Autom Softw Eng
DOI 10.1007/s10515-017-0214-5

Automatic performance prediction of multithreaded programs: a simulation approach

Alexander Tarvo[1] · **Steven P. Reiss**[2]

Received: 13 July 2016 / Accepted: 13 March 2017

Abstract The performance of multithreaded programs is often difficult to understand and predict. Multiple threads engage in synchronization operations and use hardware simultaneously. This results in a complex non-linear dependency between the configuration of a program and its performance. To better understand this dependency a performance prediction model is used. Such a model predicts the performance of a system for different configurations. Configurations reflect variations in the workload, different program options such as the number of threads, and characteristics of the hardware. Performance models are complex and require a solid understanding of the program's behavior. As a result, building models of large applications manually is extremely time-consuming and error-prone. In this paper we present an approach for building performance models of multithreaded programs automatically. We employ hierarchical discrete-event models. Different tiers of the model simulate different factors that affect performance of the program, while interaction between the model tiers simulates mutual influence of these factors on performance. Our framework uses a combination of static and dynamic analyses of a single representative run of a system to collect information required for building the performance model. This includes information about the structure of the program, the semantics of interaction between the program's threads, and resource demands of individual program's components. In

This work was carried out when Alexander Tarvo was at Brown University.

✉ Alexander Tarvo
alextarvo@gmail.com

Steven P. Reiss
spr@cs.brown.edu

[1] Google Inc, Kirkland, WA, USA

[2] Brown University, Providence, RI, USA

Published online: 28 March 2017

our experiments we demonstrate that models accurately predict the performance of various multithreaded programs, including complex industrial applications.

Keywords Program analysis · Performance · Modeling · Simulation

1 Introduction

Multithreaded programs utilize resources of modern hardware efficiently. However, the behavior of multithreaded programs is significantly more complex than the behavior of single-threaded applications. Threads rely on synchronization to enforce the ordering of computations and to protect shared data. Moreover, multiple threads use shared hardware resources, such as the CPU, disks, and the network, simultaneously. This results in the parallel execution of some parts of the program's code and the sequential execution of others.

As a result, multithreaded programs demonstrate complex non-linear dependency between the configuration and performance. Configurations may reflect variations in the workload, program options such as the number of threads, and characteristics of the hardware. To better understand this dependency a *performance prediction model* is used. Such a model predicts performance of a program in different configurations.

Performance models are essential for a variety of applications (Israr et al. 2005; Bennani and Menasce 2005; Narayanan et al. 2005). For example, a model may be used to find a good configuration for deploying the Tomcat web server. For each combination of configuration parameters, such as the number of available CPU cores, the number of Tomcat working threads, and the rate of incoming connections, the model will predict response time, throughput, and resource utilization for Tomcat. A configuration that utilizes resources efficiently and satisfies a service agreement can be used for deployment. Performance models also can be used to detect performance anomalies and discover bottlenecks in the program.

Modern multithreaded applications can be large and complex, and are updated regularly. Building their models manually is extremely time-consuming and error-prone. To be practical, building such models should be automated.

However, building performance models of such applications in general is quite difficult. First, it requires discovering the queues, threads, and locks in the program; details of their behavior; and the semantics of their interaction. Doing this automatically requires complex program analysis. Second, it requires measuring demand for hardware resources such as the CPU, disk, and network. This is a complex problem that requires collecting and combining information from multiple sources. Third, the performance of a parallel system is dependent on its contention for computation resources and locks. Accurate modeling requires simulating these in detail.

This paper presents an approach towards automated performance modeling of multithreaded programs. Its main contribution is a combination of a model that accurately simulates complex synchronization operations in a program and a methodology to build such models automatically. Specifically, the paper makes the following technical contributions:

– A simulation model for predicting performance of multithreaded programs;

– A combination of static and dynamic analyses for understanding the structure and semantics of multithreaded programs automatically;
– An approach for collecting parameters of performance models from user- and kernel-mode traces;
– Verification of our approach by constructing models of various multithreaded programs.

We implemented our approach into a PERformance SImulation Kit (PERSIK) framework for building models of Java programs. We employed PERSIK to build performance models of various multithreaded programs. These programs were subjected to both CPU-bound and I/O bound workloads, and they were executed on different hardware, including systems with 4 and 16 CPU cores. In all cases our models were built from a single configuration of the program, and predicted its performance in a variety of configurations, including different number of working threads, CPU cores, and workload intensities. Our predictions included performance metric of the program, such as a throughput or a response time, and utilization of resources, such as a hard drive or a CPU.

