Parallel Simulation of Queueing Petri Nets

Jürgen Walter
University of Würzburg
97074 Würzburg Germany
juergen.walter@uni-
wuerzburg.de

Simon Spinner
University of Würzburg
97074 Würzburg Germany
simon.spinner@uni-
wuerzburg.de

Samuel Kounev
University of Würzburg
97074 Würzburg Germany
samuel.kounev@uni-
wuerzburg.de

ABSTRACT

Queueing Petri Nets (QPNs) are a powerful formalism to model the performance of software systems. Such models can be solved using analytical or simulation techniques. Analytical techniques suffer from scalability issues, whereas simulation techniques often require very long simulation runs. Existing simulation techniques for QPNs are strictly sequential and cannot exploit the parallelism provided by modern multi-core processors. In this paper, we present an approach to parallel discrete-event simulation of QPNs using a conservative synchronization algorithm. We consider the spatial decomposition of QPNs as well as the lookahead calculation for different scheduling strategies. Additionally, we propose techniques to reduce the synchronization overhead when simulating performance models describing systems with open workloads. The approach is evaluated in three case studies using performance models of real-world software systems. We observe speedups between 1.9 and 2.5 for these case studies. We also assessed the maximum speedup that can be achieved with our approach using synthetic models.

Keywords

Parallel discrete event simulation, performance prediction, stochastic performance modeling, queueing petri nets

1. INTRODUCTION

Queueing Petri Nets (QPNs) are a combination of Colored Generalized Stochastic Petri Nets (CGSPNs) [6] and Queueing Networks (QNs) [20]. In [23], the authors show the benefits of using QPNs in terms of modeling power and expressiveness when analyzing the performance behavior of software systems. In contrast to traditional QNs and CGSPNs, QPNs enable the description of hardware and software aspects of system behavior in the same model [23]. Software contention effects, such as synchronization, simultaneous resource possession and blocking, can be easily described using QPNs [23]. In recent years, QPNs have been successfully used to model the performance of different types of software systems (e.g., component-based systems, event-based systems, database systems, computer networks, or multi-tenant systems [22, 38, 34, 37, 39], see [1] for more). The respective case studies use QPN models for the quantitative analysis of the performance and the scalability of a system under test. QPN models can be analyzed quantitatively using analytical or simulation-based solution techniques. Analytical solution techniques for QPNs are based on transformations to Markov chains. However, models of realistic software systems often result in Markov chains with a too large state space to be analytically tractable [23]. In contrast, simulation techniques provide better scalability and thus enable the analysis of models that could not be solved analytically. SimQPN, which is part of the Queueing Petri Net Modeling Environment (QPME) [42, 24], is the only discrete-event simulator for QPNs currently available. However, the implementation of SimQPN is strictly sequential, limiting its performance on modern multi-core computer systems.

Different approaches to parallel discrete-event simulation have been proposed in the literature to leverage the potential speedup of modern multi-core computer systems [15]. While the general challenges of parallelizing a discrete-event simulation are well understood, the actual speedup heavily depends on the characteristics of the used modeling formalism and the concrete model. So far, only the authors of [18] have evaluated the potential speedup for QPNs using parallel event-discrete simulation. However, this work does not consider queueing places and their associated scheduling strategies, which are an integral part of QPNs (e.g., to model hardware contention effects in a system). The choice of scheduling strategies can severely impact the parallelization potential of discrete-event simulation, as shown in [27] for traditional QNs.

Novel use cases of QPNs identified in the last few years require substantial improvements in simulation speed. The first scenario is *online performance prediction* for run-time resource management. Many software systems are subject to Service Level Agreements (SLAs) regarding response time or throughput. In order to proactively reconfigure a system before SLAs are violated, it is necessary to be able to predict the system performance under different configurations and workloads. QPNs provide the modeling power and expressiveness required for such performance predictions. In online performance prediction scenarios [8], the time for solving QPNs models is a limiting factor in order to be able to react before SLAs are violated. Speedups through parallel simu-

---

[1] https://go.uni-wuerzburg.de/qpnbibliography

lation would enable the timely analysis of larger and more accurate models. The second scenario is the use of QPNs to analyze design-oriented *architecture-level performance models*, which describe a system with a high level of detail. In [28], the authors propose a transformation from the Palladio Component Model (PCM) [7] to QPNs for performance analysis. Applied to practical PCM models, the transformation often results in large and complex QPNs models [28]. A significant speedup of simulation speed is especially worthwhile for such models. Given that the number of cores of modern processors is continuously increasing while the clock frequency stays the same, parallel simulation techniques can provide the speedup required by the previously described usage scenarios for QPNs. In this paper, we show how parallel simulation techniques can be applied to analyze QPNs and evaluate the potential speedup of QPN simulation through parallelization. More specifically, we make the following contributions:

1. We explore the potential of QPNs for parallel event-discrete simulation, which includes the analysis of lookahead estimation for QPNs.

2. We propose a Directed Acyclic Graph (DAG) decomposition approach for QPNs required for efficient parallel discrete-event simulation.

3. We provide an implementation of a parallel discrete-event simulation engine for QPNs based on the SimQPN simulation engine.

The evaluation of our approach is focused on the following two questions: 1) What speedup can be achieved by using discrete-event simulation for QPN models of realistic systems? 2) What is the maximum speedup that can be practically reached? To answer the first question we conducted three case studies using performance models of real-world systems: a model of a large Software-as-a-Service (SaaS) provider, a model of a complex J2EE application, and a model describing the performance behavior of a data-center network. We observed speedups between 1.91 and 2.45 for the considered models. To answer the second question, we used a synthetic model to systematically benchmark the parallel simulation. With the synthetic model, we could even observe super-linear speedups.

The remainder of this paper is organized as follows. We start with a brief introduction to the QPN formalism in Section 2 and parallel simulation in Section 3. Section 4 discusses challenges for parallelization of discrete-event QPN simulation and presents the parallelization approach including our assumptions and design decisions. Section 5 describes the case studies. Section 6 is about related work. Section 7 concludes the paper.

2. QUEUEING PETRI NETS

Queueing Petri Nets (QPNs) are an extension of Colored Generalized Stochastic Petri Nets (CGSPNs) [24], a special type of Petri Nets (PNs). PNs are a general formalism for the analysis of concurrent systems. In mathematical terms, a PN is a directed bipartite graph with two different types of nodes named *places* and *transitions*. Places represent system states and can contain a number of *tokens*. Tokens can be used to represent resource availability, jobs to be executed, flow of control or synchronization conditions [32].

