Predicting Runtime Performance Bounds of Expanded Parallel Discrete Event Simulations

Georg Kunz*, Simon Tenbusch*, James Gross‡, Klaus Wehrle*
*Communication and Distributed Systems, ‡Mobile Network Performance Group
RWTH Aachen University
{kunz,tenbusch,wehrle}@comsys.rwth-aachen.de, gross@umic.rwth-aachen.de

***Abstract*—Predicting and analyzing runtime performance characteristics is a vital step in the development process of parallel discrete event simulations. For instance, model developers need to identify and eliminate performance bottlenecks within a simulation model in order to derive a model structure that aids parallel execution. Similarly, developers of parallel simulation frameworks require means of assessing the efficiency of the framework. In this paper, we present a performance prediction methodology that computes the best possible performance bound for *expanded* parallel discrete event simulations in the context of our Horizon simulation framework. The methodology builds upon a linear program which calculates an optimal event execution schedule for a given simulation and a set of CPUs. In order to mitigate the complexity of this NP-complete scheduling problem, we introduce performance optimizations and relaxations of the linear program.**

I. Introduction

Parallel discrete event simulation [1] can significantly improve the runtime performance of complex simulation models [2]. However, developing high-performance parallel simulation frameworks and the corresponding parallel simulation models is a difficult and complex task. As a result, developers need performance analysis and prediction tools which provide an insight into the behavior of parallel simulations throughout the development and evaluation process.

Model Development Support: Developing a parallel simulation model is considerably more complex than creating an equivalent sequential model. Among the key contributing factors are restrictions of the programming model, e.g., no global data structures, and the demand for structuring the model in a way which allows for efficient parallel execution. Hence, model developers need to analyze the performance of a model as early as possible in the development process in order to identify and eliminate performance bottlenecks.

Simulation Setup: Parallel simulation models exhibit a certain degree of parallelism given by the maximum number of events that can be processed in parallel. The degree of parallelism obviously limits the number of CPUs that can speed up a parallel simulation run. Thus, accurate knowledge of the degree of parallelism allows maximizing the utilization of all available CPUs by assigning only as many CPUs to a simulation run as actually needed and using the remaining CPUs for additional simulation runs.

Simulation Framework Performance: Parallel simulation frameworks employ specific synchronization and event scheduling algorithms which have a major influence on simulation performance. In order to assess the efficiency of a given synchronization algorithm, framework developers need to know the optimal event schedule under consideration of event inter-dependencies and the number of available CPUs.

In this paper, we present a performance prediction methodology that calculates the lower bound on the simulation runtime for our parallel simulation framework Horizon [3]. Given a Horizon-enabled simulation model and an arbitrary number of CPUs, the performance prediction methodology finds an optimal event-to-CPU mapping that minimizes the simulation runtime. This mapping problem is NP-complete [4], [5] and is thus typically not considered by existing performance prediction tools [6], [7], [8], [9]. We address this complexity problem by modeling the parallelization approach of Horizon as a *linear program* (LP). As a result, we leave the actual problem of finding an optimal schedule to the efficient heuristics and algorithms of modern LP solvers. Moreover, to further mitigate the complexity problem, we present performance optimizations and relaxations of the linear program. Our methodology is split in two 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 linear program which calculates an optimal event schedule that minimizes the overall simulation runtime. In summary, we make the following contributions:

1) We introduce a methodology for *predicting* the runtime performance of a parallel simulation model based on *linear programming*.
2) We present a *trace splitting* scheme that significantly improves the scalability of our methodology while maintaining its accuracy. Furthermore, we prove the correctness of this optimization.
3) We discuss relaxations of the linear program which trade accuracy for scalability.

The remainder of this paper is structured as follows. First, we introduce the parallelization scheme underlying Horizon in Section II before conducting a problem analysis in Section III. Sections IV and V present our performance prediction methodology and corresponding performance optimizations, followed by an evaluation in VI. In Section VII, we discus the properties of our scheme before finally discussing related efforts in Section VIII and concluding in Section IX.

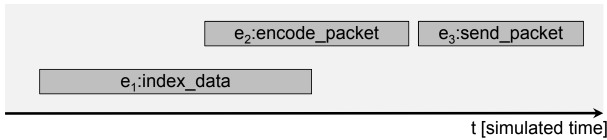


Fig. 1. The expanded events e_1 and e_2 cannot depend on each other due to their temporal overlapping and can thus be executed in parallel. e_3 must follow sequentially since it might depend on e_1 or e_2.

II. Background

This section briefly introduces the fundamentals of our simulation framework Horizon and its underlying parallelization scheme. Horizon enables a parallel execution of network simulation models by means of two properties: i) It introduces a modeling paradigm that extends discrete events with *durations* to explicitly and naturally model delays in discrete event simulation. ii) It defines a *parallelization scheme* that exploits the given event durations to determine independent events for a safe parallel execution. As the latter of the both properties indicates, Horizon employs a conservative parallelization scheme [1]. The primary challenge in those schemes is the identification of independent events that do not influence each other during parallel execution. Those events allow for a parallel execution while dependent events require a sequential execution with respect to each other.

