Runtime Efficient Event Scheduling in Multi-threaded Network Simulation

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

Developing an efficient parallel simulation framework for multiprocessor systems is hard. A primary concern is the considerable amount of parallelization overhead imposed on the event handling routines of the simulator. Besides complex event scheduling algorithms, the main sources of overhead are thread synchronization and locking of shared data. Thus, compared to sequential simulation, the overhead of parallelization may easily outweigh its performance benefits.

We introduce two efficient event handling schemes based on our parallel-simulation extension Horizon for OMNeT++. First, we present a *push-based event handling scheme* to minimize the overhead of thread synchronization and locking. Second, we complement this scheme with a novel *event scheduling algorithm* that significantly reduces the overhead of parallel event scheduling. Lastly, we prove the correctness of the scheduling algorithm. Our evaluation reveals a total reduction of the event handling overhead of up to 16x.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Performance

Keywords

Parallel Network Simulation, Multi-threading, Performance Optimization

1. INTRODUCTION

Network simulation models are steadily increasing in complexity. For instance, simulation models of wireless networks typically employ highly detailed link layer and physical layer models to allow for an accurate evaluation of advanced wireless transmission technologies. Similarly, large scale peer-to-peer or Internet backbone networks demand appropriately sized topologies to capture the behavioral characteristics of those networks. Such model complexity regularly results in extensive simulation runtimes which in turn may hamper thorough evaluations due to excessive time demands.

Parallel network simulation proved to be able to counteract this effect and significantly cut simulation runtimes [4, 6, 17]. Particularly the recent proliferation of multiprocessor hardware has once again brought parallel network simulation back in the focus of research [1, 3, 15, 18]. Still, developing an efficient parallel simulation framework that provides satisfying speedup is challenging. Parallel event handling typically imposes an increased overhead on the event handling routines of a parallel simulation framework due to more complex event scheduling schemes and/or locking of shared data structures. As a result, we face a potential dilemma: We want to gain performance through parallelization, but this comes at the price of increased event handling overhead. The latter has a specifically negative impact on simulation models whose individual events exhibit only small computational complexity such as peer-to-peer networks, for example. In those models, the ratio of event handling overhead to useful event processing is particularly disadvantageous.

We address this dilemma from an implementation perspective. Based on our parallel simulation extension Horizon [9], we present optimized implementations that slash the event handling overhead. Horizon builds on top of the OMNeT++ simulator [19] and enables parallel event execution on shared memory multiprocessor systems. In Horizon, discrete events are expanded to span a period of simulated time. During simulation, overlapping expanded events are offloaded by a centralized event scheduler to a worker thread for parallel processing. Aiming at a minimum offloading overhead, we make the following contributions:

i) We present a *push-based event handling* scheme that reduces the offloading delay, i.e., the time between offloading and actually processing an event, by enabling the event scheduler to explicitly and directly assign events to worker threads.

ii) We introduce a simplified *event scheduling algorithm* that eliminates the need for "barrier events" to indicate the end of expanded events. As a result, the algorithm removes 50% of the total number of events and the corresponding overhead. Additionally, we prove the correctness of the algorithm.

Our evaluation shows that each of our optimizations yields a considerable performance improvement over an initial prototype implementation of Horizon. In combination, both optimizations reduce the event scheduling overhead by a factor of up to 16.

OMNeT++ 2011 March 21–25, Barcelona, Spain.

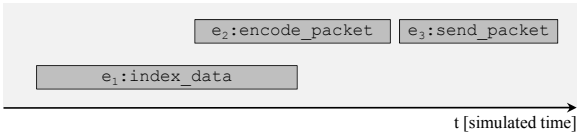


Figure 1: The extended events e_1 and e_2 cannot depend on each other due to their temporal overlapping and can thus be executed in parallel. e_3 must follow sequentially since it might depend on e_1 or e_2.

The remainder of the paper is structured as follows. Section 2 introduces the parallelization concept underlying Horizon followed by a detailed problem analysis in Section 3. Based on this analysis, we present our optimizations in Section 4 and evaluate their performance in Section 5. Finally, we discuss related efforts in Section 6 before concluding the paper in Section 7.

2. 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 methodology for extending 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 [4]. 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 reasoning on pioneering works by Lubachevsky [11] and Fujimoto [5].

Horizon employs a centralized event scheduling architecture specifically designed for multiprocessor hardware. In contrast to related parallelization frameworks, 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 other events and finally offloads it to a worker thread for parallel execution. We demonstrated the viability of our approach in previous work [9]. In this paper, we focus on the challenges of implementing runtime efficient event handling and thread synchronization algorithms.