First, we used PERSIK to build models of various small- to medium-size programs. These programs included scientific applications, a financial application, a multimedia program, and a web server. Sizes of these programs varied from 1006 to 3207 lines of code (LOC). The mean relative prediction error $\overline{\varepsilon}$ was in (0.062–0.255) for CPU-bound workloads and $\overline{\varepsilon} = 0.249$ for I/O bound workloads. By modeling these programs we demonstrate that our simulation framework can predict performance of various programs that use different synchronization constructs and hardware resources.

Second, we built models of large industrial programs such as Apache Tomcat web server[1] and Sunflow 3D renderer.[2] Size of these programs varied from 21,987 to 28,3143 LOC. The $\overline{\varepsilon}$ was in (0.032–0.134) for CPU-bound workloads and $\overline{\varepsilon} = 0.249$ for I/O bound workloads. These results demonstrate that out approach can accurately predict performance of large, industrial-grade multithreaded programs.

While working on the automatic model generation we made important findings. First, the analysis of a program could be greatly simplified if that program relies on well-defined implementation of high-level locks (semaphores, barriers, blocking queues etc.), instead of constructing them from low-level synchronization mechanisms such as monitors and critical sections. In this work we rely on this assumption to build performance models automatically.

Second, the resulting model must be simple and compact. Elaborate full-system models can be very accurate. However, due to their size and complexity they can run slower than the actual program, which defeats the whole purpose of performance prediction. Moreover, the complex performance model is very hard to understand and debug, if necessary. Building compact models requires identifying program constructs that do not have significant impact on performance, and excluding these constructs from the model.

[1] http://tomcat.apache.org/.

[2] http://sunflow.sourceforge.net/.

Third, accurate prediction requires precise measures of resource demands for the elements of the program. In certain cases small errors in measuring resource demands can lead to large prediction errors.

The presented paper extends our prior work (Tarvo and Reiss 2012, 2014) in nearly every aspect. We greatly expand the definition of our models, providing important information on components of the model, their parameters, and interactions. We describe our technique for automatic model generation in more detail, and outline its practical implementation. We extend the verification section with results from building the performance models on different hardware. Finally, we provide a more elaborate comparison of our work to the state of the art and present an extended discussion of the results of our study.

The rest of the paper is organized as following. Section 2 outlines scope of this work and challenges we faced. Section 3 defines our performance models. Section 4 describes a methodology to generate performance models automatically in detail. Section 5 discusses results of experiments with performance models and their accuracy. Section 6 discusses findings that we made about performance model, their limitations and outlines future work. Finally, Sect. 8 concludes the paper.

2 Scope and challenges

In this work we analyze performance of multithreaded applications such as servers, multimedia programs, and scientific computing applications. Such programs split their workload into separate *tasks* such as an incoming HTTP request in a web server, a scene or a some part of it in a 3D renderer, or an object in a scientific computing application (Peierls et al. 2005). We do not model the performance of individual tasks or requests; instead *we model and predict the aggregate performance of the system for a given workload*.

Processing tasks is parallelized across thread pools. A *thread pool* is a set of threads that have same functionality and can process tasks in parallel. Multiple threads rely on synchronization to ensure semantic correctness (e.g. the thread may start executing only after a barrier is lifted) and to protect shared data. This results in the parallel execution of some computations and the sequential execution of others. Threads also use shared hardware resources, such as the CPU, disks, and the network simultaneously, which may lead to their saturation. This combination of locking and simultaneous resource usage leads to complex non-linear dependencies between configuration parameters of the program and its performance. As a result, even an expert may be unable to understand such dependencies on a quantitative level. The best approach is to build a performance prediction model.

We concentrate on the following aspects of performance modeling:

Automatic generation of performance models We minimize the need for human participation in building the model. All our program analysis and model generation are done automatically. The analyst need only inspect the generated model and specify configurations in which performance should be predicted and the metrics that should be collected.

Generating models from running a program in a single configuration Building the model should not require running the program many times in many configurations. Such experimentation is time-consuming and may not be feasible in a production environment. Instead, we want to generate the model by running a program in a single *representative configuration*. In this configuration the behavior and resource demands of the program approach the behavior and resource demands of a larger set of configurations.