Figure 1 shows a simple example that illustrates how Horizon utilizes event durations to identify independent events. The figure shows three expanded events e_1, e_2 and e_3 representing a "packet encoding", a "packet sending", and a "data indexing" process. We observe that in the particular timing chosen for this example, e_1 and e_2 overlap in simulated time[1] 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 of the other two events since it begins after the earlier events finished. In this example, it is indeed dependent on e_2 which calculates the encoded packet that is sent by e_3. We base this modeling paradigm on pioneering works by Lubachevsky [10] and Fujimoto [11].

Horizon employs a centralized event scheduling architecture specifically designed for multiprocessor systems. In contrast to related parallelization frameworks [12], [13], Horizon retains a centralized event scheduler and a single event queue (future event set, FES). Similarly to sequential simulators, the scheduler continuously removes the first event from the event queue, but then analyzes its overlapping with respect to previously executed events and finally offloads it to a worker thread for parallel execution. We demonstrated the viability of our approach in previous work [3].

[1]We denote the virtual time within the simulation as "simulated time" in contrast to "simulation time" which represents the runtime of the simulation in "wall-clock time".

III. The Need for Performance Prediction

Parallel discrete event simulations can significantly reduce simulation runtimes [12], [14], [15], [16]. However, parallel simulations are not generally used in the research community. We accredit this to the fact that development and handling is noticeably more complex for parallel simulations than for sequential ones. Hence, we believe that simulation model developers need specific tools which support the development process of parallel simulations in order to foster their wide spread use. Before introducing our performance analysis methodology in detail, we first illustrate the demand for performance prediction tools in the development process of parallel simulations. We then discuss the challenges of finding an optimal event schedule.

Model Development Support: The structure of a simulation model has a significant impact on its parallel runtime performance. For instance, in order to make use of parallel execution, a simulation model needs to be partitioned in relatively independent sub-parts. The key challenge of this process lies in finding a partitioning which allows for a maximum degree of parallelism and hence an adequate utilization of the CPUs. Thus, simulation model developers need means of analyzing the runtime behavior of a given model in order to i) understand why a model exhibits a performance bottleneck and ii) derive optimizations of the model structure to improve its parallel performance.

Simulation Setup: A given parallel simulation model exhibits a certain degree of internal parallelism, i.e., the maximum number of tasks that can be executed in parallel at a given point in simulated time. Obviously, the degree of parallelism constitutes an upper bound on the useful number of CPUs: Increasing the number of CPUs beyond this threshold results in the surplus CPUs being idle at any point in simulation time. Accurate knowledge of this number of CPUs, however, is imperative to maximize the utilization of all CPUs available to the model developer. This involves primarily reserving the right number of CPUs to each instance of a parallel simulation in order to maximize the number of concurrent simulation instances. The latter is particularly important when conducting large numbers of independent simulation runs, e.g., as part of a parameter study or to obtain results with statistical confidence.

Simulation Framework Performance: Parallel simulation frameworks naturally put a special focus on efficient event handling to achieve maximum runtime performance. Hence, it is important to determine the overhead of the simulation framework to assess the potential for performance improvements. Similarly, developers require a means of measuring and comparing the effectiveness of event scheduling algorithms. In the case of Horizon for instance, given a set of independent events the event scheduler can employ different offloading strategies such as earliest deadline first (EDF). In order to determine the efficiency of a scheduling strategy and compare different strategies against each other, a global lower bound on the absolute minimum runtime is needed.

Our goal is to design a performance 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 prediction tool needs to precisely reflect the parallelization approach of Horizon in order to calculate a valid schedule. Specifically, it must respect event dependencies, partitioning related scheduling restrictions, and the available processing resources. This mapping problem is a special case of the NP-complete *parallel machine scheduling problem* [4], [5]: Given l (identical) machines and n different jobs, each with processing times p_j, the task is to assign each job to a machine, such that the maximum of the completion times c_j of the events is minimal: minimize $\max_{j \in \{1 \dots n\}} c_j$. 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 ordering, 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 (in-)dependencies among events and ii) the simulation time which specifies the utilization of processing resources. The solution of this scheduling problem yields an upper bound on the speedup one can expect when executing the given model on l CPUs compared 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 [17], [18].

IV. Performance Prediction Methodology

In this section, we introduce the general concept of our performance prediction methodology and present the formal definition of the linear program. Deriving an optimal event schedule is a two-stage process: The first step involves executing a given simulation model sequentially while tracing runtime performance information. In a second step, the linear program takes this information as input and subsequently calculates an optimal event schedule.

A. Tracing Simulation Runtime Data

In order to calculate an optimal event schedule, the linear program 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 *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*[2] in which the event is executed.

While the purpose of the event ID and the processing time is straightforward, the starting and completion times are used to determine event dependencies according to the "overlapping"-property of expanded events. Furthermore, to ensure data consistency within the simulation model, Horizon permits only one active worker thread 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 them 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.

B. Problem Definition

In this section we give a formal definition of the event scheduling problem at hand. First, we define the input parameters and introduce the notation used in following sections. We then characterize a valid solution of the scheduling problem and analyze its properties.