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

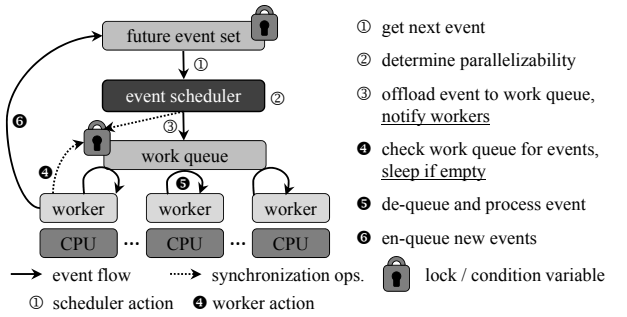


Figure 2: Event handling and synchronization operations in a straightforward, pull-based scheme. Concurrent access to the work queue is coordinated by classic locks and condition variables.

3. ANALYZING THE EVENT OVERHEAD

The centralized event scheduling approach of Horizon exhibits two primary advantages in comparison to the classic distributed event handling of parallel simulators. First, it drastically reduces the overhead incurred by distributed synchronization algorithms [2, 4, 14]. Instead, all required information for deriving a safe event scheduling is readily available in one place (the event queue) and processed by one entity (the event scheduler). Second, it allows for an even distribution of work load across CPUs without the need for specifically designed load balancing algorithms which again impose overhead. In contrast, Horizon follows a thread-pool approach in which a set of worker threads dynamically handle events marked for parallel execution.

However, the parallelization scheme of Horizon causes two specific kinds of event scheduling overhead: *Event handling overhead* and *event synchronization overhead*. In the following, we discuss both kinds of overheads in detail:

Event handling overhead. A straightforward implementation of the centralized event handling scheme of Horizon is illustrated in Figure 2. In this scheme, the scheduler buffers all events eligible for parallel processing in a work queue which is regularly checked by the worker threads. Consequently, this queue needs to be protected by locks to prevent data corruption. If the queue is empty or the protecting lock is occupied when checked by a worker, this thread is suspended. Even if a thread is suspended due to a lack of work, its sleeping period is considerably short, hence resulting in frequent suspend and resume operations. While suspending and resuming threads is considered to be resource efficient due to freeing the CPU, this approach generates considerable threading overhead because of a large number of context switches and system calls to the OS kernel. Additionally, it significantly increases the offloading delay, i.e., the time between offloading an event and actually processing it. This is particularly disadvantageous for simulation models that mainly comprise events of short processing times.

Barrier synchronization overhead. Extended events that span a period of simulated time are a fundamental design property of Horizon. Since such events exhibit distinct starting times and completion times, a straightforward integration in the simulation framework builds upon using two dis-

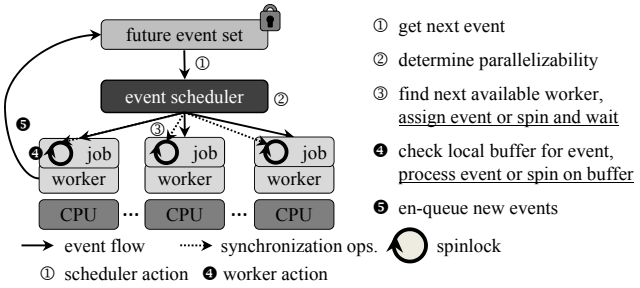


Figure 3: Event handling and synchronization operations in an optimized, push-based scheme. Events are assigned directly to available workers which actively spin on a local buffer.

crete events to represent both extremes of the interval. For each extended event, both discrete events are stored in the central event queue accordingly. The scheduler then continuously removes the first event from the queue which can be of either type: If it represents the start of an extended event, it is handed to the workers, i.e., the event is offloaded and executed in parallel. If it indicates the completion of an extended event, the scheduler potentially needs to block until the associated worker has finished processing this event. Hence, we denote the latter *barrier events*.

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

4. EFFICIENT EVENT SCHEDULING

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

4.1 Cutting Event Handling Overhead

As stated in the previous section, suspending idle threads is considered resource efficient by freeing CPUs for useful work. However, we argue that high-performance parallel simulations typically run on dedicated hardware and thus do not need to free CPUs for other tasks when a worker is blocked. Instead, it is more important to swiftly continue processing events as soon as they become available without the need for waking up a worker thread. Hence, we explicitly trade CPU resources for shorter offloading delays.