The *initial marking* can be changed by the *firing* of transitions. Transitions are connected to input and output places with forward and backward *incidence functions*. The firing of a transition changes the system state by removing one token from each input place and adding one token to each output place. While PNs are easy to understand and have a rigorous mathematical foundation, PNs of realistic systems quickly become very large. Therefore, extensions to PNs have been proposed to improve its model expressiveness.

Jensen [17] describes Colored Petri Nets (CPNs) introducing token colors which can be used to distinguish between different types of requests in a system. Firing modes can be defined depending on token colors. A color function C assigns the different firing modes to transitions. The greater modeling convenience comes along with no loss of PN properties since CPNs are a backward-compatible extension to the original PNs. In addition to immediate transitions, Generalized Stochastic Petri Nets (GSPNs) introduce *timed transitions* enabling the integration of temporal aspects by assigning firing delays to transitions. The firing delay specifies the time an enabled transition waits before it fires. If multiple timed transitions are enabled at the same time, the next transition to fire is chosen based on *firing weights* (probabilities) assigned to the transitions [26]. The combination of CPNs and GSPNs results in CGSPNs [6]. CGSPNs are a powerful modeling formalism for describing concurrency and synchronization aspects. However, CGSPNs do not support a direct representation of scheduling behaviors [4, 25] and therefore are limited when modeling hardware contention [21]. The Queueing Petri Net (QPN) formalism combines the advantages of QNs and PNs by introducing *queueing places*. A token that arrives at a queueing place is first served in a queue (see QNs), before it is put in the depository. Only tokens in the depository are available to outgoing transitions. Queueing places enable the description of scheduling aspects in QPNs. QPNs are suited for quantitative, as well as qualitative analyses. In this paper, we focus on the quantitative analysis of QPNs required for the performance analysis of computer systems. The HiQPN tool [5] enables the analytical solution of QPNs. However, it is severely limited by the model size that can be solved [23]. SimQPN [24, 42] is a discrete-event simulator for QPNs. SimQPN supports different methods for steady-state analysis (batch means, replication deletion, and method of Welch) and provides great flexibility in controlling the amount of quantitative data (e.g., utilization, throughput, and response time) collected for each place in a QPNs. SimQPN has been shown to scale well with increasing modeling size [28, 8]. However, SimQPN is currently a strictly sequential program and cannot utilize the parallelism offered by modern systems.

3. PARALLEL SIMULATION

QPN actions such as transition firing or token emittance of a queue are discrete events. This makes QPNs a natural candidate for discrete-event simulation which is preferable to continuous simulation whenever the simulation can be split into discrete events [29]. In discrete-event simulation the operation of a system is represented as a discrete sequence of events over time. When parallelizing a discrete-event simulation, the causal ordering of these events needs to be guaranteed in order to ensure correctness of the results. In general, parallelization of a discrete-event simulation can

be performed at the following three levels [19]:

- At the *application level*, replicated simulation runs can be executed in parallel. Replicated simulation runs are often used for variance reduction of a long simulation run by running multiple shorter runs or for investigating a large number of different parameter settings [15]. This parallelization approach works well independently simulated model.

- At a *functional level*, individual functions (e.g., event list manipulation and random number generation) can be executed in parallel to the main simulation thread. In order to achieve significant speed-ups, this parallelization approach requires computationally expensive functions [15]. However, such functions are often an indicator for an inefficient implementation where optimization offers more potential than parallelization[18].

- At the *event level*, a simulated model can be decomposed in spatial or temporal partitions. Each partition can be executed in its own process. In order to ensure the causal ordering of events, a synchronization between the processes is required.

In this paper, we will focus on event-level parallel simulation QPNs, as this approach promises to provide advantages when simulating large models. By decomposing a large QPNs into smaller partitions, the overall simulation time may be reduced significantly.

For an event-level parallel execution of a single simulation run several design questions need to be answered. We need to decide on: model decomposition, synchronization algorithm (conservative vs. optimistic), and communication mode (synchronous vs. asynchronous). In the following, we give a short overview of different techniques typically used for event-level parallel simulation. A detailed introduction to the topic can be found in [15].

Model decomposition: Parallel discrete-event simulation decomposes the simulation model into disjunct partitions, which are simulated in parallel. The corresponding process that executes the events within a single partition is called *logical process* (LP). The decomposition goal is to keep the load between LPs balanced while minimizing the communication costs. The decomposition is either spatial or temporal. Temporal decomposition requires (rarely available) knowledge about future system states to achieve speedup. Spatial decomposition partitions the net into several connected subnets. It scales with model size and offers more potential for speedup than temporal decomposition [33].

Synchronization algorithm: The event processing of the simulation is executed in an event loop in each LP. Hence, each LP has its dedicated virtual simulation time. In order to guarantee the local causality constraint (i.e., each LP processes events in nondecreasing timestamp order), timestamps need to be exchanged between LPs. For a correct causality of the simulation, synchronization algorithms are required to ensure that no token with a smaller timestamp than the LP's current time may arrive later in the simulation. Conservative synchronization enforces the local causality constraint at every point in time. They pause processing if they reach a lookahead border waiting for the other LPs to progress further. Lookahead is the (virtual) time span from the current clock for which an LP is guaranteed to receive no incoming events with smaller timestamps from neighboring LPs. It is the time an LP can process without synchronization. The largest possible lookahead is the time interval an LP has from the current clock time until processing of the next incoming token. Often lookahead estimation in conservative approaches is far from optimal. The predicted lookahead is less than the maximum possible lookahead. In the worst case, the predicted lookahead is zero, which results in sequential simulation. Optimistic simulation approaches allow temporary violations of the local causality constraint. Single LPs may pass the lookahead border. Consequently, events with a timestamp previous to the current LP time may arrive. Straggler messages, which is the name of late events, are cleaned by rollbacks. A rollback restores a state previous to the straggler message from history. This enables the correct processing of the straggler. Optimistic approaches aim at utilizing the model inherent parallelism, introducing additional computational costs for rollbacks and history preservation. It depends on the application whether conservative or optimistic approaches perform better. Both approaches fail at concurrent event processing if the model contains little inherent parallelism.

