Know Thy Simulation Model: Analyzing Event Interactions for Probabilistic Synchronization in Parallel Simulations

Georg Kunz*, Mirko Stoffers‡, James Gross‡, Klaus Wehrle*
*Communication and Distributed Systems, ‡Mobile Network Performance
RWTH Aachen University
*lastname@comsys.rwth-aachen.de, ‡lastname@umic.rwth-aachen.de

ABSTRACT

Efficiently scheduling and synchronizing parallel event execution constitutes the fundamental challenge in parallel discrete event simulation. Existing synchronization algorithms typically do not analyze event interactions within the simulation model – mainly to minimize runtime overhead and complexity. However, we argue that disregarding event interactions results in a lack of insight into the *behavior* of the simulation model, thereby severely limiting synchronization efficiency and thus parallel performance. In this paper, we present a probabilistic synchronization scheme that obtains extensive *knowledge* of the simulation behavior at runtime to guide event execution. Specifically, we design three *heuristics* that dynamically derive event dependencies from tracing event interactions and decide whether or not to speculatively execute events. Our evaluation shows that the proposed probabilistic synchronization scheme considerably outperforms traditional conservative and optimistic schemes.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]: Types of Simulation—*Parallel, Discrete Event*

General Terms

Algorithms, Design, Performance

Keywords

Parallel Network Simulation, Probabilistic Synchronization

1. INTRODUCTION

The primary goal of Parallel Discrete Event Simulation (PDES) is enabling a 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 [7]. 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 [17]. 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 [3, 4, 12, 13, 14] 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 lookaheads still constitute lower bounds for the actual degree of parallelism in the model. Corresponding efforts aim at restricting the degree of speculative execution in optimistic synchronization, for instance by means of time windows [25]. However, like conservative schemes, such windows impose artificial restrictions on parallel event execution if they do not base on accurate knowledge of the simulation behavior. Finally, pioneering efforts towards probabilistic synchronization [5, 6, 24] analyze the timing between events, yet such patterns do not convey enough information to accurately reconstruct event dependencies.

In previous work [11], we developed a parallel discrete event simulation framework, named HORIZON, that builds upon a centralized event scheduling architecture. Based on this work, we present in this paper a probabilistic synchronization scheme that gathers extensive *knowledge* of the simulation behavior in order to make *educated* event scheduling decisions. At simulation runtime, the scheme continuously collects event scheduling information 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. Adding a heuristic to the event

Simutools 2012, March 19-23, Desenzano del Garda, Italy

DOI 10.4108/icst.simutools.2012.247716

scheduling process obviously increases the runtime overhead. Yet, a complex heuristic might compute more accurate event dependencies than a less complex one. As a result, we design three heuristics that trade off complexity against accuracy: 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 high complexity and accuracy. In summary, we make the following two contributions in this paper:

1. We introduce a generic probabilistic synchronization scheme for parallel discrete event simulations.
2. We design and evaluate three heuristics which trade off computational complexity against prediction accuracy.

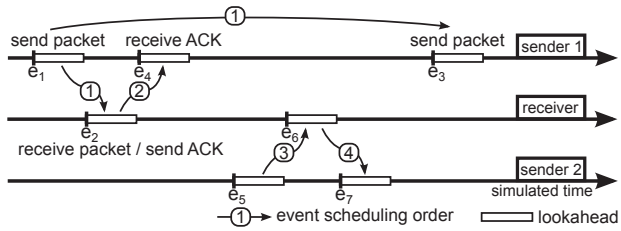
Our evaluation shows that all three heuristics are able to learn the behavior of a simulation model and considerably improve simulation performance over traditional schemes.

The remainder of this paper is structured as follows. Section 2 gives a brief introduction to our simulation framework HORIZON before we analyze the drawbacks of traditional synchronization techniques in Section 3. From this analysis we derive the design of the probabilistic synchronization scheme and the three heuristics in Section 4. We discuss limitations in Section 5, followed by an evaluation in terms of prediction quality, overhead, and performance gain in Section 6. Finally, we review related work in Section 7 and conclude in Section 8.