Our approach replaces the pull-based event handling algorithm (Figure 2), in which the worker threads pull jobs from the work queue, with a pushed-based scheme (Figure 3). When offloading an event, the central scheduler checks the current processing state of all workers and explicitly assigns the event to an idle one. To this end, each worker thread maintains a local buffer (`job`) that can hold exactly

```
1  simulate(){
2   t_b = ∞
3   O = ∅
4   while(F ∪ O ≠ ∅){
5    e = checkBarrier()
6    O = O ∪ e
7    F = F \ e
8    offloadEvent(e)
9   }
10 }
```

(a) Simulation loop.

```
1  checkBarrier(){
2   do{
3    e = min{t_s(e)|e ∈ F}
4    if t_s(e) ≤ t_b
5     return e
6    else
7     waitFor(w|_{t_b}.job=⊥)
8     t_b = min{t_c(e')|e' ∈ O}
9   }while(true)
10 }
```

(b) Checking the barrier.

```
1  offloadEvent(e){
2   t_b = min{t_c(e')|e' ∈ O}
3   w = one{w ∈ W|w.job=⊥}
4   w.job = e
5  }
6
```

(c) Event assignment.

```
1  worker(){
2   waitFor(job≠⊥)
3   process(job)
4   O = O\job
5   job=⊥
6  }
```

(d) Event processing.

Figure 4: The main building blocks of the event scheduling algorithm. The scheduler (a) continuously checks the current barrier (b) and offloads an event to one of the workers (c) for processing (d).

one event. If the buffer is empty, the scheduler assumes that the worker is currently not processing any event and consequently puts an event into the buffer. Simultaneously, the worker threads poll their local buffer. As soon as a buffer is filled, the corresponding worker starts processing the event and finally empties the buffer again. By means of this busy waiting scheme, workers immediately recognize newly offloaded events and the time between assigning a new event and processing it is reduced to a minimum.

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

4.2 Eliminating Barrier Events

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

Our improved event scheduling algorithm is defined as follows. Let E be the set of all events that occur in a simulation run and $F \subseteq E$ a particular FES. For all events $e \in E$, $t_s : E \to \mathbb{R}$ denotes the starting time in simulated time and $t_c : E \to \mathbb{R}$ the corresponding completion time. Furthermore, $t_b \in \mathbb{R}$ represents the currently active barrier, $O \subseteq E$

is the subset of presently offloaded events, and W is the set of workers. Finally, $\perp$ represents an empty buffer and "one" : $2^W \rightarrow W$ returns an idle worker or blocks until one is available.

At runtime, the scheduler (Figure 4(a)) continuously i) checks the current barrier t_b and ii) offloads independent events to the worker threads. *Checking the barrier* (Figure 4(b)) involves dequeuing the first event e from the event queue and comparing its starting time $t_\mathrm{S}(e)$ to the minimum barrier t_b. If the barrier precedes e, the scheduler blocks at the barrier by waiting for the specific worker which handles the event that is associated to the barrier ($w|_{t_\mathrm{b}}$) to finish. Subsequently, the scheduler determines a new minimum barrier and again compares the first event from the event queue to the new barrier. *Event offloading* (Figure 4(c)) bases on our push-based event handling approach. The scheduler first updates the current barrier if the newly offloaded event e finishes before the established barrier. In this case, the barrier must be moved to $t_\mathrm{C}(e)$ to ensure the correctness of the algorithm. The scheduler then either determines an idle worker to which it assigns e for processing, blocks until a worker becomes available, or processes the event itself. Finally, the last component of the overall algorithm constitutes the workers (Figure 4(d)) which continuously process events according to the push-based event handling scheme introduced previously.

4.3 Correctness of the Scheduling Algorithm

In this section, we present a proof that shows the correctness of the optimized barrier synchronization algorithm. We need to show that the algorithm guarantees causal correctness, i.e., non-independent events are processed only in increasing starting time order. For this purpose, we first introduce a formal definition of event overlapping, followed by a key assumption.

Definition 1. Two extended events e_1, e_2 *overlap* in simulated time (denoted by $e_1 \parallel e_2$) if and only if the duration intervals intersect:

$$e_1 \parallel e_2 \Leftrightarrow [t_\mathrm{S}(e_1); t_\mathrm{C}(e_1)] \cap [t_\mathrm{S}(e_2); t_\mathrm{C}(e_2)] \neq \emptyset.$$

Assumption 1. If two events overlap, both may be executed simultaneously.

Event durations express the fact that processes within a real system typically span a certain processing time. Thus, if two expanded events overlap, their corresponding processes in the real system also overlap and are hence executed in parallel. Based on this observation, we formulate the fundamental modeling paradigm underlying Horizon: Overlapping processes/events must not influence each other and their respective results must become visible to the rest of the system only after they are finished. Following a similar reasoning, we state Definition 2:

Definition 2. An event e can only create new events e' with $t_\mathrm{S}(e') \geq t_\mathrm{C}(e)$.

This means that new events may only start after the event that created them is complete. As pointed out previously, the results of an expanded event may only become visible to the entire system after the event has been processed.

We moreover adopt the definition of causal correctness regarding a parallel simulation run from [4] and modify it according to Assumption 1:

Definition 3. A discrete event simulation obeys the causality constraint if and only if each pair of non-overlapping events is processed in non-decreasing starting time order.