Definition 1: The *input* of an event scheduling problem is a 6-tuple (E, C, p, s, f, m) where:

- $E = \{e_1 \dots e_n\} \subset \mathbb{N}$ represents the set of *events*.
- $C = \{c_1 \dots c_l\} \subset \mathbb{N}$ represents the set of *CPUs*.
- $p : E \to \mathbb{R}^+,\ e \mapsto$ Event processing time of e.
- $s : E \to \mathbb{R}^+,\ e \mapsto$ Starting time of e.
- $f : E \to \mathbb{R}^+,\ e \mapsto$ Completion 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 $d, e \in E$ with $d < e$ it holds $s(d) \leq s(e)$. Further, two events $d, e \in E$ *overlap* iff

$$s(d) \leq s(e) \wedge s(e) \leq f(d) \text{ or } s(e) \leq s(d) \wedge s(d) \leq f(e)$$

In the following, we denote two overlapping events d and e with $d \parallel e$. Finally, it is important to note that $f(e) - s(e)$ is the duration of event e in simulated time which is not to be confused with the processing time $p(e)$ 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 compute 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) derives a valid ordering of events for each of the CPUs. We formally define a schedule as follows:

Definition 2: For an input (E, C, p, s, f, m) a *schedule* $\mathcal{S}$ is a tuple of mappings (x, y) where:

[2]An OMNeT++/Horizon simulation model is composed of modules which implement the actual model logic. In Horizon, modules are similar to the concept of logical processes in classic parallel discrete event simulation since they communicate only by sending messages and maintain a local state.

- $x : E \to C,\ e \mapsto c$.
  $x(e)$ assigns event e to CPU c for execution.
- $y : E \to \mathbb{R}^+,\ e \mapsto t$.
  $y(e)$ denotes the point t in simulation time at which the execution of e starts on the CPU assigned by $x(e)$.

Furthermore, we define a feasible schedule as follows:

Definition 3: For an input (E, C, p, s, f, m) a schedule $\mathcal{S} = (x, y)$ is *feasible* iff:

a) For all $d, e \in E$ with $x(d) = x(e)$ or $m(d) = m(e)$:

$$y(d) \geq y(e) + p(e) \text{ or } y(e) \geq y(d) + p(d)$$

"Events mapped to the same CPU and events of the same module are processed sequentially."

b) For all $d, e \in E$ with $f(d) < s(e)$:

$$y(d) + p(d) \leq y(e)$$

"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 4: For an input (E, C, p, s, f, m) a schedule $\mathcal{S} = (x, y)$ is *optimal* iff:

a) $\mathcal{S}$ is feasible

b) For all feasible schedules $\mathcal{S}' = (x', y')$:

$$R := \max_{e \in E}(y(e) + p(e)) \leq \max_{e' \in E}(y'(e') + p(e'))$$

Definition 4 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.

C. Linear Program Formulation

We now formulate a linear program that takes an input (E, C, p, s, f, 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 simulation time and $z_{d,e} = 0$ if the execution of e starts before d.

Additionally, we define the variable R which holds the total runtime of the schedule. Furthermore, M is a large positive constant. Based on these variables, the linear program is defined as follows:

Objective function:

$$\text{minimize } R$$

subject to the following constraints:

$\forall e \in E$:

$$\sum_{c \in C} x_{e,c} = 1 \quad (1)$$

$$y_e + p(e) \leq R \quad (2)$$

$\forall c \in C$, $\forall d, e \in E$ with $d < e$ and $d \parallel e$ and $m(d) \neq m(e)$:

$$y_e - y_d + (1 - z_{d,e}) \cdot M \geq (x_{d,c} + x_{e,c} - 1) \cdot p(d) \quad (3)$$

$$y_d - y_e + z_{d,e} \cdot M \geq (x_{d,c} + x_{e,c} - 1) \cdot p(e) \quad (4)$$

$\forall c \in C$, $\forall d, e \in E$ with $d < e$ and $d \parallel e$ and $m(d) = m(e)$:

$$y_e - y_d + (1 - z_{d,e}) \cdot M \geq p(d) \quad (5)$$

$$y_d - y_e + z_{d,e} \cdot M \geq p(e) \quad (6)$$

$\forall d, e \in E$ with $f(d) < s(e)$:

$$y_d + p(d) \leq y_e \quad (7)$$

Constraint 1 ensures that each event is assigned to exactly one CPU. Additionally, Constraint 2 guarantees that R is an upper bound on the runtime of the schedule. Constraints 3 to 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 $z_{d,e}$ (Constraints 3 and 4). Furthermore, events on the same module are executed sequentially, again with the order depending on the value of $z_{d,e}$ (Constraints 5 and 6). Finally, Constraint 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 $x_{e,c} = 1$.

V. Scalability Improvements

Since solving this NP-complete scheduling problem is hard, we rely on the heuristics and algorithms [19] of modern LP solvers which still allow for computing a (nearly) optimal solution in a reasonable amount of time. However, for large inputs, the scheduling problem becomes computationally intractable. In this section, we present approaches to counteract the scalability problem with and without sacrificing the accuracy of the results.