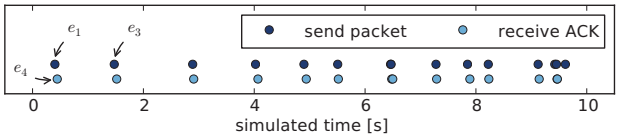
2. BACKGROUND

This section briefly introduces the fundamentals of our simulation framework HORIZON and its underlying parallelization scheme. HORIZON enables parallel execution of discrete event simulations by means of two properties: i) It defines a modeling paradigm that extends discrete events with durations to explicitly and naturally model delays in discrete event simulations [11]. This modeling paradigm lays the foundation for a conservative parallelization scheme that exploits the given event durations to determine independent events for parallel execution. ii) HORIZON employs a centralized event scheduling architecture specifically designed for multiprocessor systems. In contrast to related parallelization frameworks [18, 23], HORIZON retains a centralized event scheduler and a single Future Event Set (FES). Similar to sequential simulators, the central scheduler continuously removes the first event from the FES, but then determines whether or not it is independent to currently executing events and finally *offloads* it to a worker thread for parallel execution. The centralized architecture limits scalability, but avoids the need for load-balacing and partitioning schemes. Horizon hence aims at making parallelization available to users who do not want to deal with load balancing and partitioning, but still want to exploit the parallel processing power of their multi-core desktop systems. As a result, Horizon focuses on small to medium scale workstation systems and explicitely trades scalability for simplicity.

Of those two properties of HORIZON, only the centralized event scheduling architecture is of direct relevance in the context of this paper. The centralized scheduler allows for efficiently collecting, maintaining, and querying all event interactions in a central location without the need for distributed data management. In contrast, any conservative synchronization algorithm can form the basis for the probabilistic synchronization scheme presented in this paper.



(a) Global sequence of events modeling send-packet/receive-ACK transmissions between senders and receiver.



(b) Local sequence of send and receive events at sender 1.

Figure 1: Global (upper) and local (lower) sequence of events in a simple network model consisting of two senders and a receiver.

3. CHALLENGES

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

3.1 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 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.

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, giving rise to the "blocked waiting problem" [17].

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 per-

formance. For instance, Figure 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, the subsequent "send packet" event is eligible for parallel execution because no other event arrives in-between anymore. Thus, our goal in this paper is designing a synchronization scheme which analyzes such event patterns in order to improve parallel simulation performance.

3.2 Complexity vs. Prediction 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 results. 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 complexity of the simulation model. Probabilistic synchronization creates a performance gain over conservative synchronization if the heuristic decides to offload an event *before* all blocking events have been processed. This blocking period grows with increasing complexity 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 rolling back a computationally complex event has a greater impact on performance than rolling back a less complex one. 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. PROBABILISTIC SYNCHRONIZATION

We now introduce the goals and the concept of probabilistic synchronization and present the design of each heuristic.

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 following.

Because speculative execution might inflict a causal violation, the simulation framework periodically stores checkpoints of the simulation state and checks for causal correctness. Exploiting the fact that simulation models typically exhibit a modular structure, 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 of this paper, we distinguish between the set of events E and the set of event types T of a simulation model. Each event $e \in E$ is an instance of exactly one event type $\tau \in T$. Moreover, τ_e denotes the type of event e which can be uniquely identified at runtime.

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 [5, 6, 24]. Specifically, the heuristic tracks at each module

i) the type τ of the event which arrived last,
ii) for every event type υ, its number of occurrences n_υ,
iii) for every pair (τ, υ), the number $n_{\tau\upsilon}$, 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

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

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.