After providing the required basics, we now prove the correctness of the scheduling algorithm: First, we show that no events with starting times preceding the barrier are inserted into F (Lemma 1). By means of this lemma we show that the scheduler handles events in increasing starting time order (Lemma 2). Note that this does not imply that events are executed by the worker threads in increasing starting time order. Then we can show that only overlapping events are executed in parallel (Lemma 3). Applying Lemma 2 and 3 we finally proof the causal correctness property of our event scheduling algorithm.

Lemma 1. No event e with $t_\mathrm{S}(e) < t_\mathrm{b}$ is inserted into F by another event e'.

Proof. Based on Definition 2, no event $e \in O$ may insert another event $e' \in E$ into F with $t_\mathrm{S}(e') < t_\mathrm{C}(e)$. Since t_b is the minimum over the completion times of all offloaded events, i.e., $t_\mathrm{b} = \min\{t_\mathrm{C}(e) | e \in O\}$ (Figure 4(b), Line 8), it follows that $\forall e \in O : e$ may not insert an event e' into F with $t_\mathrm{S}(e') < t_\mathrm{b}$. This property obviously also holds for all $e \notin O$ as a non-offloaded event is not executed and hence cannot create new events. Concluding, no new event preceding t_b can be inserted into F. □

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

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

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

Case 2: $e_2 \in F, e_1 \in F$. Because of the ordering constraint of F and the fact that the scheduler only removes the event with the smallest starting time from F (Figure 4(b), Line 3), it first handles e_1 and then e_2.

Case 3: $e_2 \in F, e_1 \notin F, t_\mathrm{b} < t_\mathrm{S}(e_2)$. The scheduler does not handle e_2, but it blocks at t_b, because the condition $t_\mathrm{S}(e_2) \leq t_\mathrm{b}$ (Figure 4(b), Line 4) is not met.

Case 4: $e_2 \in F, e_1 \notin F, t_\mathrm{b} \geq t_\mathrm{S}(e_2)$. The scheduler selects e_2 for processing. However, e_1 cannot be inserted into F afterwards due to Assumption 2. Thus, e_1 was either processed before or it will never be processed.

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

Lemma 3. The central event scheduler never offloads non-overlapping events.

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

$$
\begin{aligned}
& t_\mathrm{S}(e) \leq t_\mathrm{b} && (1)\\
\Rightarrow\ & t_\mathrm{S}(e) \leq \min\{t_\mathrm{C}(e') | e' \in O\} && (2)\\
\Rightarrow\ & t_\mathrm{S}(e) \leq t_\mathrm{C}(e'),\ \forall e' \in O && (3)\\
\Rightarrow\ & (t_\mathrm{S}(e) \leq t_\mathrm{C}(e')) \wedge (t_\mathrm{S}(e') \leq t_\mathrm{S}(e))),\ \forall e' \in O && (4)\\
\Rightarrow\ & (t_\mathrm{S}(e') \leq t_\mathrm{S}(e) \leq t_\mathrm{C}(e')) \wedge (t_\mathrm{S}(e) \leq t_\mathrm{C}(e)),\ \forall e' \in O && (5)\\
\Rightarrow\ & [t_\mathrm{S}(e); t_\mathrm{C}(e)] \cap [t_\mathrm{S}(e'); t_\mathrm{C}(e')] \neq \emptyset && (6)\\
\Rightarrow\ & e \parallel e' && (7)
\end{aligned}
$$

(1) follows directly from Line 4 of the algorithm in Figure 4(b). Similarly, (2) results from Line 8 in the same listing. (3) is a simple logical conclusion from (2). We further derive from $e' \in O$ that e' was already handled and offloaded before e. By applying Lemma 2, we conclude that e' exhibits a smaller or equal starting time than e, showing (4). The first part of the conjunction in (5) is a reformulation of (4), the second part results from the simple fact that event durations must not be negative. The ordering of the timestamps in (5) shows that the intervals do overlap, resulting in (6). Finally, by applying Definition 1, we conclude that e and e' overlap. □

The above lemmas enable us to show the central theorem:

Theorem 1. The event scheduling algorithm guarantees causal correctness according to Definition 3.

PROOF. Assume two events $e, e' \in E$. We distinguish two different cases:

Case 1: $e \parallel e'$. According to Assumption 1, e and e' are independent and hence cannot violate causal correctness which is only defined for non-overlapping events.

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

Concluding, causal correctness is guaranteed for both overlapping and non-overlapping events. □

5. EVALUATION

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

5.1 Setup and Methodology

We benchmark the event handling overhead by means of a specially designed simulation model for two reasons: i) It is difficult to accurately measure the overhead imposed by thread synchronization primitives such as locks. There is no simple way to determine the time consumed by the function call without also potentially measuring the time a thread was suspended. ii) Event handling operations are split across the scheduler and the workers. Hence, determining the overhead per event as the sum of both does not accurately reflect the performance behavior as observed by the user. Instead, due to the parallelization of the scheduler and the workers, both overheads actually overlap.