Accurate performance prediction for a range of configurations Our goal is to accurately predict program-wide performance metrics such as the response time, throughput, or the running time of the program; as well as the utilization of hardware resources, such as the CPU, hard drive, and network.

Modeling programs running on commodity hardware We concentrate on predicting the performance of programs on commodity hardware. Predicting the performance of programs running on specialized systems such as a grid or cluster would require developing an additional set of hardware models and potentially different approach for program analysis, which is beyond the scope of this paper.

Building a general-purpose model We didn't tailor our models to a specific use case. We build predictive models that could be used for a variety of tasks. Porential applications of our models include:

- serving as a decision making element in the autonomic data center (Bennani and Menasce 2005). Given an application and a workload, the model would automatically detect configurations providing both high performance and efficient resource utilization;
- answering "what-if" questions. Answering questions like "what will be the performance of the system for a given combination of configuration parameters?" or "what combination of configuration parameters will produce the desired performance?" is essential for capacity planning (Narayanan et al. 2005);
- detecting performance anomalies in the running software system (Thereska and Ganger 2008). Significant and systematic deviations of measured performance from the predicted performance are manifestations of the system's abnormal behavior.

Constructing performance models of complex, multithreaded systems is a challenging problem. The primary challenges are:

Accurate modeling of locks and hardware resources Performance of a multithreaded program is determined by contention of shared resources such as the CPU, disks, and locks. To accurately simulate resource contention the model must simulate locks, hardware, and corresponding OS components, such as the thread and I/O schedulers, and interactions between those. Building models of locks, OS and hardware that are both fast and accurate is challenging.

Discovering the semantics of thread interaction Building the performance model requires knowledge of the queues, buffers, and the locks in the program, their semantics (e.g. is this particular lock a semaphore, a mutex, or a barrier), and interactions (e.g. which thread reads or writes to a particular queue or accesses a particular lock). There are numerous ways to implement locks and queues, and to expose their functionality to

threads. Discovering this information automatically requires complex program analysis.

Discovering parameters of the program's components Performance of the program depends on parameters of its locks and queues, and on the resource demands of its threads. For example, the amount of time the thread has to wait on a semaphore depends on the number of available semaphore permits; the amount of time the program spends on the disk I/O depends on the amount of data it has to transfer. However, retrieving the parameters of locks and queues may require further program analysis and obtaining resource demands may require instrumenting the OS kernel.

3 Model definition

Below we define the model for predicting performance of multithreaded programs.

Our models rely on the concept of a task, which is a discrete unit of work that can be performed by the thread in the program (see Sect. 2). The performance of the task processing system can be described by various metrics, such as the response time R (an overall delay between task arrival and its completion), the throughput T (the number of task served in the unit of time), or the number of task dropped.

We use discrete-event simulation models, where the simulation time t is advanced by discrete steps (Law and Kelton 1997). It is assumed that the state of the system does not change between time advances.

Our models are built according to the hierarchical principle (Ferrari et al. 1983) and consist of three levels (tiers).

The high-level model explicitly simulates the flow of tasks as they are being processed by the program. The high-level model is a queuing network (Lazowska et al. 1984) with some important extensions. The middle-level models simulate delays that occur inside the program's threads as they process tasks. It is a probabilistic execution graph (PEG) of the thread, where vertices of the graph correspond to fragments of the thread's code. The lower-level model simulates delays that occur when multiple threads compete for a particular resource, such as a CPU, a hard drive, or a synchronization construct. The lower-level models are queuing networks with elements of statistical modeling.

The combination of three different tiers into a single performance prediction model is a key innovation of our work. Different model tiers simulate different factors that affect performance of the system, while interaction of these tiers simulate a mutual influence of these factors. Such architecture allows building accurate performance models of various multithreaded programs.