Communication mode: Asynchronous or synchronous communication can be used to exchange timestamp guarantees between LPs. In synchronous approaches all LPs progress until they reach a certain barrier. For conservative simulation, this barrier is the lookahead. When entering the barrier, an LP waits until all other LPs have also reached the barrier, before continuing with the processing. During barrier operation, the lookahead value is updated in a global operation. In contrast, asynchronous approaches update the lookahead value using message-based approaches and have no global operation.

Parallel discrete-event simulation is an established research area. The basic algorithms are well understood and a lot of effort was put in optimizing them. Independent of the chosen paradigm additional overheads are introduced by parallelization. The parallelization of simulation will only yield a speedup when the simulated model has good lookahead characteristics. The effort for additional data structures – like incoming token lists – is inevitable to all parallel simulation approaches. Synchronization operations are costly as well. These overheads require reasonably sized lookahead intervals enabling multiple events to be processed before the next synchronization operation. Previous studies show that the potential speedup through parallelization is limited by characteristics inherent to the simulation model. The overhead introduced by parallelization even slows down the simulation in many situations when compared to a sequential execution.

4. PARALLEL SIMULATION OF QUEUEING PETRI NETS

The main research questions for parallel simulation are: Which simulation models offer speedup potential through parallel simulation and how to maximize the speedup for these models. While this question has been answered for other types of models in different domains, to the best of our knowledge, no previous work considered the potential of QPN models (including queueing places) for speedup through parallel discrete-event simulation. Generally speaking, QPNs

inherit the challenges of both formalisms, QNs *and* PNs. Approaches for QNs have been proposed to estimate lookaheads for different queue scheduling strategies (FCFS, PS, IS, LCFS, PRIO). One challenge is that scheduling strategies other than First-Come-First-Serve (FCFS) and Infinite Server (IS) exhibit a gap between the predicted and the actual next token emittance. Performance engineering models often use a Processor Sharing (PS) strategy to model CPUs. PS offers less potential for lookahead prediction as jobs leave in a different order than they arrived at the queue. To predict the next token emittance time it is necessary to maintain a list of future events. The length of this list limits the achievable parallelism as it is an upper bound on the maximum possible lookahead. Furthermore, maintaining the future event list introduced additional overheads reducing the potential speedup even further. In addition to this, not all relevant queueing strategies for performance modeling have been covered so far. For example, many system designs give preference to short jobs, applying policies like Non-Preemptive-Shortest-Job-First (SJF) or Preemptive-Shortest-Job-First (PSJF) to disk scheduling [47] and web server scheduling [40, 41]. To apply classical algorithms, we would have to extend lookahead approaches to further scheduling strategies. Moreover, QPN performance models often use multiple token colors. Token colors increase the overhead for lookahead computation as a prediction for each color is required.

Due to the challenges introduced by the scheduling strategies and the different token colors, we argue that the lookahead potential of general QPNs is relatively low and the synchronization overhead outweighs the benefits. Initial experiments using a parallel implementation with conservative synchronization and synchronous communication, also supported this conclusion resulting in a slow-down even for primitive nets limited to FCFS scheduling and one token color. Without constraining the problem by introducing additional assumptions on the simulation model, reasonable speedups are in general unrealistic, as also noted by Fujimoto [14]. However, parallel simulation of QPNs can benefit under the assumption that there are no cycles between LPs. In this section, we propose a novel approach to circumvent the lookahead estimation challenge for QPNs and thus leverage the speedup potential for a certain class of QPN models. Section 4.1 illustrates the high-level idea of our approach. As the high-level idea is not sufficient to achieve speedup, 4.2 provides implementation details. Section 4.3 explains our approach to QPN decomposition into LPs.

4.1 General Idea

For an efficient parallelization, we assume that the QPNs model can be decomposed into cycle-free partitions which can be described as Directed Acyclic Graph (DAG). This may sound like a strong restriction but can be accomplished throughout a smart decomposition for all performance models with an open workload (i.e., the arrival rate is constant and independent of the processing of the system). We illustrate the idea using the example depicted in Figure 1. In this example, model elements B and C are cyclically connected and have to be merged to receive a DAG. We name an LP A preceding to another LP B if A may send tokens to B. Then B is successor of A. For an efficient parallelization, each predecessor has to provide a good lookahead guarantee to its successor. In our approach, each LP passes its *clock value as*
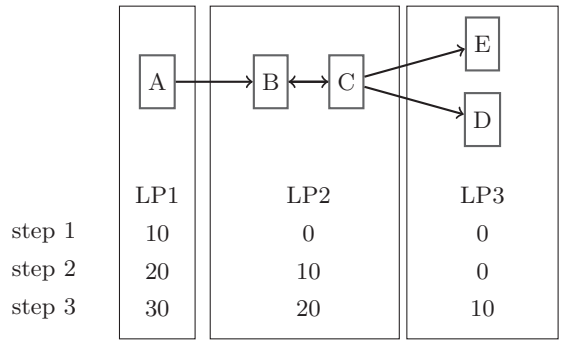

	LP1	LP2	LP3
step 1	10	0	0
step 2	20	10	0
step 3	30	20	10

Figure 1: Stepwise virtual time propagation

a lookahead border to its succeeding LPs. To enable parallel execution, it is necessary that LPs are ahead in time of preceding LPs, which has to be ensured by the program logic. Figure 1 shows an example procedure. In the first step only LP1 is processing ten time steps. In the next step, additionally LP2 processes until reaching the lookahead border of 10 granted by LP1. Before entering the third step, LP1 grants lookahead border of 20 to LP2 and LP2 grants execution until 10 to LP3. Starting from the third step, all LPs may process ten time units per synchronization step. For every step, the lookahead of an LP can be set to the minimum time of its predecessor(s). Referring to the example, the key parts for our parallel simulation are:

Stepwise time propagation. Predecessors of LP have to be ahead in time to enable parallel execution. To reduce the synchronization overhead, we we propose a stepwise increment if no cycles exist. The time step size, which has been set to 10 in the example, can be tuned for a concrete scenario.

DAG decomposition. The requirement of a cycle-free initial model can be relaxed dramatically through smart decomposition. The initial model may contain cycles. The transformation to a DAG happens by merging all cycle elements into one LP. As a consequence, the model is free of cycles on the LP meta-level.