A. Splitting Schedules

The first approach towards increasing the scalability of the performance prediction methodology aims at reducing the input size of the linear program while at the same time retaining the correctness of the resulting schedule. It bases on the observation that in general the sequence of expanded events contains regions of non-overlapping events (see Figure 2). Since the conservative synchronization algorithm of Horizon

executes only safe, i.e., overlapping, events in parallel, the event scheduler blocks until all events preceding the non-overlapping region have been processed. Hence, these regions act as natural synchronization points in the parallel simulation. 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 linear program. We then iteratively reconstruct a valid schedule for the full trace from the set of sub-schedules. Specifically, the total runtime is given by the sum of the runtimes calculated for each sub-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 5: For an input (E, C, p, s, f, 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 $e \in E_2 : f(d) < s(e)$

Based on this definition, we now formulate the central theorem.

Theorem 1: Given an input (E, C, p, s, f, m), with a split (E_1, E_2) of E and optimal schedules $\mathcal{S}_j = (x_j, y_j)$ on $(E_j, C, p|_{E_j}, s|_{E_j}, f|_{E_j}, m|_{E_j})$, $j \in \{1, 2\}$, then, $\mathcal{S} = (x, y)$ with

$$
\begin{aligned}
x(e) \quad &:= \quad x_j(e) \text{ for } e \in E_j \\
y(e) \quad &:= \quad \begin{cases} y_1(e) & \text{for } e \in E_1 \\ \max\limits_{d \in E_1} y_1(d) + p(d) + y_2(e) & \text{for } e \in E_2 \end{cases}
\end{aligned}
$$

is an optimal schedule on (E, C, p, s, f, m).

By iterated 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 1: Given an input (E, C, p, s, f, m), with a split (E_1, E_2) of E and feasible schedules $\mathcal{S}_j = (x_j, y_j)$ on $(E_j, C, p|_{E_j}, s|_{E_j}, f|_{E_j}, m|_{E_j})$, $j \in \{1, 2\}$, then $\mathcal{S} = (x, y)$ as defined in Theorem 1 is a feasible schedule for (E, C, p, s, f, m).

Proof: We prove the two conditions of feasibility for the combined schedule $\mathcal{S}$.
Definition 3 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 $d \in E_1, e \in E_2$:

$$
\begin{aligned}
y(d) + p(d) &= y_1(d) + p(d) \\
&\leq \max_{e' \in E_1} y_1(e') + p(e') \\
&\leq \max_{e' \in E_1} y_1(e') + p(e') + y_2(e) \\
&= y(e). \qquad (*)
\end{aligned}
$$

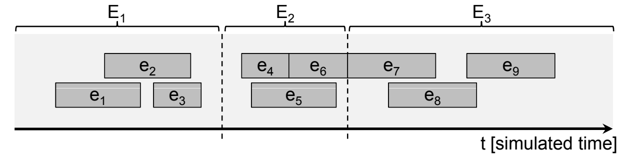


Fig. 2. The sequence of expanded events in a simulation might contain regions of non-overlapping events which allow for splitting the event trace.

Definition 3 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 $f(d) < s(e)$. Finally, from (*) follows Definition 3 b). □

We can now prove Theorem 1:

Proof: We show that the combined schedule $\mathcal{S}$ fulfills the two conditions of optimality.
Definition 4 a): Follows directly from Lemma 1.
Definition 4 b): The runtime of the combined schedule is

$$
\begin{aligned}
R &= \max_{e \in E} y(e) + p(e) \\
&\overset{(*)}{=} \max_{e \in E_2} y(e) + p(e) \\
&= \max_{e \in E_2} \max_{d \in E_1} y_1(d) + p(d) + y_2(e) + p(e) \\
&= \left(\max_{d \in E_1} y_1(d) + p(d) \right) + \left(\max_{e \in E_2} y_2(e) + p(e) \right) \\
&= R_1 + R_2
\end{aligned}
$$

Proof by contradiction: Assume there exists a feasible schedule $\mathcal{S}' = (x', y')$ with $R' = \max_{e \in E} y'(e) + p(e) < R$. 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 3 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. □

B. Eliminating Events with Insignificant Processing Times

A further approach towards reducing the input size of the linear program 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 [3]. As a result, long running events can completely dominate short running events in terms of the total simulation runtime. We conclude from this 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 linear program.

By carefully eliminating all events below a given runtime threshold from the event trace, the complexity of solving the linear program decreases while the error introduced to the solution remains within certain bounds. The maximum error of the lower bound is easily derivable from the total runtime of all dropped events D, the number of CPUs $|C|$, and the calculated total runtime R. When events with a total runtime of D are dropped from the trace, $R + \frac{1}{|C|} \cdot D$ is a lower bound even

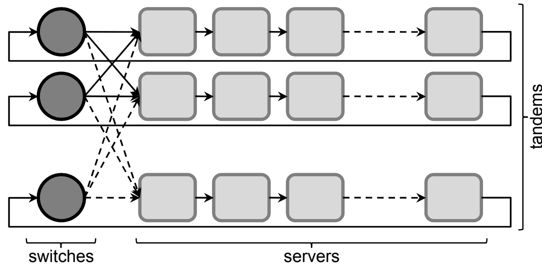


Fig. 3. Structure of the closed queueing network utilized in the evaluation. The servers continuously process and forward incoming tasks. Each switch distributes incoming tasks randomly to the tandem queues.

when the dropped events can be equally distributed across all available CPUs. The maximum error in this case is $\frac{|C|-1}{|C|} \cdot D$ if the events instead need to be computed sequentially.