As a result, we utilize a "null" simulation model whose events do not perform any computations except for re-inserting themselves in the event queue – which again is event handling overhead. Thus, the null-model exclusively generates overhead which allows us to we derive the overall event handling overhead, including overhead parallelization effects, by measuring the total runtime of this model (without setup and teardown times). In order to provide parallelism for the workers, the model consists of 110 independent modules. Further, the expanded events span a duration of 1s and are timed at fixed 20s intervals 50000 times per module, resulting in a total of 5.5 million events. However, since the null-model does not perform any useful work, the event processing times are exceptionally short, thereby limiting the amount of achievable parallel performance. Consequently, we do not expect a performance increase when adding more workers, but instead a performance degradation due to increased contention.

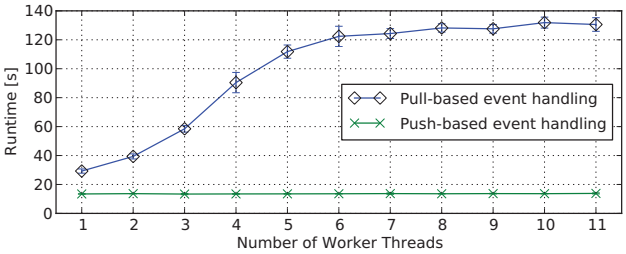
This evaluation of Horizon is based on OMNeT++ 3.3 while a port to OMNeT++ 4.1 is nearly complete. All performance results show average values collected over ten independent runs and the corresponding 99% confidence intervals. We utilized an AMD Opteron compute server providing 32GB of RAM and a total of 12 processing cores, organized in two six-core CPUs running a 64-bit Ubuntu 9.10 server OS.

5.2 Event Handling

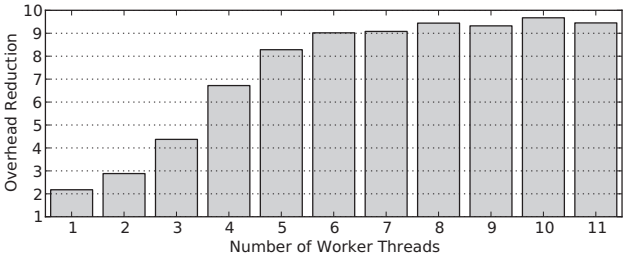
In this section, we compare the pull-based event handling scheme to the push-based one. Figure 5(a) shows the total simulation time for both approaches over a varying number of worker threads. For the pull-based implementation, we observe a linear to super-linear growth in simulation time when using up to six worker threads. As predicted, due to the workload characteristics of the null-model, we do not gain a speedup by adding more workers. Instead, additional worker threads increase the contention on the shared work queue and its synchronization primitives, hence resulting in significantly longer simulation times. When utilizing six up to eleven workers, the total simulation time remains relatively constant. We attribute this to the fact that due to the small amount of workload the additional workers remain mostly suspended, thus limiting the level of contention. In contrast, we observe constant simulation runtimes for the push-based scheme regardless of the number of worker threads. Moreover, the total simulation runtimes are considerably shorter. From this we deduce a significant reduction in the event handling overhead. Figure 5(b) compares the simulation times of both approaches in terms of the overhead reduction, i.e., the speedup factor achieved by our event handling scheme. For a single worker thread, the modified scheme achieves a speedup of 2 and gains a maximum speedup of approximately 9.5 for eight and more workers.

In addition to the computation time, we also measured the amount of context switches during a simulation run. In Figure 5(c), we observe a cutback in the number of context switches by three orders of magnitude. This confirms that the push-based event handling scheme prevents excessive numbers of context switches caused by frequent thread synchronization.

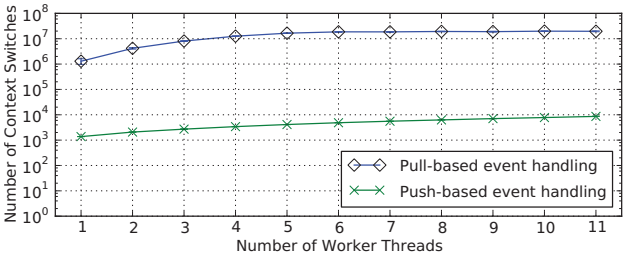
In general, the amount of context switches grows in both schemes with the number of worker threads. However, in the pull-based scheme, the number of context switches stagnates for six or more workers while it still increases notably in the push-based event handling scheme. In order to understand this behavior, we recall the conditions that initiate a context switch: i) The time slice allocated to a thread has expired: If a thread utilizes a CPU for too long, it is suspended by the operating system. ii) Synchronization and I/O operations: A thread voluntarily suspends itself while waiting for a signal from another thread or the completion of an I/O operation. Figure 6(a) clearly illustrates that the number of context switches in the pull-based scheme is dominated by synchronization related context switches as a result of using classic locks. However, the amount of both types of context



(a) Total simulation runtime.