3.1 High-level model

The high-level model is based on a queuing network model (Lazowska et al. 1984). Service nodes of the model $\{tr_1, \ldots, tr_m\}$ correspond to the program's threads (the full notation used to describe the model is provided in the Table 1). Queues $\{q_1, \ldots, q_n\}$ in the model correspond to the program's queues and buffers used to exchange the

Table 1 Notation used for description of the model and its parameters

Notation used in a high-level thread model	
$q_1, \ldots, q_n$	A set of queues and buffers in the program
$tr_1, \ldots, tr_m$	A set of all threads in the program
$Tp_1, \ldots, Tp_k, k \leq m$	A set of all threads pools in the program, where $Tp_k = \{tr_i, \ldots, tr_j\}$
Notation used in a mid-level thread model	
$S = \{s_1 \ldots s_n\}$	The set of all nodes (code fragments) in the probabilistic execution graph (PEG)
$\delta : S \rightarrow P(S)$	Transition probabilities between nodes of the PEG
τ_i	Delay caused by executing CF $s_i \in S$
$c_i \in C$	Class of the CF s_i
$C = \{c_{CPU}, c_{IO}, c_{sync}, c_{in}, c_{out}\}$	CF classes: CPU-bound computations, I/O operations, synchronization operations, fetching and sending data to queues
$\Pi_{disk} = \langle dio_1, \ldots, dio_k \rangle$	Parameters of an I/O CF: a sequence of low-level I/O operations initiated by the CF
$\Pi_{CPU} = \langle CPUtime \rangle$	Parameters of a computation CF: the amount of CPU time
$\Pi_{sync} = \langle l_i, optype, tmout_{sync} \rangle$	Parameters of a synchronization CF: the lock that is called, the type of synchronization operation, the timeout
$\Pi_{inout} = \langle \{q_i, \ldots, q_j\}, optype, tmout_{inout} \rangle$	Parameters of c_{in} and c_{out} CFs: a set of queues that can be accessed, the type of the operation (send or fetch), the timeout
Notation used in a low-level model	
$L = \{l_1 \ldots l_m\}$	The set of all locks in a program
$\Pi_{lock} = \langle ltype, lparam \rangle$	Parameters of a lock: the lock type and the type-specific parameters

tasks between the different components of the software system. This includes queues and buffers present in the program itself and in the operating system (OS).

Each thread tr_i can be related to one (and only one) thread pool Tp_j. The *thread pool* or the thread group $Tp_j \in \{Tp_1, \ldots, Tp_k\}, k \leq m$ is a set of one or more threads that have same functionality and can process tasks in parallel. The number of threads in the pool is an important configuration parameter that can significantly affect the performance of the program. Each thread in a thread pool is represented as a separate service node in the model.

Figure 1 (top) depicts a high-level model of a simple web server. The server puts incoming connections into the OS connection queue q_1. The accept thread tr_1 fetches a connection from q_1 and forms a task object, which represents the HTTP request. Then accept thread tr_1 places that task into the program's task queue q_2. One of the

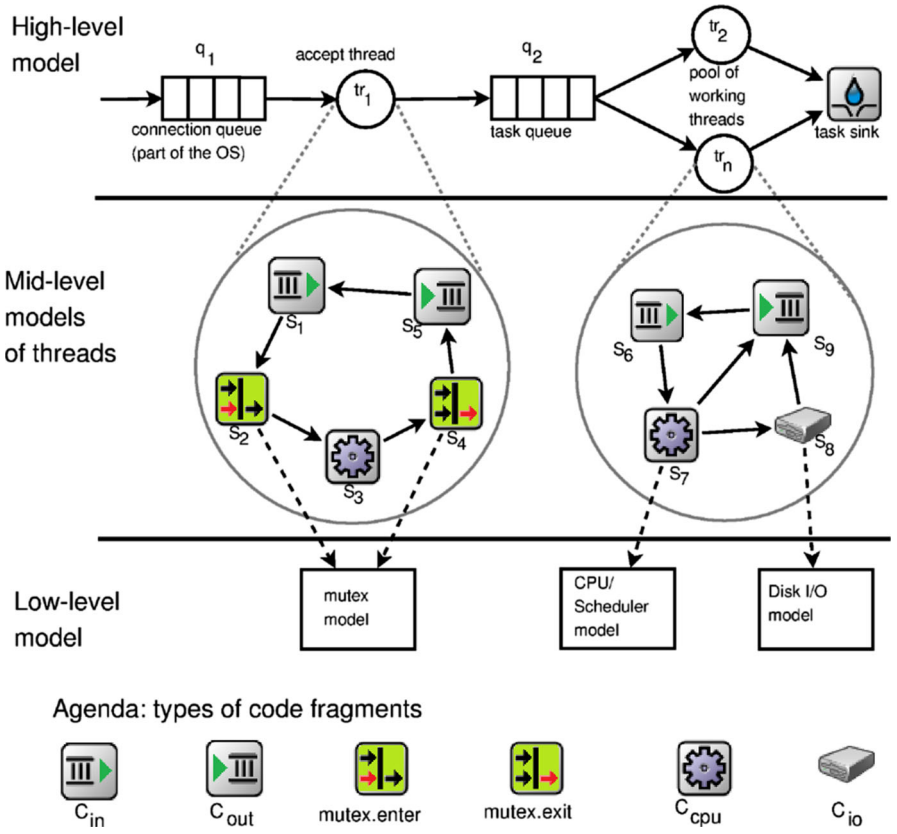

Fig. 1 A model for a web server

working threads $tr_2, \ldots, tr_n$ fetches the task from the queue q_2 and processes the request.