C. Relaxations

In addition to reducing the input size of the linear program, its scalability can also be improved by relaxing constraints at the price of less accurate runtime bounds. In the following, we discuss two different relaxations.

1) Relaxed CPU Load Limit: Replace Constraints 3 and 4 with a less strict formulation:

$$\forall c \in C : \sum_{e \in E} (x_{e,c} \cdot p(e)) \leq R$$

"The sum of the processing times of all events assigned to CPU c is bounded by the overall runtime R."

In contrast to requiring that a CPU must not be overloaded at any point in time, i.e., handling not more than one event, the relaxed constraint only 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 linear program 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 linear program while still retaining a reasonable lower bound on the overall simulation runtime.

2) No CPU Scheduling: Remove Constraints 1, 3 and 4 from the linear program.

This relaxation does not consider the workload of CPUs anymore, but focuses on event dependencies only. Thus, the predicted simulation runtime is a lower bound on the simulation runtime with an infinite number of CPUs. It hence provides information on the degree of parallelism within the model. From this information, one can easily derive the maximum possible speedup factor and an upper bound on the number of useful CPUs.

Concluding, by means of trace splitting, event dropping, and by carefully applying relaxations, the complexity of the linear program reduces significantly- We evaluate the exact impact of the aforementioned techniques in Section VI.

VI. Evaluation

In the following sections, we evaluate the accuracy, complexity, and performance of the runtime prediction scheme and its optimizations. To this end, we first introduce the evaluation methodology before presenting the actual evaluation results.

A. Methodology

1) Evaluation Model: We base the evaluation of our performance prediction methodology on a model of a closed queueing network as depicted in Figure 3. The network consists of 10 tandem queues – each composed of a chain of 7 servers and a switch. Initially, each server creates a task with a time-to-live (ttl) value of 23 and sends it to its neighboring server. Subsequently, the servers continuously process incoming tasks according to an exponentially distributed service time (mean 0.1s), 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. For simplicity, the links in the network exhibit a static delay of 1.5s. In order to confront the linear program with a wide range of event processing times, handling a task involves a dummy computation of uniformly distributed length (0s – 0.3s).

2) Implementation and Setup: We utilize an extended implementation of Horizon which builds upon OMNeT++ 3.3 [20]. The modifications of Horizon mainly target the central event scheduler and comprise the necessary functionality to create an event trace during sequential execution. The linear program is implemented using Zimpl v3.1.0 (Zuse Institute Mathematical Programming Language) [21]. The Zimpl compiler translates the linear program and a given event trace into a format suitable for the IBM ILOG CPLEX v12.2 LP solver used in this evaluation. All measurements were conducted on a 6-core AMD Opteron 2431 CPU using 32GB of RAM.

B. Accuracy

We assess the prediction accuracy of the linear program 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 the case of sequential execution on 1 CPU. For each number of CPUs, we then apply the linear program, Relaxation 1, and Relaxation 2 to an event trace previously collected during a sequential simulation run. For simplicity, the event trace only includes the pure processing time of each event, not considering the event handling overhead of the simulation framework. Furthermore, because of the regular behavior of the simulation model, we don't expect changes in the runtime behavior of the simulation. Hence, we restrict the ttl of each task to 23 which results in a relatively short trace of about 1600 events. For all three versions of the linear program, the LP solver computes a solution within at least 0.1% optimality.

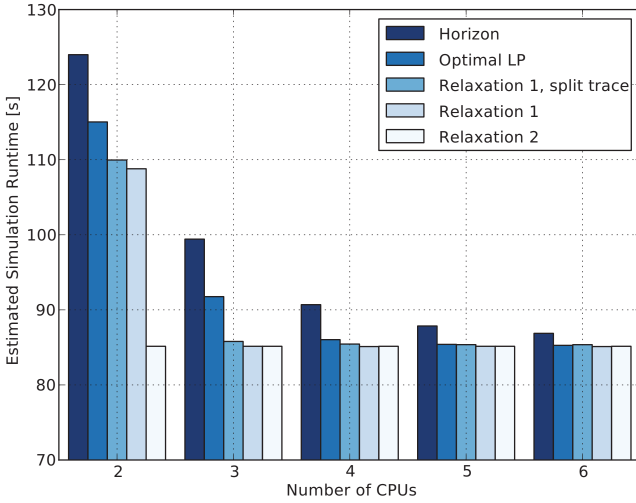
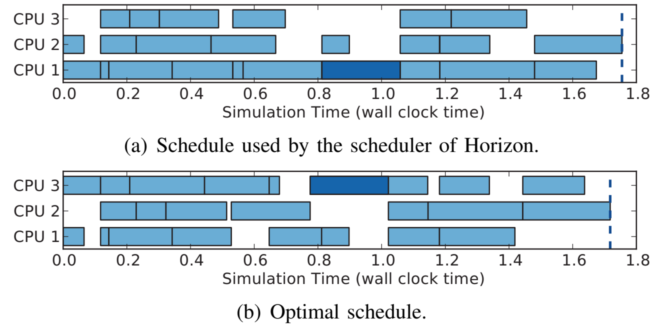


Fig. 4. Predicted simulation runtimes of the linear program and its relaxations in comparison to the actual runtime of Horizon. We only focus on parallel runtime performance and hence omit sequential execution on 1 CPU.