(b) Overhead reduction.



(c) Total number of context switches.

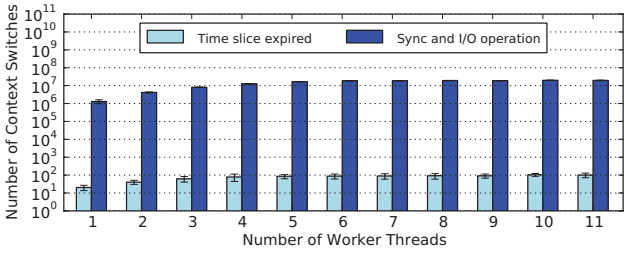
Figure 5: Performance comparison of the pull-based and push-based event handling implementation.

switches stabilizes for six and more workers due to the fact that additional threads remain mostly suspended because of the low level of parallelism achievable by the null-model. In contrast, we observe in Figure 6(b) that in the push-based approach the number of context switches caused by expired time-slices linearly increases while the number of synchronization based context switches levels off for more than six workers. The former suggests that due to busy waiting, the worker threads run until their time slices expire and the operating system enforces a context switch. The remaining synchronization related context switches are caused by barrier events and I/O operations of the simulator.

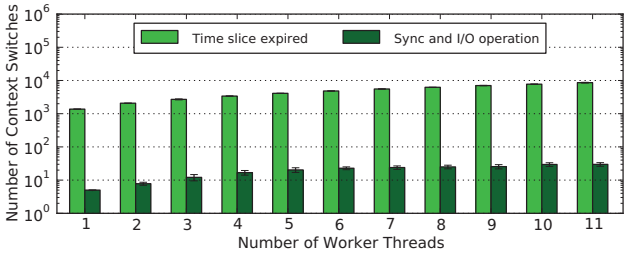
We finally derive from both figures that the relationship between the two types of context switches changes: While the synchronization related context switches dominate the total number of context switches in the pull-based implementation, their number drops to a small fraction in the push-based event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

5.3 Event-free Barriers

In this section, we analyze the performance gain achieved by eliminating barrier events from the push-based event handling scheme. Figure 7(a) illustrates the total simulation runtimes of both implementations. The event-free barrier



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



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

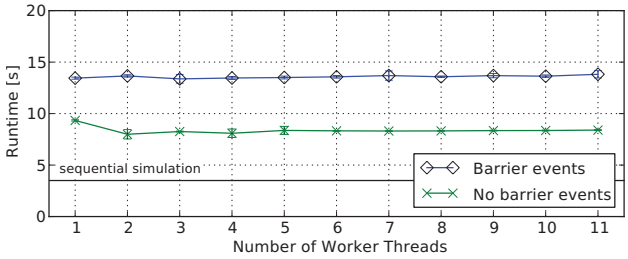
Figure 6: Detailed examination of the type and the number of context switches.

algorithm clearly outperforms the scheduling approach that depends on barrier events. In particular, we observe a drop in the runtime when adding a second worker. We derive from this that the modified event scheduling algorithm allows for a better parallelization of its overhead between the scheduler and the worker threads. Overall, the simulation runtime for the null-model decreases by a factor of 1.4x for one worker thread and up to a factor of 1.6x for two and more workers as shown in Figure 7(b). Considering that the modified algorithm removes 50% of the total number of events, this constitutes a satisfying result. Additionally, the performance improvement is also reflected in the number of context switches as depicted in Figure 7(c). The event-free barrier algorithm generates slightly fewer context switches than the non-optimized version. Still, both algorithms show the same growth in the number of context switches which is dominated by time-slice-related context switches as discussed previously.

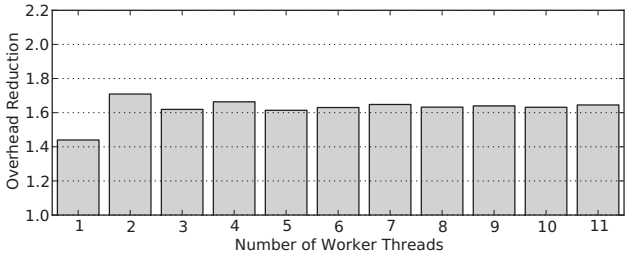
Finally, the straight black line in Figure 7(a) illustrates the runtime of a purely sequential simulation. Hence, the difference between this line and the one of the improved scheduling algorithm reveals the remaining parallelization overhead. However, by efficiently parallelizing a real-world simulation model, we highlight in Section 5.5 that this additional overhead is indeed worth paying for.