Our model has important differences from the classical queuing networks. First, it does not restrict the structure of the model, the number of service nodes, or the network's parameters. Second, the service nodes are models on their own that simulate program's threads. When the service node receives a task, it calls the model of the corresponding thread to simulate the amount of time necessary to process that task. Finally, the high-level model does not explicitly define service demand for a task; these are implicitly defined by parameters of lower-level thread models. Nevertheless, the high-level model is capable of collecting same performance measures as queuing models, such as response time, throughput, or the number of task in the system through simulation.

3.2 Middle-level model

The middle-level model simulates the *delays* that occur in the program's threads as they process tasks. The thread model is a probabilistic execution graph (PEGs) of

the corresponding thread. Each vertex $s_i \in S$ of the PEG corresponds to a piece of the thread's code—a code fragment. The special vertex s_0 corresponds to the code fragment executed upon a thread start.

A *code fragment (CF)* is a contiguous sequence of instructions that perform one of the following: a synchronization operation, an I/O operation, a CPU-intense computation, or accessing one of the program's task queues. Some examples of code fragments are: a bytecode instruction that unlocks a mutex; a call to a low-level library function that can perform a disk I/O operation, such as `stat()` `libc` function; or a number of sequentially executed functions that perform CPU- or memory-intensive computations. It is important that a code fragment is either executed completely, or not executed at all. The Sects. 4.2 and 4.3 provide examples of code fragments and discuss their detection in detail.

Edges of the PEG represent possible transitions of control flow between the CFs and are labeled with their probability. For each vertex $s_i \in S$ there is a subset of vertices $S_{next} = \{s_k, \ldots, s_m\}$ that can be executed after s_i. The probability that the the CF s_j, $j \in (k \ldots m)$ will be executed after $s_i \in S$ is denoted as $p(s_i, s_j)$, where

$$\sum_{j=k}^{m} p(s_i, s_j) = 1 \tag{1}$$

Probabilities of transitions between all the CFs constitute a mapping $\delta : S \rightarrow P(S)$. For certain CFs the set S_{next} can be empty, such that $S_{next} = \varnothing$. These are *terminal* CFs. After executing a terminal CF a thread stops its execution.

Computations performed by every CF $s_i \in S$ take a certain amount of time to complete. In the terms of the model computations performed by s_i are simulated as introducing a delay with duration τ_i. The duration of the delay τ_i may vary between different invocations of the same CF.

We distinguish three major sources of delays in processing tasks, which correspond to three distinct *classes of code fragments*: I/O code fragments (denoted as c_{IO}) represent I/O operations; synchronization (c_{sync}) CFs represent synchronization operations; computation (c_{CPU}) CFs represent computations and memory operations. In addition, we define c_{in} and c_{out} CFs that communicate with the high-level queuing model. c_{in} CFs fetch tasks from the queues of the upper-level queuing model. As a part of this the thread model can suspend its execution until the request become available. c_{out} CFs send tasks to the upper-level queuing model. In the context of the multithreaded program, c_{in} and c_{out} CFs correspond to operations on the program's shared task queues.

Figure 1 (middle) depicts the mid-level model of a web server. In the model of the accept thread the s_1 CF fetches incoming connections from the queue q_1, s_2–s_4 CFs create a task object, and s_5 sends it into the task queue q_2. In the model of the working thread the s_6 CF fetches the task from the task queue and processes it (s_7–s_8). The working thread verifies that the requested page exists, reads it from the disk, and sends it to the client. Finally, the thread closes the connection and fetches the next task from the queue.