Figure 4 illustrates the measured runtimes of Horizon in comparison to the runtimes predicted by the linear program and its relaxations. We observe that the runtimes of Horizon clearly exceed the predicted runtimes. Since the event trace considers only pure event processing times, this difference in performance may be caused either by the event handling overhead of the simulation framework or a sub-optimal event schedule. However, we argue that the event handling overhead is negligible in comparison to the processing times which average around $0.15s$. Hence, we conclude that the event scheduling algorithm of Horizon does not achieve an optimal event schedule. We identify two primary reasons for this result: i) The event scheduler of Horizon has no global knowledge as opposed to the linear program which computes an optimal schedule for the whole event trace. In particular, at any point in time in the simulation, the event scheduler can only incorporate those events in its scheduling decision which are in the future event set of the simulation. ii) In its current implementation, the event scheduler only examines the first event of the future event at any time during the simulation. Instead, it may achieve a better performance by analyzing sets of overlapping events in order to increase its knowledge.

As expected, both relaxations compute runtime bounds which deviate 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 V-C, Relaxation 1 increases in accuracy when used in conjunction with our splitting scheme. Figure 4 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.

The charts in Figure 5 compare the optimal schedule against the one used by Horizon for executing the first 24 events of the simulation. Besides a shorter overall runtime of the optimal schedule, we observe that the highlighted event blocks the following events. Given this information, developers can now analyze this event with respect to its interdependencies and its runtime demand in order to optimize parallel performance.



(a) Schedule used by the scheduler of Horizon.

(b) Optimal schedule.

Fig. 5. Comparison of the schedule used by Horizon with an optimal schedule for the first 24 events of the evaluation trace.

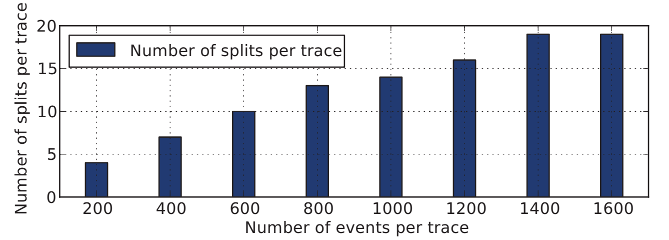
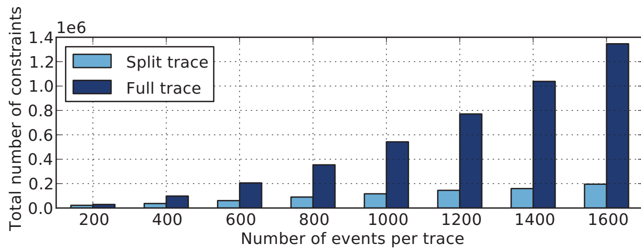


Fig. 6. Number of splits generated from input traces of specific length. The regular behavior of the queueing network allows for evenly distributed splits.

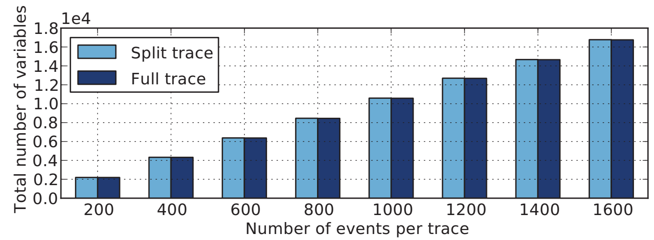
C. Scalability

In the following, we evaluate the performance improvements achieved by the trace splitting scheme and the relaxations of the linear program.

1) Input Complexity: The primary goal of the trace splitting scheme is to reduce the input size of the LP-solver by dividing the full scheduling problem into a set of sub-problems. We measure the input size of the LP-solver in terms of the number of constraints and the number of variables. As described in Section VI-A2, we use Zimpl to specify the 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. Please note that in the following, results regarding split traces always refer to the sum over all splits.



(a) Number of constraints in the linear program after running Zimpl.



(b) Number of variables in the linear program after running Zimpl.

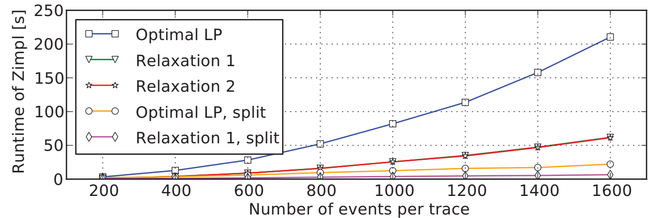
Fig. 7. Size of the linear program in terms of the number of constraints and the number of variables after running Zimpl on full and split input traces of varying length. The number of CPUs in the scheduling problem is fixed to 5. Results for split traces are the sum over all sub-traces.

Figure 7(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 3-7. However, the small and equal size of the splits prevents this polynomial characteristic from gaining considerable impact per split. Instead, the splitting scheme effectively transforms the polynomial growth into a linear growth over the number of splits. In contrast, Figure 7(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 (see Def. 5).