Applying this heuristic to the previously discussed example shown in Figure 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 $n_{\tau\upsilon}$ for the respective event types. Hence, these operations exhibit constant complexity. Similarly, the memory overhead is limited to

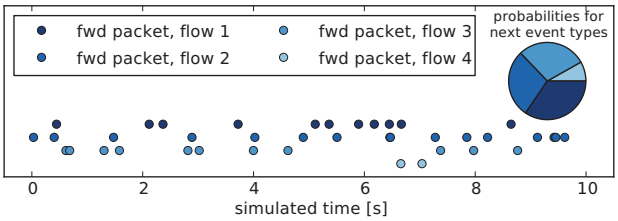


Figure 2: The local sequence of events on a network node that forwards four uncorrelated data streams. The pie chart on the right illustrates the computed probabilities that the next event is of the correspondingly colored type.

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 in certain scenarios. 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 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 segments of the pie chart in Figure 2). In addition, the heuristic does not distinguish between single events, 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 and induce a rollback. We conclude that analyzing the local arrival patterns of events, as done in related efforts, does not allow for accurately predicting future events and hence limits synchronization efficiency.

4.3 Global Order Heuristic

To obtain a better insight into causal dependencies among events, the Global Order Heuristic analyzes the "scheduled-by" relationship among events. Revisiting the initial example shown in Figure 1, we observe that a causal violation occurs at a module m if

i) two events e_1, e_2 execute speculatively in parallel, and
ii) e_1 with $t(e_1) < t(e_2)$ creates event e_3 with $t(e_3) < t(e_2)$,

where $t(e)$ denotes the timestamp of event e. 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 $t(e_\mathrm{s}) < t(e_\mathrm{p})$. 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 events 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

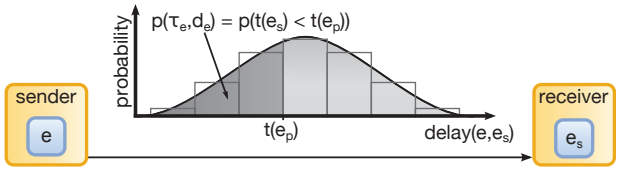


Figure 3: The delays between an event e and its earliest successor events e_s constitute the samples of a delay distribution. Each sample belongs to one bucket of the underlying histogram. The highlighted area under the curve represents the probability that an event e_s precedes e_p.

Figure 3. The probability $p(\tau_e, d_e)$ that an event e of type τ_e schedules another event within a delay of $d_e := t(e_\mathrm{p}) - t(e)$ 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 probability p_v for a causal violation as

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

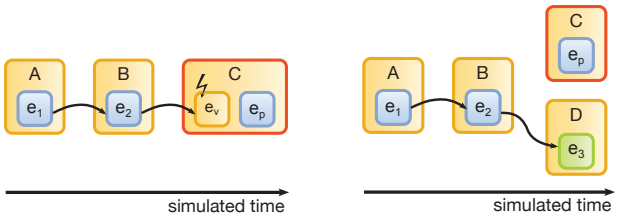
If this probability is below a user-defined threshold, the pending event e_p is executed speculatively.

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 3). Hence, determining p_v 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 $\mathcal{O}\big(\mathrm{ld}(|B|)\big)$ using binary search. In Horizon, $|O|$ is limited by the number of available CPUs, thus ranging between 4 and 32 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 6.1.3, 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 parallel performance. 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(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 modules increases monotonically [7]. 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(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 Local Order Heuristic

In order to follow the aforementioned path through the simulation model, the Local Order Heuristic needs to sample



(a) Causal violation. (b) No causal violation.

Figure 4: Causal correctness is a local property of each module.