Through the DAG decomposition, we can extend the applicability of our approach to *all open workload models* where cycles are either inexistent or can be merged. Models with a closed workload cannot benefit from our idea. They contain one all-encompassing cycle which results in a single LP decomposition equal to sequential simulation. The assumption of the absence of cycles allows to avoid complex lookahead computation and even grant arbitrary lookahead intervals. This arbitrary lookahead is the key to achieving an efficient parallel simulation. The restriction on open workloads for QPNs is necessary, as the synchronization overhead for conservative approaches significantly outweighs the benefits through parallelization. It depends on system design whether to model open or closed workloads. Schroeder et al. [41] provide a detailed discussion and guidelines for the choice of open or closed workloads. Current trends like cloud applications or internet of things result in increasing numbers of performance models with open workloads, as, according to Schroeder et al., systems with a high number of simultaneous users should be modeled with an open workload.

Our approaches for stepwise time propagation and DAG decomposition may be ported to formalisms other than QPNs as well. The LPs `A, B` or `C` could stand for queues of a QN or a set of places and transitions in a Stochastic Petri Net (SPN) or subparts of other formalisms.

4.2 Implementation

Our simulation process for the three LPs of the initial example is depicted in Figure 2. The newly implemented parallel simulation starts if the model can be decomposed to a DAG. Otherwise, sequential simulation is started automatically. Thereby, the user does not need to consider the the suitability of a model for parallel simulation. The limita-
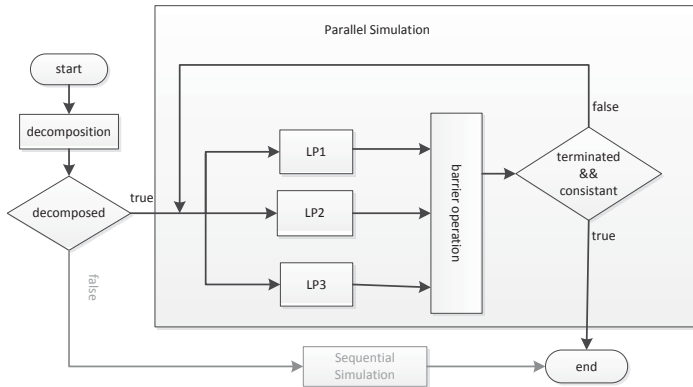

Figure 2: Parallel Simulation Process

tion to models that can be decomposed to a DAG influences the choice of parallelization techniques and paradigms. Optimistic parallelization would result in additional overheads (e.g., for state saving) in order to increase intervals between synchronizations. The stepwise time concept enables to increase these intervals to an arbitrary length. Hence, we minimize computational overhead by applying conservative synchronization where the minimal clock of the predecessors of an LP is used as its lookahead. The next choice is on synchronous or asynchronous communication. We apply a synchronous barrier-based parallelization, as the global barrier operation enables efficient lookahead updates. During each barrier operation the simulation engine updates the lookahead time for each LP. Then each LP processes all events in its safe to process time range and enters the barrier afterwards. The procedure repeats until the stopping criterion of the simulation is fulfilled and and the model is in a consistent state again. During parallel simulation the LPs have different virtual times which results in a globally inconsistent state of the model, as depicted in Figure 1. A consistent state is restored on simulation termination by choosing the event with the highest timestamp (which resides within an LP with no predecessors) and processing all events with smaller timestamps. Token generation at LPs without predecessors stops and then the LPs process all remaining events up to the stopping virtual time. Finally, we equalize time so that all LPs have the same virtual time which results in a consistent global state again. During this consistency step, a few additional events may be simulated compared to sequential simulation. However, given that the

stopping criteria is specified as a minimum requirement on the accuracy, this does not invalidate the results.

The time safe to process for each LP is set during barrier operation. The mechanism depends on whether an LP has predecessors or not. In models with open workloads, at least one LP exists without predecessors exists (called *workload generator* in the following). The absence of predecessors (i.e. no incoming events) implies the absence of lookahead constraints. An infinite lookahead prohibits a balanced simulation as the LP would never enter the barrier. Consequently, we require an artificial lookahead border which forces the LP to enter the barrier. Ideally, this artificial border creates a balanced token flow between LPs to get a balanced simulation. The decision for the artificial lookahead border size depends on the implementation and is a possible candidate for auto-tuning mechanisms. If the border is chosen too small the workload-generator LP sends few or even no tokens to its successors. This results in multiple barrier operations with low progress. A long period allows for processing of multiple events which may result in a surge of events being sent to the successor LPs. The incoming token list of an LP is a priority queue and insertions get more expensive with increasing queue length. We refined the idea of a user defined time interval in the form of a specified number of tokens to be processed before entering the barrier. Thereby, a more constant token flow can be generated and a sensitive time step size parameter can be avoided.

Parallel simulation depicts a special case, where standard barrier implementations fail to achieve good performance. In standard barrier implementations, as e.g., provided by the Java standard library, threads leave the CPU while waiting on a barrier. Especially in Java, this wait step causes an expensive operating system operation [3]. Instead of expensive passive wait, active wait saves the costs for leaving and reentering CPU. In many application scenarios, the costs for barrier operations are negligible. In parallel simulation, we enter the barrier very frequently (every few *microseconds*). Hence, active wait helps to reduce the synchronization overheads significantly. Another problem of standard implementations is the access contention on the entering function of a barrier. This access contention can be parallelized using hierarchical barriers [3]. We employ the barrier implementation of Peschlow et al. [35] for our parallel simulation engine. Their implementation utilizes the described techniques for high frequency parallel simulation.