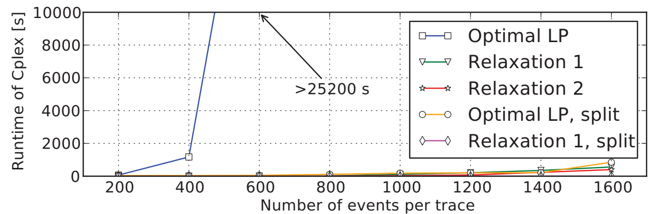
2) Runtime: We analyze the performance improvements of the splitting scheme and the relaxations by investigating the runtime of the performance prediction scheme. Since our implementation builds upon Zimpl and CPLEX, the total runtime of the prediction scheme is the sum of the runtimes of both tools. We measure the runtime of Zimpl by means of the tool *time* while CPLEX itself provides detailed information on the time to solve a given problem. Furthermore, we restrict CPLEX to single threaded execution.

The combined Figures 8 illustrate the runtimes of CPLEX and Zimpl for the linear program, Relaxation 1 (both with and without splitting), and Relaxation 2 for traces ranging from 200 to 1600 events. Specifically, Figure 8(a) focuses on the runtimes of Zimpl. For full event traces, the runtimes for the 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 the relatively equal size of the sub-traces. The next two Figures depict the runtimes of CPLEX. As shown in Figure 8(b), the runtimes of CPLEX for solving the linear program dwarf the time demand of all other schemes. This exponential growth confirms the computational 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 linear program and such traces.

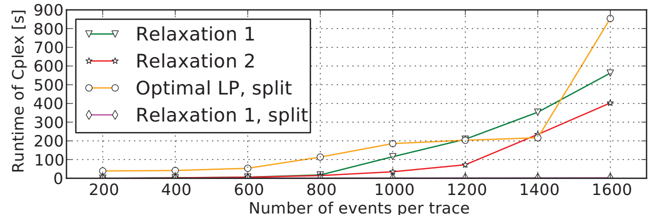
To allow for a more detailed analysis of the remaining



(a) Runtimes of Zimpl.



(b) Runtimes of CPLEX including the optimal linear program.



(c) Runtimes of CPLEX without the runtimes of the optimal linear program.

Fig. 8. Runtimes of the performance prediction schemes for traces of specific length. The runtimes of CPLEX for the linear program without splitting for are omitted for trace of more than 600 events due to excessive runtimes.

schemes, Figure 8(c) 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 linear program. In the case of the linear program with split inputs, the graph shows alternating regions of no increase in runtime and regions of considerable increases. We accredit this to the fact that the runtimes under 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 reveals that no splittable region exist between event 1400 and 1600. As a result, the size of the last split of the 1600 event trace increases which results in a longer overall runtime. Due to the exponential complexity of the scheduling problem, an increase of the input size has a larger impact on

the runtime than the polynomial complexity of Zimpl. This is why the results for Zimpl in Figure 8(a) do not show similar fluctuations. Finally, we observe from Figure 8(c) that the runtime demand of Relaxation 1 for split input traces is barely noticeable at the bottom of the figure. Hence, the combination of the splitting scheme and a relaxed scheduling problem achieves a highly scalable runtime performance.

VII. Discussion

The evaluation of our performance prediction methodology illustrates that the trace splitting scheme and the relaxations of the linear program achieve a significant performance improvement over the (non-relaxed) linear program. 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 outside the reach of our scheme to predict the runtime of a whole simulation run. Nevertheless, we argue that the presented performance prediction 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. In this case, selectively tracing the model under investigation is sufficient and yields traces of acceptable size.

Due to the trace-based approach of our performance prediction methodology, the estimated runtimes are tightly coupled to the performance of the underlying evaluation hardware. 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 constant and thus the event schedule calculated by the linear program stays valid.

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 simulation framework. The latter is caused by all management operations within the framework such as event creation, deletion, insertion into the event queue or removal from the event queue. In order to accurately compare the quality of a given scheduling strategy against the optimal schedule, the overhead of the simulation framework should be included in the event processing times. Since the overhead is mostly static for all events, it suffices to determine the overhead once for the simulation framework and subsequently add it to the traced event processing times.

VIII. Related Work

This section discusses closely related research efforts regarding performance prediction of parallel simulations.

Critical Path Analysis: Pioneered by Berry et al. [22], critical path analysis [23], [24] 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 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 infinite 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. [6] introduce an alternative construction algorithm 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 infinite number of CPUs. In order to apply critical path analysis to a limited set of CPUs, Lin et al. [7] 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.

Synchronization Overhead Estimation: In contrast to critical path analysis, OSim [25] by Swope et al. utilizes the event trace to optimally synchronize an actual parallel simulation. The goal of OSim is to reconstruct event dependencies from the event trace at runtime and thereby eliminate the need for actual synchronization protocols such as the null-message algorithm [26]. 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) [8]. 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.

Resource-based Performance Analyzers: Juhasz et al. [9] 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 scheme.

Finally Liu et al. [27] illustrate an alternative approach to 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 across the CPUs and hence cannot very well handle load asymmetries which naturally exist in complex simulation models.

IX. Conclusion and Future Work

In this paper, we presented a performance prediction methodology which provides simulation model developers with an insight into the execution behavior of a parallel simulation model for the Horizon simulation framework. Specifically, our methodology calculates an optimal event schedule and thus the lower bound on the runtime of a given parallel simulation model. This insight supports developers in deriving performance optimizations for simulation models, frameworks, and evaluation setups. Furthermore, we presented a novel trace splitting scheme that transforms the exponential complexity of the scheduling problem into a linear one.