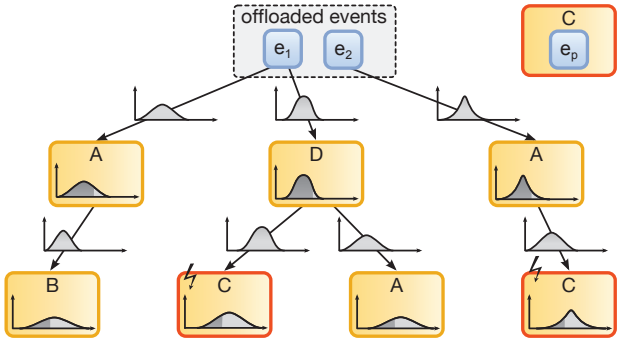


Figure 5: Successor tree constructed by the Local Order Heuristic during the decision making process. The edges show the delay distributions between consecutive event types on different modules. Each node contains the sum of the delay distributions along the path from the root to itself. Aggregating the probabilities of all paths to all conflicting nodes (module C) gives the final conflict probability.

the delay 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

ii) the difference in simulated time between e and e_s.

The latter constitutes the sampling 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.

Upon a request regarding a pending event e_p, the heuristic constructs a successor tree as follows (see Figure 5): First, it adds all currently offloaded events in O to the tree, forming a virtual root node. Second, it appends a new node to the tree for each event type succeeding the events in O. Third, the heuristic traverses the tree in a breadth-first manner and adds for all existing nodes the respective successor event type nodes. Within this tree, we denote the set of nodes which represent event types occurring on the same module as e_p, *conflicting nodes*. Furthermore, the edges between nodes contain the delay distributions between event types, while each node contains the *sum* of the delay distributions on the path from the root node to itself. Using the aggregated delay distributions in every conflicting node, the heuristic can determine the probability that an event creates a conflicting event e_c with $t(e_c) < t(e_\mathrm{p})$. Computing an exact value for p_c, however, requires constructing the complete tree in order to find all conflicting nodes. Instead, we iteratively compute lower and upper bounds $(p_\mathrm{l}, p_\mathrm{u})$ 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 leafs $L \subseteq T$ in the tree

$$p_\mathrm{u} = 1 - \prod_{\substack{\tau \in L, e \in O, \\ \mathrm{path}(e,\tau)}} \Big(1 - p(\tau, d_e)\Big),$$

where $\mathrm{path}(e, \tau)$ denotes the existence of a unique path in the tree from an event e in the root to the leaf node τ. 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 lower bound p_l is determined analogously to p_u, but solely based on the conflicting nodes among the leafs. This probability only increases when encountering new conflicting nodes which contribute a delay distribution with a larger $p(\tau, d_e)$.

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 convolute two delay distributions in order to compute their sum. The complexity of such a convolution is $\mathcal{O}\big(|B|^2 \cdot \mathrm{ld}(|B|)\big)$ due to pairwise combining all buckets of both input histograms and sorting the resulting $|B|^2$ buckets. For a tree with $|T|$ nodes, we hence determine an overall complexity of $\mathcal{O}\big(|T| \cdot |B|^2 \cdot \mathrm{ld}(|B|)\big)$ 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 $|T| \gg |O|$. Nevertheless, the complexity for recording a sample remains at $\mathcal{O}\big(\mathrm{ld}(|B|)\big)$, 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 $\mathcal{O}\big(|B|\big)$ if one delay is static or even down to $\mathcal{O}\big(1\big)$ if both delays are static.

Distribution Function Cut-off. Figure 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.

5. DISCUSSION

The histogram-based learning process assumes that communication patterns between modules follow fixed distributions with steady parametrization. Based on this assumption, we simply add new samples to the existing histograms during the course of a simulation run. Consequently, this simplified 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.

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 [27]. In addition, our probabilistic event scheduling scheme could be nicely complemented with a probabilistic checkpointing mechanism [21] to drastically improve the performance over our simple periodic checkpointing mechnism. In general, the contribution of this paper 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.

