnership Project. Technical Specification Group Radio Access Network. *Physical layer aspects for evolved Universal Terrestrial Radio Access (UTRA) (Release 7), 3GPP TR 25.814 V7.1.0*, 2006.

[Lub88] B. D. Lubachevsky. Efficient Distributed Event Driven Simulations of Multiple-loop Networks. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1988.

[Mat93] F. Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. In *Journal of Parallel and Distributed Computing*, 1993.

[MB98] R. A. Meyer and R. L. Bagrodia. Improving Lookahead in Parallel Wireless Network Simulation. In *Proceedings of the 6th International IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1998.

[MB99] R. A. Meyer and R. L. Bagrodia. Path Lookahead: A Data Flow View of PDES Models. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, 1999.

[MCE+02] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, 2002.

[MCM11] M. Moeng, Sangyeun Cho, and R. Melhem. Scalable Multi-cache Simulation Using GPUs. In *Proceedings of the 19th International IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2011.

[MF99] S. McCanne and S. Floyd. *UCB/LBNL/VINT Network Simulator - ns (version 2)*, 1999.

[mob] Mobility Framework. `http://mobility-fw.sourceforge.net/`.

[Moo65] G. E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8):82–85, 1965.

[MSMO97] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *SIGCOMM Computer Communication Review*, 27(3):67–82, July 1997.

[MTK06] A. Markopoulou, F. Tobagi, and M. Karam. Loss and Delay Measurements of Internet Backbones. *Computer Communications*, 29(10):1590–1604, June 2006.

[MVB10] S. De Munck, K. Vanmechelen, and J. Broeckhove. Design and Performance Evaluation of a Conservative Parallel Discrete Event Core for GES. In *Proceedings of the 3rd International Conference on Simulation Tools and Techniques*, 2010.

[NG10] F. Naghibi and J. Gross. How Bad is Interference in IEEE 802.16e Systems? In *Proceedings of the 16th European Wireless Conference*, 2010.

[Nic96] D. M. Nicol. Principles of Conservative Parallel Simulation. In *Proceedings of the 28th Winter Simulation Conference*, 1996.

[Nic03] D. M. Nicol. Darpa Network Modeling and Simulation (NMS) Baseline Network Topology. online, [last accessed 3rd June 2012], 2003.

[NL02] D. M. Nicol and J. Liu. Composite Synchronization in Parallel Discrete-Event Simulation. *IEEE Transactions on Parallel Distributed Systems*, 13(5):433–446, May 2002.

[NPJS10] M. Nanjundappa, H. D. Patel, B. A. Jose, and S. K. Shukla. SCGPSim: A fast SystemC Simulator on GPUs. In *Proceedings of the 15th Asia and South Pacific Design Automation Conference*, 2010.

[NVI] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Whitepaper, V1.1.

[OLG+07] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[omna] *OMNeT++ 4.2 Manual*. online [last accessed 26 September 2012].

[omnb] OMNeT++ Website. online [last accessed 26 September 2012].

[opn] OPNET Modeler, OPNET Technologies, Inc., Bethesda, Maryland, USA. online [last accessed 26 September 2012].

[PAYS09] K. S. Perumalla, B. G. Aaby, S. B. Yoginath, and S. K. Seal. GPU-based Real-Time Execution of Vehicular Mobility Models in Large-Scale Road Network Scenarios. In *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation*, 2009.

[pdn] PDNS - Parallel/Distributed NS. `http://www.cc.gatech.edu/computing/compass/pdns/`.

[PEG11] O. Puñal, H. Escudero, and J. Gross. Performance Comparison of Loading Algorithms for 80 MHz IEEE 802.11 WLANs. In *Proceedings of the 73rd IEEE Vehicular Technology Conference*, 2011.

[Pei12] M. Peiter. Load Balancing in Parallel Discrete Event Simulations on Multiprocessor Systems. Bachelor's Thesis, RWTH Aachen University, 2012.

[Per05] K. S. Perumalla. µsik – A Micro-Kernel for Parallel/Distributed Simulation Systems. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, 2005.

[Per06a] K. S. Perumalla. Discrete-event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs). In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, 2006.

[Per06b] K. S. Perumalla. Parallel and Distributed Simulation: Traditional Techniques and Recent Advances. In *Proceedings of the 38th Winter Simulation Conference*, 2006.

[PF10] H. Park and P. A. Fishwick. A GPU-Based Application Framework Supporting Fast Discrete-Event Simulation. *Simulation*, 86(10):613–628, October 2010.

[PF11] H. Park and P. A. Fishwick. An Analysis of Queuing Network Simulation using GPU-based Hardware Acceleration. *ACM Transactions on Modeling and Computer Simulation*, 21(3):18:1–18:22, February 2011.

[PFTK98] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. *SIGCOMM Computer Communication Review*, 28(4):303–314, October 1998.

[PG13] D. Parruca and J. Gross. Rate Selection Analysis under Semi-Persistent Scheduling in LTE Networks. In *Proceedings of the International Conference on Computing, Networking and Communications*, 2013.