5.4 Combined Speedup

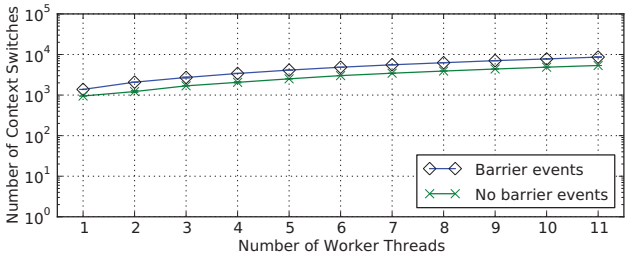
Finally, we briefly evaluate the combined performance improvement of both event handling optimizations. To this end, we compare the performance measurements obtained for the initial prototype implementation to those for the optimized version with explicit event assignment and event-free barriers. Figure 8 shows the corresponding results in terms of the reduction in the event scheduling overhead. In accordance with the previously presented results, the total performance gain ranges between a 3x speedup, obtained for just one worker, up to a peak value of approximately 16x for eight or more workers.



(a) Total simulation runtime.



(b) Overhead reduction.



(c) Total number of context switches.

Figure 7: Performance comparison of the event handling schemes with and without barrier events.

5.5 Case Study: LTE Network Model

All benchmarks up to now focused solely on determining the event handling overhead without evaluating the actual parallel performance of the simulation framework. To fill this gap, we now conduct a brief case study based on a parallelized LTE network simulation model. We demonstrate that i) our parallelization framework indeed achieves a considerable parallel speedup, and ii) the performance improvements presented in this paper are necessary to enable an efficient parallelization of simulation models comprised of events of low computational complexity. For the sake of brevity, we do not introduce the network model in detail here but refer the reader to [13]. The evaluation scenario for this case study consists of a network of ten cells, each containing five mobile stations which handle VoIP calls.

As pointed out before, the events of the null-model do not perform any computations except for re-inserting themselves into the FES. Hence, it does not inflict a real workload on the simulation framework. In contrast, the LTE model used in this study generates a diverse workload as illustrated in Figure 9(a). The histogram shows the distribution of the event processing times, i.e., how many events of a certain computational complexity occur in the selected evaluation scenario. Clearly, the event complexities span four orders of magnitude, ranging from milli- to microseconds. In comparison to [9], the model utilizes a simpler resource allocation algorithm which results in significantly shorter event

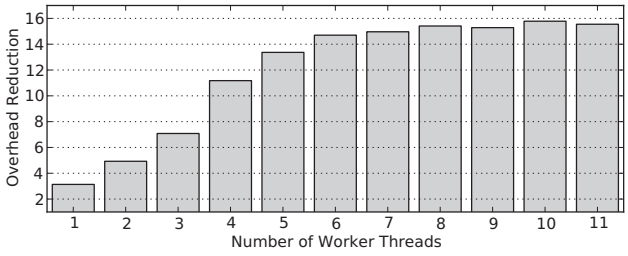
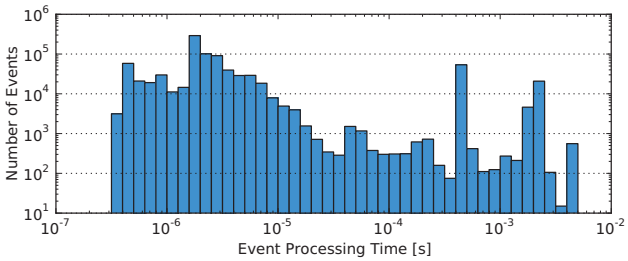
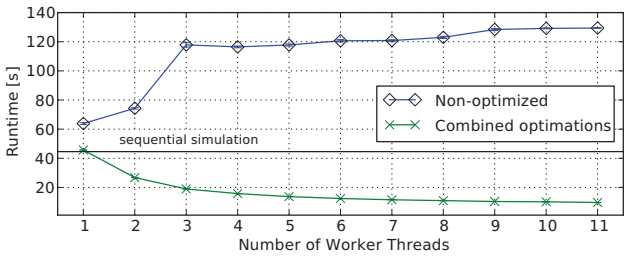


Figure 8: Overhead reduction with combined optimizations.



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



(b) Total simulation runtime of the LTE model.

Figure 9: Performance analysis of a parallel LTE Network Model.

processing times. As a result, the selected scenario is more demanding w.r.t. the efficiency of the simulation framework.

Figure 9(b) plots the runtimes for simulating 1s of network traffic. The graphs show that the non-optimized prototype implementation is not able to generate any speedup, but instead the runtimes increase with the number of workers as seen in the null-model. In contrast, the optimized event handling scheme achieves a maximum speedup of approximately five in the selected scenario. This underlines the viability of our approach as well as the efficiency of its event handling algorithms.