3.3 Low-level model

Execution of each code fragment (CF) results in the delay τ. While the call graph structure $\langle S, \delta \rangle$ does not change between different configurations, *execution times for code fragments can be affected by resource contention*. Resource contention occurs when multiple threads simultaneously attempt to access a shared resource such as the CPU, the disk, or a lock. For example, if the number of working threads that perform CPU-intense computations exceeds the number of physical CPUs, the time required for each thread to finish computations will be higher than if that thread was running alone. Similarly, as more threads compete for a mutex, the waiting time for each of those threads increases. As a result of resource contention, the time delay τ_i for the CF s_i can vary significantly across different configurations of the program and cannot be specified explicitly in the mid-tier thread model.

To accurately simulate the time delays τ that occur due to contention we use lower-level models. The lower-level model simulates the system's shared resources: the CPU, the OS thread scheduler, the disk I/O subsystem, and the set $L = \{l_1, \ldots, l_m\}$ of locks in the program. These models are part of $Q(t)$ – the state of the whole simulation at each moment of time t.

To accurately compute the delay τ_i of each fragment, we augment each code fragment s_i with a set of parameters Π_i, which represent the resource requirements for s_i. When the thread model needs to compute as part of a simulation the τ_i, it calls the corresponding low-level model, passes it the parameters Π_i, and waits for the response. When the lower-level model receives the call, it updates the state $Q(t)$ and simulates the delay τ_i. Once the delay has passed, the lower-level model returns control back to the thread model.

The nature of the parameters Π_i and the actual semantics of interaction between the thread model and the low-level model depends on the class c_i of the code fragment s_i. We describe discovery of parameters Π and provide examples in the Sect. 4.3.2. Below we describe modeling different types of computations in general terms.

Modeling CPU computations CPU computations and memory operations are simulated by the c_{CPU} computation CFs. The parameter of a computation CF $\Pi_{CPU} = \langle CPUtime \rangle$ is the *CPU time* for that fragment. The CPU time is the amount of time required for the computation CF to complete if it would run on a CPU uninterrupted. As $CPUtime$ fluctuates across different executions of s_i, Π_{CPU} is represented as a probability distribution of CPU times $\mathbb{P}^{\Pi}_{CPU}$.

When the thread model has to compute τ for the computation CF, it samples $CPUtime$ from the $\mathbb{P}^{\Pi}_{CPU}$ and calls the CPU/Scheduler low-level model. The CPU/Scheduler model simulates a round-robin OS thread scheduler with equal priority of all the threads. It is a simple queuing model, whose queue corresponds to the queue of "ready" threads in the OS thread scheduler, and service nodes correspond to the cores of a simulated CPU.

Upon receiving the request the CPU/Scheduler model creates a new job with service time $S_{CPU} = CPUtime$ and inserts it into the back of the "ready" queue. Once the service node becomes available, it fetches the job from the queue and introduces a delay equal to $min(CPUtime, OS\ time\ quantum)$. After the delay is expired, the CPU/Scheduler checks if computations are complete for the job. In this case the

CPU/Scheduler deletes the job and notifies the thread model. Otherwise it places the job back into the "ready" queue, where it awaits another time quantum. Multiple core simulations are handled appropriately.

Modeling disk I/O operations I/O operations are simulated using c_{IO} I/O code fragments, whose parameters form a distribution $\mathbb{P}^{\Pi}_{IO}$. Members of this distribution are tuples $\Pi_{disk} = \langle dio_1, \ldots, dio_k \rangle$ of low-level disk I/O operations initiated by that CF. Properties of each I/O operation dio_j include the amount of data transferred and the type of the operation such as "metadata read" or "readahead".

The number k of I/O operations is used to implicitly simulate the OS page cache. It was shown (Feng and Zhang 2008) that after serving a sufficient number of requests (10^4 to 10^5 in our experiments), the cache enters a steady state, where the probability of cache hit converges to a constant. In terms of our model, k follows a stationary distribution, where $k = 0$ indicates a cache hit.

When the mid-level thread model must simulate the I/O CF, it fetches a sample of disk I/O operations $\langle dio_1, \ldots, dio_k \rangle$ from the distribution $\mathbb{P}^{\Pi}_{IO}$ and issues a sequence of calls to the DiskIO low-level model. Here each call represents a corresponding I/O operation $dio_j \in \Pi_{disk}$. If the I/O operation is synchronous (file read or metadata read), the thread model waits for the response from the low-level model. If the operation is asynchronous (readahead) the thread model does not introduce such wait.