4.3 Decomposition

Decomposition into LPs is a precondition for parallel simulation. Zeng et al [48] name the primary partitioning goal for parallel simulation as decomposition "of the simulation model into a number of components that keep the computational load approximately balanced while minimizing the communication overheads". The decomposition problem can be reduced to an NP-hard bin-packing problem. The NP-hard problem suggests a greedy approach that not necessarily represents an optimal solution. One approach is to use predefined hierarchies of Hierarchical Queueing Petri Nets (HQPNs) which has been proposed in [18]. However, this approach is inapplicable due to the lack of predefined hierarchies and the author does not provide guidelines on how to define hierarchies for a specific scenario. Originally developed for Timed (Transition) Petri Nets (TPNs), one can apply the minimum region concept of Chiola and Fer-
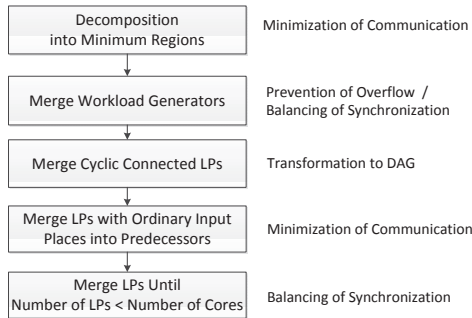
Figure 3 Decomposition Scheme:

Decomposition into Minimum Regions	Minimization of Communication
Merge Workload Generators	Prevention of Overflow / Balancing of Synchronization
Merge Cyclic Connected LPs	Transformation to DAG
Merge LPs with Ordinary Input Places into Predecessors	Minimization of Communication
Merge LPs Until Number of LPs < Number of Cores	Balancing of Synchronization

Figure 3: Decomposition Scheme

scha [10] to QPNs. A minimum region includes all places that share one transition and all transitions for which one of these places is an input place. The advantage of the minimum region idea is that the choice which transition fires next resides in one minimum region which reduces communication overheads for parallel simulation. After decomposition in minimum regions, they can be assembled to larger units. We now describe our merging process for the minimum regions which results in a DAG which has only one LP without predecessors. An overview of the decomposition process is depicted in Figure 3. At first, we ensure constant token flow in case more than one workload generator exists. In that scenario, one workload generator may increase its virtual time faster than the others which may result in overflow situations. We prevent this by merging all workload generators into one LP. Another option would be to artificially set one workload generator as predecessors for the other workload generators. Next, we remove any cycles between LPs by merging all cyclically connected LPs. We receive a DAG by merging all strongly connected sets detected by Tarjan's algorithm [43]. Through the previous two steps, we obtain a DAG with only one LP without predecessors. Next, we merge LPs having ordinary places as input places into their predecessors. Transitions of such LPs fire directly whenever a token arrives at their ordinary incoming place(s). Consequently, there is no virtual time delay to their predecessors which makes them a suitable candidate for merging. Finally, we merge LPs until the number of LP is less or equal to the number of processor cores. The merging algorithm starts from the element with no predecessors and merges its successors until a certain size is reached. Then the procedure repeats with one of the successors. A more formal view of this algorithm is depicted in Algorithm 1. It receives a set of LPs (`lps`) which is merged until the number of LPs sinks below the `upperBound`. This parameter can either be set by the user or is automatically set to the number of available processor cores. Lines 4 to 14 show the merging process for the first $\#core - 1$ LPs. In the last step all remaining LPs are merged into the current to ensure that the total number of LPs is less or equal to the upper bound. The function `hasAdequateSize` in line 8 is used to decide whether to increase the size of the current LP or to select one of the successors as new initial merging spot. The decision is based on a ratio of the current LP's computational effort compared to the total effort. Our actual implementation uses LP's place count as indicator for this but further factors, e.g., number of branchings, internal recursion depth, queues and their queueing strategies, may be included. In general, the number of LPs has not necessarily to be less or equal to the

number of cores when multiple LPs are assigned to a thread. We decided against this option as computational overheads would increase.

Algorithm 1 Final Merging Step

1: **function** MERGEFINAL(lps, upperBound)
2:     current ← getLPWithoutPredecessor(lps);
3:     reachable ← ∅
4:     numLPs ← 1
5:     **while** numLPs < cores; **do**
6:         reachable.append(current.getSuccessors());
7:         **while** ¬ reachable.isEmpty() **do**
8:             successor ← reachable.removeFirst();
9:             **if** ¬ hasAdequateSize(current) **then**
10:                current ← merge(current, successor);
11:            **else if** numLPs < cores **then**
12:                current ← successor;
13:                numLPs = numLPs + 1
14:            **else**
15:                **if** ¬ reachable.isEmpty() **then**
16:                    reachable.append(current.getSuccessors());
17:                **end if**
18:                break;
19:            **end if**
20:        **end while**
21:    **end while**
22:    **while** ¬ reachable.isEmpty() **do**
23:        successor ← reachable.removeFirst();
24:        current ← merge(current, successor);
25:        **if** ¬ reachable.isEmpty() **then**
26:            reachable.append(current.getSuccessors());
27:        **end if**
28:    **end while**
29: **end function**

---

5. CASE STUDIES

We conducted a series of case studies with open workload models to evaluate the speedup achieved through barrier-based parallel simulation. For each model, we compare the runtime of the parallel to the sequential case. Speedup is defined as the runtime of parallel implementation divided by the runtime of the sequential one. The significance level is determined using a two-sided Student's t-test rejecting the hypothesis that the runtime of parallel and sequential executions belong to one set. A validation which shows that sequential and parallel simulation yield the same results with equal accuracy has been performed in [46].

Except for the network traffic model, we ran our experiments on an Intel Xeon E5430 2x4core, 2.66GHz, 12MB L2 Cache per CPU and 32 GB RAM running Linux CENTOS 5.8 (Final). The network traffic model was simulated on a MacBook Pro with 4 cores Intel i7 having 2.2 GHz running OS X Yosimite(10.10.2). The number of used cores during simulation equals the number of LPs. We marked LPs decomposition by surrounding blue boxes in the figures. The achieved speedup depends on various factors, e.g., the size and decomposability of the model. We investigate the following aspects: (a) speedup for small-scale models, (b) speedup for medium-scale models, and (c) divide-and-conquer effects. Section 5.1 presents a case study with a performance model provided by a large SaaS provider. Sec-

tion 5.2 presents a medium-scale model of the SPECj Application Server benchmark. The case study presented in Section 5.3 uses a network model that has been generated by an automated transformation. Section 5.4 systematically evaluates different aspects of our implementation using a synthetic model.

5.1 Layered Architecture Model of a Large SaaS Provider

The QPN model shown in Figure 4 describes the performance behavior of a Customer Relation Ship (CRM) application of a large SaaS cloud provider in a simplified form. The model represents an application scenario comprising
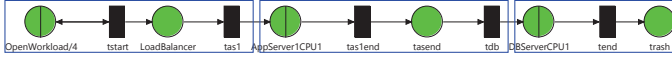
Figure 4: Performance Model Decomposition

two application servers (in our case each with 6 cores), a load balancer, a database server and a delay incurred by the storage system. The model has nine token colors, representing nine different transaction types. We measured the simulation runtime at QPME statistics level 4 (i.e., including response time histograms). The average speedup was 1.91 with significance level of $7.70290e^{-9}$. This example shows that even for small models speedup is possible with our approach.