6. RELATED WORK

The recent development towards multiprocessor systems has sparked intensive research in the domain of parallel network simulation. In this section, we present closely related efforts and discuss how our work deviates from previous approaches.

In [15], Peschlow et al. present a multi-threaded parallel simulation framework which primarily focuses on wireless networks. Parallel execution is coordinated by means of a conservative barrier synchronization algorithm that bases on a specifically selected tournament barrier implementation. A similar synchronization approach is taken by a multi-threaded extension of the ns-3 simulator [7, 18]. In the context of these works, thorough performance evalua-

tions of the used thread synchronization primitives revealed a poor performance of the conventional pthread barriers. Consequently, the corresponding high-performance implementations base on specially designed spinlocks. In contrast to Horizon, the barriers in both simulators achieve a global synchronization of all threads whereas Horizon utilizes barriers for blocking only the central event scheduling thread.

In order to avoid locking mechanisms altogether, Liu et al. present a lock-free event scheduling algorithm for parallel simulations on shared memory machines [10]. The algorithm makes use of atomic fetch&add operations to asynchronously determine safe time bounds for conservative event synchronization. In comparison to lock-based implementations, the asynchronous algorithm avoids unnecessary blocking caused by slow threads and thus reduces the number of context switches significantly. As a result, the authors report considerable speedups in comparison to an equivalent lock-based algorithm. A major disadvantage of the proposed algorithm, however, is its complexity which causes extensive overhead for large numbers of threads.

7. CONCLUSION AND FUTURE WORK

In this paper, we present two optimizations to reduce the event handling overhead of our parallel simulation extension Horizon for the OMNeT++ simulator. The first optimization replaces the classic pull-based event processing scheme with a push-based one, in which the event scheduler explicitly assigns events to worker threads. In combination with actively spinning worker threads, this approach results in a significant overhead reduction of a factor of up to 9.5. The second optimization eliminates the need for barrier events to represent the end of event durations. Instead, a lightweight scheduling algorithm continuously determines the relevant barrier – thereby reducing the scheduling overhead by a factor of up to 1.6. We furthermore prove that this algorithm guarantees causal correctness throughout parallel execution. Finally, by combining both optimizations, we yield an overhead reduction of up to a factor of 16 in comparison to our initial implementation.

Motivated by the promising results of our work, we continue to improve our implementation. Future work primarily addresses the reduction of cache misses to further increase performance. Currently, the event scheduler assigns events to the next available CPU without considering on which CPU previous events of the same type were executed. This may result in similar events continuously "moving" across all CPUs, hence preventing efficient cache re-use. Instead, the scheduler should keep track of previous event-to-CPU mappings and aim to re-assign subsequent events accordingly. Additional efforts target the locking mechanism which protects the central event queue. The implementation underlying the evaluation presented in this paper utilizes a relatively simple TTAS (test-test-and-set) spinlock [8]. It is however well known that this kind of lock does not scale well with an increasing number of threads due to severe cache contention. To provide better scalability particularly on ccNUMA architectures, we intend to implement more sophisticated hierarchical spinlocks such as hierarchical backoff locks or the HCLH lock [12, 16].

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.

8. REFERENCES

[1] 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.

[2] K. M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, September 1979.

[3] S. De Munck, K. Vanmechelen, and J. Broeckhove. Design and Performance Evaluation of a Conservative Parallel Discrete Event Core for GES. In *Proc. of the 3rd International Conference on Simulation Tools and Techniques*, 2010.

[4] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.

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

[6] 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.

[7] T. R. Henderson, S. Roy, S. Floyd, and G. F. Riley. ns-3 Project Goals. In *Proc. of the 2006 Workshop on ns-2: The IP network simulator*, 2006.

[8] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient Synchronization of Multiprocessors with Shared Memory. In *Proc. of the 5th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1986.

[9] 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.

[10] J. Liu, D. M. Nicol, and K. Tan. Lock-free Scheduling of Logical Processes in Parallel Simulation. In *Proc. of the 15th Workshop on Parallel and Distributed Simulation*, 2001.

[11] 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.

[12] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Proceedings of the European Conference on Parallel Computing (EuroPar)*, 2006.

[13] F. Naghibi and J. Gross. How Bad is Interference in IEEE 802.16e Systems? In *Proceedings of the 16th European Wireless Conference (EW)*, 2010.

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

[15] 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.

[16] Z. Radovic and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proc. of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 241–252, 2003.

[17] G. F. Riley, M. H. Ammar, R. M. Fujimoto, A. Park, K. Perumalla, and D. Xu. A Federated Approach to Distributed Network Simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 14:116–148, April 2004.

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

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