6. EVALUATION

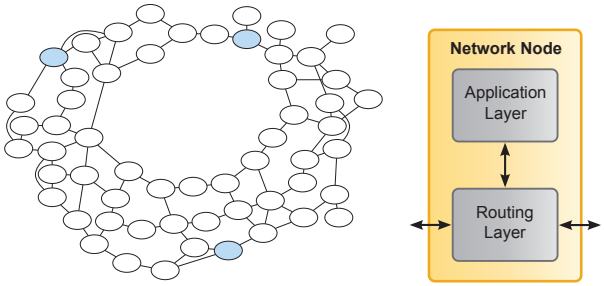
In this section, we evaluate the probabilistic synchronization scheme and the three heuristics in terms of prediction quality, overhead, and performance gain. The evaluation conceptually consists of two parts. First, we evaluate the accuracy and overhead of the three heuristics by means of a simple evaluation 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 realistic 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. Our implementation bases on HORIZON for OMNeT++ 3.3 [26]. The integration of the synchronization scheme with HORIZON for OMNeT++ 4.x is currently ongoing. Each data point shows the mean and the 99 % confidence interval over 30 independent runs.

6.1 Synthetic Benchmarks

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

6.1.1 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 6(a)). Each node consists of an application module sending data packets and receiving



(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 6: Components of the synthetic evaluation model.

acknowledgments, and a routing module, forwarding incoming packets towards the destination (see Figure 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. We express the degree of optimism of a heuristic in terms of the *Positive Rate (PR)*:

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

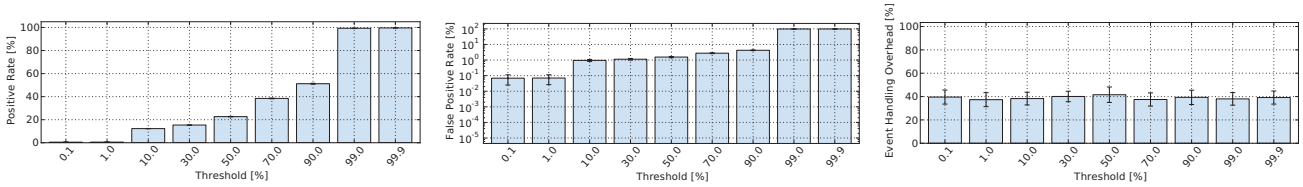
A PR of 0 means purely conservative synchronization while a PR of 1 corresponds to completely optimistic synchronization. Furthermore, we measure the prediction quality by means of the *False Positive Rate (FPR)* which is the ratio of actually false positive decisions to *all possible* false positive decisions:

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

Again, conservative synchronization always yields an FPR of 0 while the FPR in optimistic synchronization is 1. We vary the 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.

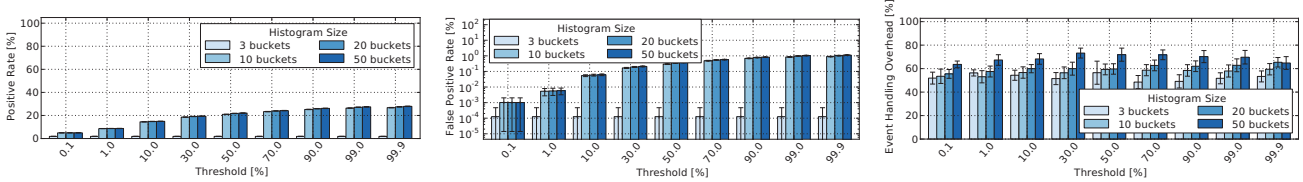
6.1.2 Arrival Pattern Heuristic

Figure 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 7(b)). Thus, de-



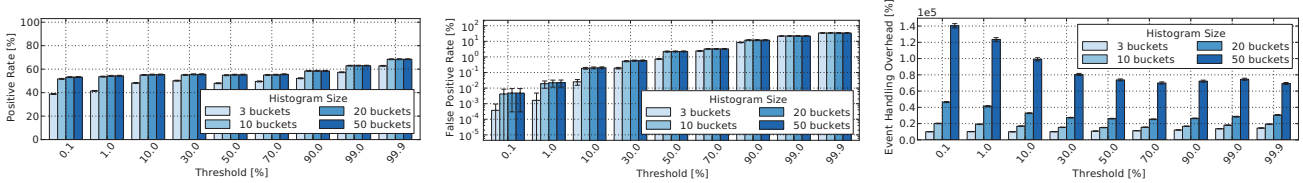
(a) Degree of optimism: Positive rate. (b) Quality: False positive rate. (c) Event handling overhead.