[PGM08] P. Peschlow, M. Geuer, and P. Martini. Logical Process Based Sequential Simulation Cloning. In *Proceedings of the 41st Annual Simulation Symposium*, 2008.

[PHM07] P. Peschlow, T. Honecker, and P. Martini. A Flexible Dynamic Partitioning Algorithm for Optimistic Distributed Simulation. In *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, 2007.

[PM07] P. Peschlow and P. Martini. Efficient Analysis of Simultaneous Events in Distributed Simulation. In *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, 2007.

[PML08] P. Peschlow, P. Martini, and J. Liu. Interval Branching. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, 2008.

[PPFR03] K. S. Perumalla, A. Park, R. M. Fujimoto, and G. F. Riley. Scalable RTI-Based Parallel Simulation of Networks. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, 2003.

[PR11] J. Pelkey and G. F. Riley. Distributed Simulation with MPI in ns-3. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, 2011.

[PVM09] P. Peschlow, A. Voss, and P. Martini. Good News for Parallel Wireless Network Simulations. In *Proceedings of the 12th International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2009.

[PVPS09] T. Preis, P. Virnau, W. Paul, and J. J. Schneider. GPU Accelerated Monte Carlo Simulation of the 2D and 3D Ising Model. *Journal of Computational Physics*, 228(12):4468–4477, July 2009.

[PVQ09] A. Pellegrini, R. Vitali, and F. Quaglia. Di-DyMeLoR: Logging only Dirty Chunks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, 2009.

[PVQ11] A. Pellegrini, R. Vitali, and F. Quaglia. The ROme OpTimistic Simulator: Core Internals and Programming Model. In *Proceedings of the 4th International ICST Conference of Simualtion Tools and Techniques*, 2011.

[QS03] F. Quaglia and A. Santoro. Nonblocking Checkpointing for Optimistic Parallel Simulation: Description and an Implementation. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):593–610, 2003.

[qua] Qualnet Simulator, SCALABLE Network Technologies, Inc., Los Angeles, California, USA. online [last accessed 26 September 2012].

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

[RAT93] H. Rajaei, R. Ayani, and L.-E. Thorelli. The Local Time Warp Approach to Parallel Simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, 1993.

[RBF08] M. Ramadas, S. Burleigh, and S. Farrell. Licklider Transmission Protocol - Specification. RFC 5326 (Proposed Standard), September 2008.

[Rei07] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[RFA99] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. A Generic Framework for Parallelization of Network Simulations. In *Proceedings of the 7th International IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1999.

[Ril03] G. F. Riley. The Georgia Tech Network Simulator. In *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, 2003.

[SB07] K. L. Scott and S. Burleigh. Bundle Protocol Specification. RFC 5050 (Proposed Standard), November 2007.

[SBW88] L. M. Sokol, D. P. Briscoe, and A. P. Wieland. MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution. In *SCS Multiconference on Distributed Simulation*, 1988.

[Sch11] D. Schemmel. Exploiting Multi-Level Parallelism in Discrete Event Simulations using General Purpose Programming Techniques on Graphics Processing Units. Bachelor's Thesis, RWTH Aachen University, 2011.

[Sch12] S. Schöppel. Predicting the Resource Usage of Web Applications. Diploma Thesis, RWTH Aachen University, 2012.

[Seg09] G. Seguin. Multi-core Parallelism for ns-3 Simulator. Technical report, INRIA Sophia-Antipolis, 2009.

[SF87] S. M. Swope and R. M. Fujimoto. Optimal Performance of Distributed Simulation Programs. In *Proceedings of the 19th Winter Simulation Conference*, 1987.

[SHC+04] V. Shnayder, M. Hempstead, B. Chen, G. Werner Allen, and M. Welsh. Simulating the Power Consumption of Large-scale Sensor Network Applications. In *Proceedings of the 2nd Internal Conference on Embedded Networked Sensor Systems*, 2004.

[SL05] H. Sutter and J. Larus. Software and the Concurrency Revolution. *Queue – Multiprocessors*, 3(7):54–62, September 2005.

[SMD+10] A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh. Parallelism via Multithreaded and Multicore CPUs. *Computer*, 43(3):24–32, March 2010.

[SPBP06] N. Spring, L. Peterson, A. Bavier, and V. Pai. Using PlanetLab for Network Research: Myths, Realities, and Best Practices. *ACM SIGOPS Operating Systems Review*, 40(1):17–24, January 2006.

[SR93] S. Srinivasan and P. F. Reynolds. On Critical Path Analysis of Parallel Discrete Event Simulations, May 1993. Technical Report No. CS-93-29, University of Virginia.

[SR95] S. Srinivasan and P. F. Reynolds. Super-criticality Revisited. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, 1995.

[SRTR09] R. Seggelmann, I. Rüngeler, M. Tüxen, and E. P. Rathgeb. Parallelizing OMNeT++ Simulations using Xgrid. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, 2009.

[SS98] T. K. Som and R. G. Sargent. A Probabilistic Event Scheduling Policy for Optimistic Parallel Discrete Event Simulation. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, 1998.

[Ste93] J. S. Steinman. Breathing Time Warp. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, 1993.