5.2 SPECjAppServer2004

The QPN model in Figure 5 describes a deployment of the SPECj Application Server benchmark. The model contains a load balancer, a replicated application server tier and a replicated database tier. The model is based on the case study presented in [23]. We changed the model to an open workload using an equivalent parametrization. The model
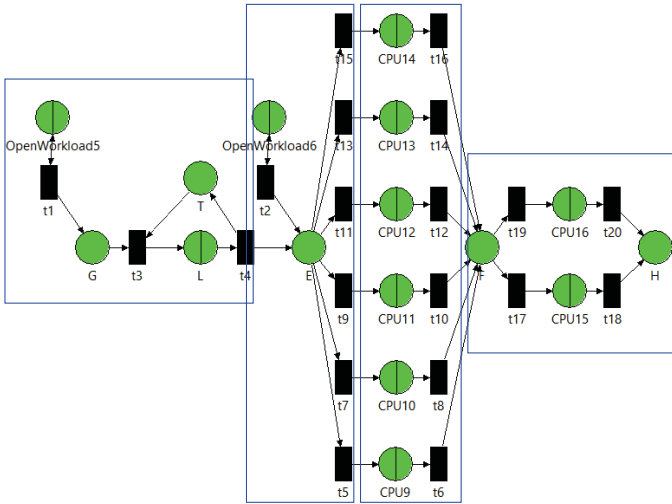

Figure 5: SPECj Application Server Benchmark Modeled With Open Workload

of the load balancer (`T,L, t3, t4`) includes a circular structure. To apply our stepwise virtual time all elements of a cycle have to be merged into one LP. Compared to the decomposition algorithm described in Section 4.3, we merged

`OpenWorkload5` into `E` instead of merging it with `OpenWorkload6` to improve load balancing. We obtained an average speedup of 2.45 using 4 cores with a significance level of $2.04099e^{-18}$. In general, we expect increased speedups through better load balancing as the model size increases.

5.3 Network Traffic Model

The network traffic model for this case study was generated using the DNI-to-QPN transformation published in [37]. Compared to QNs, QPNs offer advantages for modeling software contention in switches. The model has one server connected to another server by one switch. We applied a fully automated decomposition. We omit the depiction of the model due to space constraints.

Number of Threads/ LPs	2	3	4
Speedup	1.61	1.92	2.22

5.4 Synthetic Model

The previous subsections demonstrated the real-world applicability of parallel QPN simulation. However, a synthetic model allows for a systematic investigation of the scalability and speedup potential of our implementation. This section shows that the main reason for deviation from linear speedup is a non-optimally balanced decomposition.

We used a synthetic model that generates tokens and distributes them to multiple lanes. The structure for token generation and distribution remains constant whereas the number and length of lanes can be varied. Models are generated as $m \times n$. Variable m represents the number of lanes and n is the number of queueing places. Figure 6 shows an example of a 3×2 model.
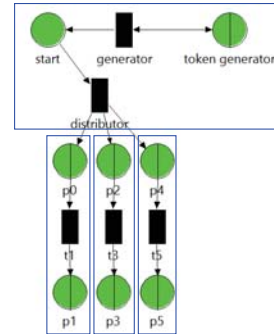

Figure 6: Generated Model With Three Lanes, Each of Length Two Queueing Places.

The model is partitioned into the constant token generation part and lanes, each representing an LP, so that the lanes execute in parallel. The two parameters m and n vary two aspects. The length of a lane determines the amount of operations between barrier synchronizations. The number of lanes determines the number LPs and thereby the number of cores that can be utilized. The synthetic model allows to specify an upper bound for the theoretical speedup which enables a comparison with actual measured execution times.

$$speedup = m \times f(m,n) + k \qquad (1)$$

whereas k represents the constant part of token generation, and $f(m,n)$ maps the synchronization overheads into a factor. Theoretically: $f(m,n) \leq 1$. We estimate a theoretical

upper bound for speedup as

$$speedup = m + 1 \qquad (2)$$

The first experiment we present keeps $m = 6$ fixed and varies the length of the lane n. Figure 7 shows speedups compared to n. As expected, an increased n increases speedup.
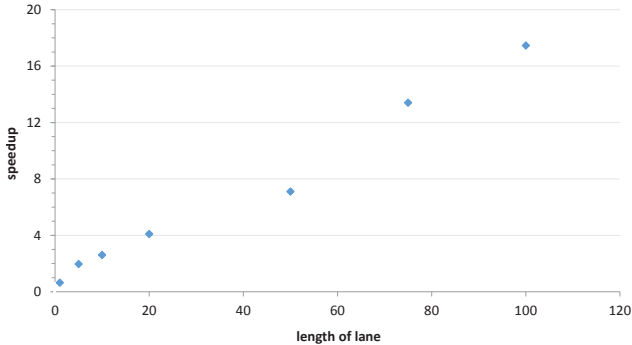

Figure 7: Speedup for Length of Lanes Variation

However, the experiments even show speedups above the theoretical optimum. This can be explained by two divide-and-conquer benefits of the parallel implementation. Firstly, we benefit form *cache effects*. Subproblems are smaller and can be kept in the cache. The parallel version is faster because of cheaper cache accesses instead of "expensive" RAM accesses. The second divide and conquer effect is algorithmic. The parallel version performs the choice of the next transition to fire LP-locally. Thereby less concurrently enabled transitions occur compared to the global solution and unnecessary choices for the next transition can be omitted.
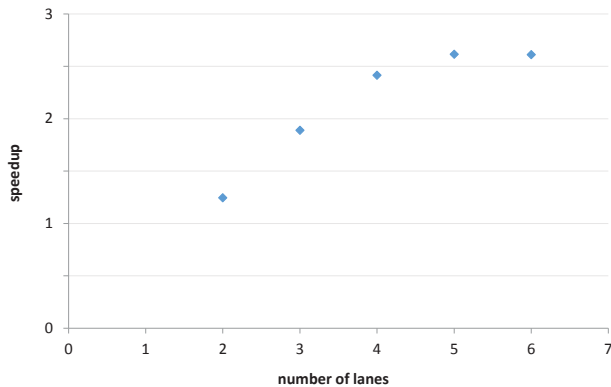

Figure 8: Speedup for Lane Number Variation

The second experiment, depicted in Figure 8, varies the number of lanes and keeps the length $n = 2$ fixed. The setup poses a very small load for the LPs, which reveals the barrier contention effect. The speedup is proportional to the number of lanes up to an asymptotic border. After that the speedup may even decline. The reason is the global barrier which becomes a bottleneck for high numbers of LPs. We optimized for common use cases at the expense of a scalability limitation. However, in scenarios where the load between barrier operations is higher, the asymptotic bound would appear at a much higher number of cores.