Our future work focuses on the centralized event scheduler of Horizon. Currently, the event scheduler relies on a simple FIFO strategy (with respect to the FES) when offloading independent events. This results, however, in a suboptimal simulation performance: Since the scheduler does not take the processing times of events into account it cannot maximize the overall CPU utilization. Instead, we intend to develop and adopt more advanced online scheduling strategies such as longest processing time first. The evaluation of these scheduling strategies fundamentally relies on the performance prediction tool presented in this paper.

Acknowledgments This research was funded by the DFG Cluster of Excellence on Ultra High-Speed Mobile Information and Communication (UMIC), German Research Foundation grant DFG EXC 89.

References

[1] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, no. 10, 1990.

[2] R. M. Fujimoto, K. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley, "Large-Scale Network Simulation: How Big? How Fast?" in *Proc. of 11th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2003.

[3] G. Kunz, O. Landsiedel, J. Gross, S. Götz, F. Naghibi, and K. Wehrle, "Expanding the Event Horizon in Parallelized Network Simulations," in *Proc. of the 18th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.

[4] T. Cheng, "A state-of-the-art Review of Parallel-machine Scheduling Research," *European Journal of Operational Research*, vol. 47, no. 3, August 1990.

[5] J. K. Lenstra and A. H. G. R. Kan, "Complexity of Scheduling under Precedence Constraints," *Operations Research*, vol. 26, no. 1, 1978.

[6] F. Wieland, T. Som, P. Reiher, J. Wedel, and D. Jefferson, "A Critical Path tool for Parallel Simulation Performance Optimization," in *Proc. of the 25th Hawaii International Conference on System Sciences*, 1992.

[7] Y. B. Lin, "Parallelism Analyzers for Parallel Discrete Event Simulation," *Transactions on Modeling and Computer Simulation*, vol. 2, no. 3, July 1992.

[8] R. L. Bagrodia and M. Takai, "Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 4, 2002.

[9] Z. Juhasz, S. Turner, K. Kuntner, and M. Gerzson, "A Performance Analyser and Prediction Tool for Parallel Discrete Event Simulation," *International Journal of Simulation*, vol. 4, no. 1, May 2003.

[10] B. D. Lubachevsky, "Efficient Distributed Event Driven Simulations of Multiple-loop Networks," in *Proc. of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1988.

[11] R. M. Fujimoto, "Exploiting Temporal Uncertainty in Parallel and Distributed Simulations," in *Proc. of the 13th Workshop on Parallel and Distributed Simulation*, 1999.

[12] P. Peschlow, A. Voss, and P. Martini, "Good News for Parallel Wireless Network Simulations," in *Proc. of the 12th International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2009.

[13] G. Seguin, "Multi-core Parallelism for ns-3 Simulator," INRIA Sophia-Antipolis, Tech. Rep., 2009.

[14] L. Bononi, M. Di Felice, M. Bertini, and E. Croci, "Parallel and Distributed Simulation of Wireless Vehicular Ad hoc Networks," in *Proc. of the 9th International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2006.

[15] R. Fujimoto, "Performance of Time Warp under Synthetic Workloads," *Proceedings of the SCS Multiconference on Distributed Simulation*, vol. 22, no. 1, 1990.

[16] G. Chen and B. K. Szymanski, "DSIM: Scaling Time Warp to 1,033 Processors," in *Proc. of the 37th Winter Simulation Conference*, 2005.

[17] D. Jefferson and P. Reiher, "Supercritical Speedup," *ACM SIGSIM Simulation Digest*, vol. 21, Apr. 1991.

[18] S. Srinivasan and P. F. Reynolds, "Super-criticality Revisited," in *Proc. of the 9th Workshop on Parallel and Distributed Simulation*, 1995.

[19] T. K. Ralphs, L. Ladányi, and M. J. Saltzman, "Parallel Branch, Cut, and Price for Large-scale Discrete Optimization," *Mathematical Programming*, vol. 98, no. 1, Sep. 2003.

[20] A. Varga, "The OMNeT++ Discrete Event Simulation System," in *Proc. of the 15th European Simulation Multiconference (ESM)*, 2001.

[21] T. Koch, "Zimpl User Guide," 2011. [Online]. Available: http://zimpl.zib.de/

[22] O. Berry and D. Jefferson, "Critical Path Analysis of Distributed Simulation." in *Proc. of the SCS Conf. on Distributed Simulation*, 1985.

[23] C. Q. Yang and B. P. Miller, "Performance Measurement for Parallel and Distributed Programs: A Structured and Automatic Approach," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, 1989.

[24] 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.

[25] S. M. Swope and R. M. Fujimoto, "Optimal Performance of Distributed Simulation Programs," in *Proc. of the 19th Winter Simulation Conference*, 1987.

[26] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, September 1979.

[27] J. Liu, D. Nicol, B. J. Premore, and A. L. Poplawski, "Performance Prediction of a Parallel Simulator," in *Proc. of the 13th Workshop on Parallel and Distributed Simulation (PADS '99)*, 1999.