Figure 7: Degree of optimism, prediction quality, and overhead of the Arrival Pattern Heuristic.



(a) Degree of optimism: Positive rate. (b) Quality: False positive rate. (c) Event handling overhead.

Figure 8: Degree of optimism, prediction quality, and overhead of the Global Order Heuristic.



(a) Degree of optimism: Positive rate. (b) Quality: False positive rate. (c) Event handling overhead.

Figure 9: Degree of optimism, prediction quality, and overhead of the Local Order Heuristic.

spite its simplicity, the Arrival Pattern Heuristic is able to predict the next incoming event type with reasonable accuracy. Figure 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.

6.1.3 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 8(a) and 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 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.

6.1.4 Local Order Heuristic

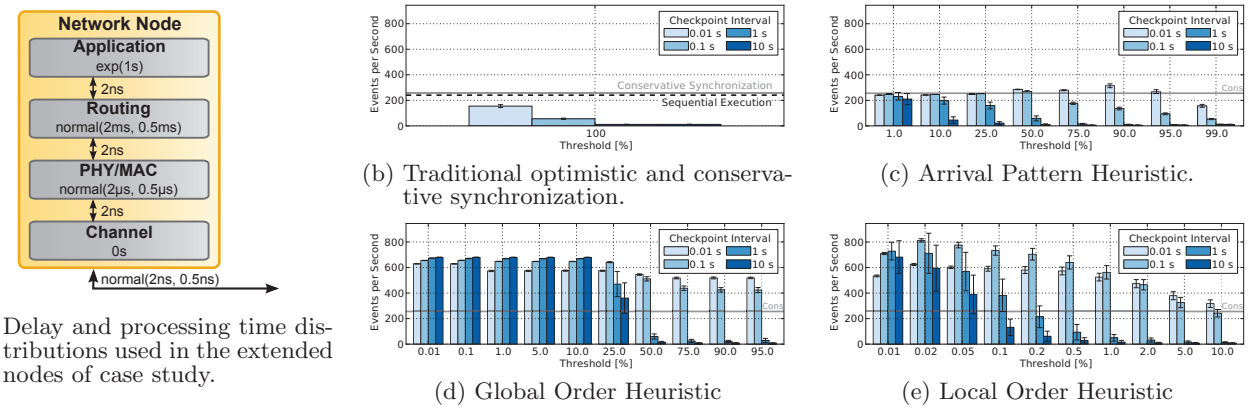
The PR of the Local Order Heuristic exhibits a relatively constant value of approx. 50 % for threshold values ranging from 0.01 % up to 90 %, before finally increasing to 70 % for extremely large thresholds (Figure 9(a)). Thus, already for very small thresholds, the heuristic offloads every second event. 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 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, 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 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.

6.2 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.

6.2.1 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 broad-



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

(b) Traditional optimistic and conservative synchronization.

(c) Arrival Pattern Heuristic.

(d) Global Order Heuristic

(e) Local Order Heuristic

Figure 10: Performance of the traditional synchronization techniques and our heuristics in the case study.

cast 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. For brevity, we do not discuss these models here, but refer to work by Puñal et al. [19]. Furthermore, the model abstracts from a concrete MAC scheme, interference, and transport layer protocols. Figure 10(a) 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.

6.2.2 Traditional Techniques

Figure 10(b) 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 10(a)). 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.

6.2.3 Arrival Pattern Heuristic

The Arrival Pattern Heuristic performs generally worse than conservative synchronization (see Figure 10(c)). 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.

6.2.4 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 10(d)). 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 accredit 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.

6.2.5 Local Order Heuristic

Finally, Figure 10(e) 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.

6.3 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 11.

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 11, 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 sampling data. However, with increasing sampling data, 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, after about 3 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 sampling 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.