6. RELATED WORK

The question about suitability for parallel simulation has been answered for different types of models in different domains. To the best of our knowledge, no previous work considered the potential of QPN models for speedup through parallel discrete-event simulation. The only previous work we are aware of that considers the concurrent simulation of QPNs was presented in a master's thesis [18]. However, the thesis does neither consider any formalism specific aspect like lookahead for queues nor does it consider model decomposition. Instead the thesis provides a performance test for well known conservative and optimistic scheduling algorithms. Our work is the first to consider formalism specific discrete-event simulation of QPNs. The parallelization of a DAG, as we proposed, has been performed in different research areas. However, we could not find applications to simulation. The majority of related work is on parallelization on the event level for TPNs and QNs. Most publications in this area propose different algorithms optimized for different model use cases. In the following, we present a selection: A set of approaches build on specialized hardware, like vector machines or Graphics Processing Units (GPUs), which have a Single Instruction Multiple Data (SIMD) architecture. This hardware is able to solve vector operations very fast that appear when solving PN recurrence equations. For vector machines, SPN analysis has been parallelized [2]. For GPUs, SPN [16] and Hybrid Functional Petri Nets (HFPNs) [9] analysis has been parallelized. All SIMD approaches have in common that speedup is reached by specialized hardware matrix operations on sparse matrices but not by an efficient strategy. Multiple LP-based discrete-event parallelizations for QNs and TPNs have been proposed. Nicol and Roy [31] apply a conservative synchronization protocol for discrete-event analysis of TPNs. They propose a tripartition event types to improve efficiency. Chiola and Ferscha [10] describe how TPNs can be split in LPs to apply conservative and time warp synchronization. They define *minimum regions* by the use of (extended) conflict sets which can be merged to LPs. In general, they provide a good overview of advantages of different partitionings. Nketsa and Khalifa [33] modified the lookahead definition of [10] to better suit asynchronous communication. Fang et al. [11] refine the approach in [33] to fit for extended TPNs. Ammar and Deng [1] apply an optimistic time warp simulation to SPNs. They use a combination of spatial and time scale decomposition to derive LPs. Their time scale decomposition divides a large network into small subnets by separating short activities and long activities into different subnets. Ferscha [12] studies approaches to make the time warp algorithm more robust to different model domains for TPN. Besides TPNs, QNs have been a common benchmark for the evaluation of different synchronization protocols, e.g., [30, 36, 13, 45]. Nicol [30] proposed an efficient lookahead computation for FCFS queues with the future list technique. Based on Nicols future list, Lazowska et al [44, 27] extended the approach for further scheduling strategies (PS, LCFS, PRIO). Their approach assumes an upper bound of tokens in the queue.

7. CONCLUSION

We investigated the feasibility for parallel discrete-event QPN simulation. The QPN formalism inherits bad lookahead characteristics from its subformalisms, which makes QPNs a challenging candidate for parallel simulation. However, we showed that QPN models with open workloads are suitable for parallel execution. Support for parallel simulation of such models has been integrated into a general opensource simulation engine and extensively tested.

The experiments show speedups of 1.9 using 3 threads and 2.5 using 4 threads in the context of real-world case studies. In general, higher speedups are possible. The scalability analysis we presented using artificial models demonstrates that our implementation can provide up to superlinear speedups. During our work on parallel QPN simulation, many new questions arose. We consider improvements on decomposition to be the most promising field for future work. We apply a decomposition to minimum regions and use a set of merging rules. The decomposition can be improved by the use of runtime statistics. Furthermore, our currently static partitioning does not consider changing workloads during simulation. For further research, we propose a dynamic partitioning approach which could improve load balancing during a simulation run. Furthermore, additional case studies with large scale models can be conducted to further evaluate the scalability of the parallel simulation approach.

8. REFERENCES

[1] H. Ammar and S. Deng. Time warp simulation of stochastic petri nets. In *Petri Nets and Performance Models, 1991. PNPM91., Proceedings of the Fourth International Workshop on*, pages 186–195, Dec 1991.

[2] F. Baccelli and M. Canales. Parallel simulation of stochastic petri nets using recurrence equations. *ACM Transactions on Modeling and Computer Simulation*, 3:20–41, 1993.

[3] C. Ball and M. Bull. Barrier synchronisation in java. *Available from World Wide Web: http://www. ukhec. ac. uk/publications/reports/synch_java. pdf*, 2003.

[4] F. Bause. Queueing petri nets-a formalism for the combined qualitative and quantitative analysis of systems. In *Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on*, pages 14 –23, oct 1993.

[5] F. Bause, P. Buchholz, and P. Kemper. Qpn-tool for the specification and analysis of hierarchically combined queueing petri nets. In *BAUSE (EDS.) QUANTITATIVE EVALUATION OF COMPUTING AND COMMUNICATION SYSTEMS, LECTURE NOTES IN COMPUTER SCIENCE*, pages 224–238. Springer, 1995.

[6] F. Bause and P. S. Kritzinger. *Stochastic Petri nets - an introduction to the theory (2. ed.)*. Vieweg, 2002.

[7] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[8] F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, and S. Kounev. Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-based Architectures. *IEEE Transactions on Software Engineering (TSE)*, 2014. Accepted for publication.

[9] G. Chalkidis, M. Nagasaki, and S. Miyano. High performance hybrid functional petri net simulations of biological pathway models on cuda. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 8(6):1545–1556, Nov.-Dec. 2011.

[10] G. Chiola and A. Ferscha. Distributed simulation of timed petri nets: Exploiting the net structure to obtain efficiency. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 146–165. Springer Berlin Heidelberg, 1993.

[11] X. Fang, Z. Xu, and Z. Yin. Distributed processing based on timed petri nets. In *Proceedings of the Third International Conference on Natural Computation - Volume 05*, ICNC '07, pages 287–291, Washington, DC, USA, 2007. IEEE Computer Society.