[Sto11] M. Stoffers. Heuristic Based Synchronization in Parallel Discrete Event Simulations using Automatic Dependency Analysis. Master's Thesis, RWTH Aachen University, 2011.

[Sut05] H. Sutter. The Free Lunch is Over: A Fundamental Turn toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[SVE03] A. Sekercioglu, A. Varga, and G. Egan. Parallel Simulation Made Easy with OMNeT++. In *Proceedings of the European Simulation Symposium*, 2003.

[SW05] R. Steinmetz and K. Wehrle. *Peer-to-Peer Systems and Applications*, volume 3485. Springer, 2005.

[SWM91] L. M. Sokol, J. B. Weissman, and P. A. Mutchler. MTW: An Empirical Performance Study. In *Proceedings of the 23rd Winter Simulation Conference*, 1991.

[Ten10] S. Tenbusch. Load Scheduling and Performance Estimation in Parallel Network Simulation. Bachelor's Thesis, RWTH Aachen University, 2010.

[TLP05] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, 2005.

[TMB01] M. Takai, J. Martin, and R. L. Bagrodia. Effects of Wireless Physical Layer Modeling in Mobile Ad-hoc Networks. In *Proceedings of the 2nd ACM Internal Symposium on Mobile Ad-hoc Networking and Computing*, 2001.

[Tod02] M. J. Todd. The Many Facets of Linear Programming. *Mathematical Programming*, 91(3):417–436, February 2002.

[TPF+05] Y. Tang, K. S. Perumalla, R. M. Fujimoto, H. Karimabadi, J. Driscoll, and Y. Omelchenko. Optimistic Parallel Discrete Event Simulations of Physical Systems Using Reverse Computation. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, 2005.

[TQ08] R. Toccaceli and F. Quaglia. DyMeLoR: Dynamic Memory Logger and Restorer Library for Optimistic Simulation Objects with Generic Memory Layout. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, 2008.

[TX92] S. Turner and M. Xu. Performance Evaluation of the Bounded Time Warp Algorithm. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, 1992.

[Var01] A. Varga. The OMNeT++ Discrete Event Simulation System. In *Proceedings of the 15th European Simulation Multiconference*, 2001.

[VH08] A. Varga and R. Hornig. An Overview of the OMNeT++ Simulation Environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, 2008.

[VPQ10] R. Vitali, A. Pellegrini, and F. Quaglia. Autonomic Log/Restore for Advanced Optimistic Simulation Systems. In *Proceedings of the 18th International IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.

[VPQ12] Roberto V., A. Pellegrini, and F. Quaglia. Towards Symmetric Multithreaded Optimistic Simulation Kernels. In *Proceedings of the 26th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, 2012.

[VSE03] A. Varga, Y. A. Sekercioglu, and G. K. Egan. A Practical Efficiency Criterion for the Null Message Algorithm. In *Proceedings of the European Simulation Symposium*, pages 26–29, 2003.

[Wey11] M. Weyres. Clustering of Mobile Stations for Interference Coordination in Dynamic OFDMA Systems. Diploma Thesis, RWTH Aachen University, 2011.

[WGG10] K. Wehrle, M. Günes, and J. Gross, editors. *Modeling and Tools for Network Simulation*. Springer Verlag, Berlin, Germany, April 2010.

[WGLW12] E. Weingaertner, R. Glebke, M. Lang, and K. Wehrle. Building a modular BitTorrent model for ns-3. In *Proceedings of the 2012 workshop on ns-3*, 2012.

[WSR+92] F. Wieland, T. Som, P. Reiher, J. Wedel, and D. Jefferson. A Critical Path tool for Parallel Simulation Performance Optimization. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, 1992.

[XB07] Z. Xu and R. Bagrodia. GPU-Accelerated Evaluation Platform for High Fidelity Network Modeling. In *Proceedings of the 21st Internal Workshop on Principles of Advanced and Distributed Simulation*, 2007.

[YM89] C. Q. Yang and B. P. Miller. Performance Measurement for Parallel and Distributed Programs: A Structured and Automatic Approach. *IEEE Transactions on Software Engineering*, 15(12):1615–1629, 1989.

[YWC07] J. Yang, Y. Wang, and Y. Chen. GPU Accelerated Molecular Dynamics Simulation of Thermal Conductivities. *Journal of Computational Physics*, 221(2):799–804, February 2007.

[ZBG98] X. Zeng, R. L. Bagrodia, and M. Gerla. GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, 1998.

[ZPK00] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Complex Dynamic Systems*. Academic Press, 2000.

Curriculum Vitae

Personal Details

Last Name:	Kunz
First Name:	Georg Johannes
Date of Birth:	January 25, 1982
Place of Birth:	Emmerich, NRW, Germany
Nationality:	German

Education

High School	Freiherr-vom-Stein Gymnasium, Kleve
1992 – 2001	Abitur: June 2001
University	RWTH Aachen University
2002 – 2007	Major: Computer Science, Minor: Biology
	Degree: Dipl.-Inform., with honors
PhD Student	RWTH Aachen University
2008 – 2013	Chair of Communication and Distributed Systems
	Adviser: Prof. Dr.-Ing. Klaus Wehrle