7. RELATED WORK

The community has developed a plethora of optimizations for the two basic synchronization paradigms. Due to space constraints, we can only focus on closely related efforts and refer the reader to an extensive survey [17] for further details.

Probabilistic Synchronization. In their pioneering work on probabilistic synchronization, Ferscha et al. [5, 6] analyze the arrival patterns of events. Using statistical methods 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. [24] construct PDFs from the time differences of committed events at each Logical Process (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.

Dynamic 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. [4] 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. [13, 14] extend this work by modeling the data flow *between* components of a simulation model in a dependency graph. In contrast to our work, the construction of the data flow graph requires manual specification of paths instead of automatic learning. Similar efforts focus on deriving larger lookaheads from domain specific model properties. For instance, Liu et al. [12] extract extended lookaheads from wireless networks based on packet air times and multi-hop propagation. Furthermore, Chung et al. [3] 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.

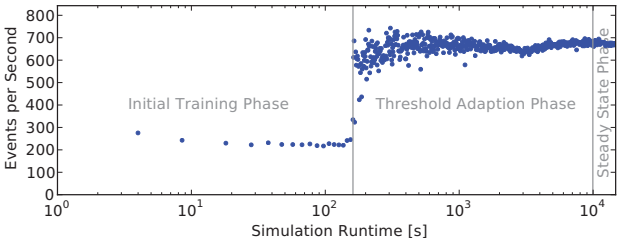


Figure 11: In a probabilistic synchronization scheme simulation runs consist of three phases.

Advanced Synchronization Approaches. Nicol et al. propose the concept of composite synchronization [15] 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 the domain of optimistic synchronization, reverse computation [2] constitutes an approach to mitigate the complexity of state saving. Instead of saving and reloading the values of state variables, a rollback consists of an inverse computation of the previously and erroneously executed events. Hence, as opposed to probabilistic synchronization, reverse computation does not intend to reduce the number of rollbacks, but merely their overhead. In addition to those approaches, combined synchronization [10] 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 [1], the High Level Architecture [8], and µsik [16]. Although those frameworks support dynamic switching between conservative and optimistic schemes, each scheme is applied to a whole LP or even a group of LPs [22]. In contrast, our approach dynamically decides for each event individually whether conservative or optimistic execution is the most fa-

vorable option to maximize performance. It hence allows exploiting differences in workload and timing on a much finer scale. Quaglia [20] proposes a scheduling algorithm for optimistic synchronization that considers the state of neighboring LPs in addition to the local state of each LP. As opposed to our work, this approach however does not attempt to collect long-term state information, but bases its scheduling decisions on the recent state of the LPs. Finally, relaxed synchronization [9] 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.

8. CONCLUSION AND FUTURE WORK

We presented a probabilistic synchronization scheme that learns the runtime behavior of a simulation to allow for educated decisions on speculative parallel event execution. Its core component is a heuristic that continuously collects event scheduling information at runtime. Based on this information, it computes the probability for inflicting a causal violation when speculatively executing events. Since such a heuristic adds considerable overhead to the simulation framework, we developed three different heuristics that trade off prediction accuracy for 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.

Future efforts focus primarily on extending our synchronization scheme with an automatic configuration and performance tuning component. Motivated by the large differences in performance between i) the heuristics themselves and ii) different parametrizations of the heuristics, this configuration component should automatically adjust configuration parameters, such as the threshold and the checkpointing interval. Secondary goals include reducing the runtime overhead by caching of final or intermediate prediction results. Lastly, idling worker threads can pre-compute prediction operations by convoluting distribution functions ahead of time.

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.

9. REFERENCES

[1] R. Bagrodia and W.-T. Liao. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Trans. on Software Engineering*, 20(4):225–238, 1994.

[2] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient Optimistic Parallel Simulations using Reverse Computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, July 1999.

[3] M.-K. Chung and C.-M. Kyung. Improving Lookahead in Parallel Multiprocessor Simulation Using Dynamic Execution Path Prediction. In *Proc. of the 20th Workshop on Principles of Advanced and Distributed Sim.*, 2006.