[12] A. Ferscha. Adaptive time warp simulation of timed petri nets. *Software Engineering, IEEE Transactions on*, 25(2):237–257, Mar/Apr 1999.

[13] R. M. Fujimoto. Lookahead in parallel discrete event simulation. Technical report, DTIC Document, 1988.

[14] R. M. Fujimoto. Parallel and distributed discrete event simulation: algorithms and applications. In *Proceedings of the 25th conference on Winter simulation*, WSC '93, pages 106–114, New York, NY, USA, 1993. ACM.

[15] R. M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 2000.

[16] R. Geist, J. Hicks, M. Smotherman, and J. Westall. Parallel simulation of petri nets on desktop pc hardware. In *Simulation Conference, 2005 Proceedings of the Winter*, page 10 pp., dec. 2005.

[17] K. Jensen. Coloured Petri Nets and the invariant-method. *Theoretical Computer Science*, 14:317–336, 1981.

[18] T. Jürgens. Verteilte simulation von hqpns auf einem netzwerk von workstations. Master's thesis, Universität Dortmund, Fachbereich Informatik, 1997.

[19] F. J. Kaudel. A literature survey on distributed discrete event simulation. *SIGSIM Simul. Dig.*, 18(2):11–21, June 1987.

[20] D. G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, 24(3):338–354, 1953.

[21] S. Kounev. *Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction*. Shaker Verlag, Ph.D. Thesis, Technische Universität Darmstadt, Germany, Aachen, Germany, December 2005. Best Dissertation Award from the "Vereinigung von Freunden der Technischen Universität zu Darmstadt e.V.".

[22] S. Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *Software Engineering, IEEE Transactions on*, 32(7):486 –502, july 2006.

[23] S. Kounev and A. Buchmann. Performance modeling of distributed e-business applications using queueing petri nets. In *Proceedings of the 2003 IEEE*

International Symposium on Performance Analysis of Systems and Software (ISPASS 2003), Austin, Texas, USA, March 6-8, 2003, pages 143–155, Washington, DC, USA, 2003. IEEE Computer Society. Best-Paper-Award at ISPASS-2003.

[24] S. Kounev and A. Buchmann. SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 63(4-5):364–394, May 2006.

[25] S. Kounev and A. Buchmann. On the Use of Queueing Petri Nets for Modeling and Performance Analysis of Distributed Systems. In V. Kordic, editor, *Petri Net, Theory and Application*. Advanced Robotic Systems International, I-Tech Education and Publishing, Vienna, Austria, February 2007.

[26] S. Kounev and S. Spinner. *QPME 2.0 User's Guide*, 2011.

[27] Y.-B. Lin and E. Lazowska. Exploiting lookahead in parallel simulation. *Parallel and Distributed Systems, IEEE Transactions on*, 1(4):457–469, 1990.

[28] P. Meier, S. Kounev, and H. Koziolek. Automated transformation of component-based software architecture models to queueing petri nets. In *19th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011), Singapore, July 25–27*, 2011.

[29] P. Merkle. Comparing process- and event-oriented software performance simulation. Master's thesis, Karlsruhe Institute of Technology (KIT), Germany, 2011.

[30] D. M. Nicol. Parallel discrete-event simulation of fcfs stochastic queueing networks. In *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, PPEALS '88, pages 124–137, New York, NY, USA, 1988. ACM.

[31] D. M. Nicol and S. Roy. Parallel simulation of timed petri-nets. In *Proceedings of the 23rd conference on Winter simulation*, WSC '91, pages 574–583, Washington, DC, USA, 1991. IEEE Computer Society.

[32] P. O. Nierstrasz. Concurrent programming. Lecture, 1999.

[33] A. Nketsa and N. B. Khalifa. Timed petri nets and prediction to improve the chandy-misra conservative-distributed simulation. *Applied Mathematics and Computation*, 120(1-3):235 – 254, 2001. <ce:title>The Bellman Continuum</ce:title>.

[34] R. Osman, D. Coulden, and W. Knottenbelt. Performance modelling of concurrency control schemes for relational databases. In A. Dudin and K. Turck, editors, *Analytical and Stochastic Modeling Techniques and Applications*, volume 7984 of *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg, 2013.

[35] P. Peschlow, A. Voss, and P. Martini. Good news for parallel wireless network simulations. In *Proceedings of the 12th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems*, MSWiM '09, pages 134–142, New York, NY, USA, 2009. ACM.

[36] D. A. Reed, A. D. Malony, and B. McCredie. Parallel

discrete event simulation using shared memory. *IEEE Trans. Softw. Eng.*, 14(4):541–553, Apr. 1988.

[37] P. Rygielski and S. Kounev. Data Center Network Throughput Analysis using Queueing Petri Nets. In *34th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2014 Workshops). 4th International Workshop on Data Center Performance, (DCPerf 2014)*, pages 100–105, June 2014.

[38] K. Sachs, S. Kounev, and A. Buchmann. Performance modeling and analysis of message-oriented event-driven systems. *Journal of Software and Systems Modeling (SoSyM)*, pages 1–25, February 2012.

[39] M. Saliminia. Multi-tenant performance models to guarantee performance isolation. Master's thesis, Karlsruhe Institute of Technology (KIT), 2013.

[40] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Internet Technol.*, 6(1):20–52, Feb. 2006.

[41] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, volume 6, pages 18–18, 2006.

[42] S. Spinner, S. Kounev, and P. Meier. Stochastic modeling and analysis using qpme: Queueing petri net modeling environment v2.0. In S. Haddad and L. Pomello, editors, *Proceedings of the 33rd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2012)*, volume 7347 of *Lecture Notes in Computer Science (LNCS)*, pages 388–397. Springer-Verlag, 6 2012.

[43] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[44] D. B. Wagner and E. D. Lazowska. Parallel simulation of queueing networks: limitations and potentials. In *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '89, pages 146–155, New York, NY, USA, 1989. ACM.

[45] D. B. Wagner, E. D. Lazowska, and B. N. Bershad. Techniques for efficient shared-memory parallel simulation. Technical Report TR-88-04-05, Dept of Computer Science, Univ. of Washington, Seattle, WA 98195, Aug. 1988.

[46] J. Walter. Parallel Simulation of Queueing Petri Net Models. Master thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, October 2013.

[47] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *SIGMETRICS*, pages 241–252, 1994.

[48] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. In *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, pages 154–161, May 1998.