[4] B. Cota and R. Sargent. A Framework for Automatic Lookahead Computation in Conservative Distributed Simulations. In *Proc. of the SCS Multiconference on Distributed Simulation*, 1990.

[5] A. Ferscha. Probabilistic Adaptive Direct Optimism Control in Time Warp. In *Proc. of the 9th Workshop on Parallel and Distributed Simulation*, 1995.

[6] A. Ferscha and G. Chiola. Self-adaptive Logical Processes: The Probabilistic Distributed Simulation Protocol. In *Proc. of the 27th Annual Simulation Symposium*, 1994.

[7] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10), 1990.

[8] R. M. Fujimoto. Time Management in the High Level Architecture. *Trans of The Society for Modeling and Simulation International*, 71(6):388–400, 1998.

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

[10] V. Jha and R. L. Bagrodia. A Unified Framework for Conservative and Optimistic Distributed Simulation. In *Proc. of the 8th Workshop on Parallel and Distributed Simulation*, 1994.

[11] 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.

[12] J. Liu and D. M. Nicol. Lookahead Revisited in Wireless Network Simulations. In *Proc. of the 16th Workshop on Parallel and Distributed Simulation*, 2002.

[13] R. A. Meyer and R. L. Bagrodia. Improving Lookahead in Parallel Wireless Network Simulation. In *Proc. of the 6th Intern. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1998.

[14] R. A. Meyer and R. L. Bagrodia. Path Lookahead: A Data Flow View of PDES Models. In *Proc. of the 13th Workshop on Parallel and Distributed Simulation*, 1999.

[15] D. M. Nicol and J. Liu. Composite Synchronization in Parallel Discrete-Event Simulation. *IEEE Trans. Parallel Distrib. Syst.*, 13(5):433–446, May 2002.

[16] K. S. Perumalla. μsik – A Micro-Kernel for Parallel/Distributed Simulation Systems. In *Proc. of the 19th Workshop on Principles of Advanced and Distributed Simulation*, 2005.

[17] K. S. Perumalla. Parallel and Distributed Simulation: Traditional Techniques and Recent Advances. In *Proc. of the 38th Winter Simulation Conference*, 2006.

[18] 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.

[19] O. Puñal, H. Escudero, and J. Gross. Performance Comparison of Loading Algorithms for 80 MHz IEEE 802.11 WLANs. In *Proc. of the 73rd IEEE Vehicular Technology Conference*, 2011.

[20] F. Quaglia. A State-based Scheduling Algorithm for Time Warp Synchronization. In *Proc. of the 33rd Annual Simulation Symposium*, 2000.

[21] F. Quaglia. A Cost Model for Selecting Checkpoint Positions in Time Warp Parallel Simulation. *IEEE Trans. Parallel Distrib. Syst.*, 12:346–362, Apr. 2001.

[22] H. Rajaei, R. Ayani, and L.-E. Thorelli. The Local Time Warp Approach to Parallel Simulation. In *Proc. of the 7th Workshop on Parallel and Distributed Simulation*, 1993.

[23] G. Seguin. Multi-core Parallelism for ns-3 Simulator. Technical report, INRIA Sophia-Antipolis, 2009.

[24] T. K. Som and R. G. Sargent. A Probabilistic Event Scheduling Policy for Optimistic Parallel Discrete Event Simulation. In *Proc. of the 12th Workshop on Parallel and Distributed Simulation*, 1998.

[25] S. Turner and M. Xu. Performance Evaluation of the Bounded Time Warp Algorithm. In *Proc. of the 6th Workshop on Parallel and Distributed Simulation*, 1992.

[26] A. Varga. The OMNeT++ Discrete Event Simulation System. In *Proc. of the European Simulation Multiconference*, 2001.

[27] R. Vitali, A. Pellegrini, and F. Quaglia. Autonomic Log/Restore for Advanced Optimistic Simulation Systems. In *Proc. of the 18th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.