The disk I/O model is a queuing model whose queue represents the request queue in the I/O scheduler, and the service node represents the hard drive. The service node delays the job for the τ_{disk}, which is the amount of time necessary for the hard drive to complete the I/O operation. τ_{disk} can vary depending on the locality of the operation (how close are the disk sectors accessed by different requests), the number of requests in the queue, and other factors. Many of these factors are beyond the control of the model. Thus we simulate the τ_{disk} as a conditional distribution $P(\tau_{disk} | dio\_type, dio\_rate, dio\_parallel)$, where

- dio_type: the type of the request: file read, metadata read, readahead;
- dio_rate: the intensity of the I/O workload; measured as the mean interarrival time for the previous N I/O requests (in our experiments typically $N = 20$);
- $dio\_parallel$: the degree of parallelism in I/O workload; measured as the number of distinct threads that initiated the previous N requests.

Our models currently do not explicitly simulate write I/O operations which are normally executed asynchronously. However, in our experiments we observed that unless the application performs a massive amount of writes, the write I/O requests do not have a noticeable impact on the performance of the system. Thus we leave implementation of disk I/O write model as a subject of a future work.

Modeling synchronization operations and queue accesses Synchronization operations are simulated using c_{sync} synchronization code fragments. Parameters of synchronization CFs are defined by the tuple $\Pi_{lock} = \langle l_j, optype, tmout_{sync} \rangle$, where

- $l_j \in L$ is the synchronization construct (lock) that is called;
- $optype$ is the synchronization operation performed. Possible values of $optype$ depend on the type of the lock. For example, the possible values of $optype$ are $\{acquire, release\}$ for a mutex, and $\{await\}$ for a barrier;

– $tmout_{sync} \in (0, \ldots, \infty)$ is the timeout for synchronization operation. By default timeout is $tmout_{sync} = \infty$, which denotes the infinite timeout. Correspondingly, $tmout_{sync} = 0$ denotes the absence of the timeout.

When the mid-level thread model has to simulate τ for the synchronization CF s_i, it calls the lower-level model and passes the parameters Π_i of that CF along with the call.

The lower-level model explicitly simulates behavior of each lock $\{l_1, \ldots, l_m\} \in L$ in the program. Its internal state contains the status of each lock and the list of threads currently waiting on that lock. For example, the model of the mutex contains the state of the mutex (locked or open) and the list of threads waiting on that mutex.

If the synchronization operation cannot be completed within the timeout $tmout_{sync}$, the lock model will return control to the mid-level model and the state of the simulated lock is not changed. For example, if a mid-level model calls the model of the mutex and the parameter $tmout_{sync} = \infty$, then the low-level model will not return to the mid-level model until the simulated lock can be acquired. Alternatively, if $tmout_{sync} = 0$, the low-level model will return immediately, even if the mutex can't be acquired.

We developed separate models for various types of locks such as barriers, semaphores, mutexes, etc. Each lock $l_j \in L$ is described using $\langle ltype, lparam \rangle$ parameters, where $ltype$ is the type of the lock, such as a semaphore, a barrier, or a mutex, and $lparam$ are the type-specific parameters of the lock. For example, the parameter of a barrier indicates the barrier capacity, the parameter of a semaphore is the number of permits, and a mutex has no parameters.

Fetching and sending a task to a queue are simulated by c_{in} and c_{out} code fragments. Their parameters $\Pi_{inout} = \langle qid, optype, tmout_{inout} \rangle$ are the ID of the queue, type of the operation such as $\{fetch, send\}$, and the optional timeout. Discovery of locks, queues and their parameters is presented in the Sect. 4.3.2.

4 Automatic model generation

Constructing the performance model requires collecting the following information about the program automatically:

– The set $q_1, \ldots, q_n$ of queues and buffers used to exchange tasks between program's threads. These correspond to the queues in the high-level model;
– The set $tr_1, \ldots, tr_m$ of threads in the program. Threads correspond to the service nodes of the high-level model;
– The set $Tp_1, \ldots, Tp_k$ of thread pools. The sizes of thread pools are configuration parameters that impact performance;
– Information on interactions between the threads and queues in the program. This corresponds to c_{in}/c_{out} CFs in the middle-level model;
– The computation, I/O, and locking operations in a program (correspond to the set S of CFs) and the sequence of their execution (correspond to transition probabilities δ);
– The parameters Π of each CF, required to model delays τ;
– The set L of locks in the low-level model, their types, and parameters Π_{lock}.

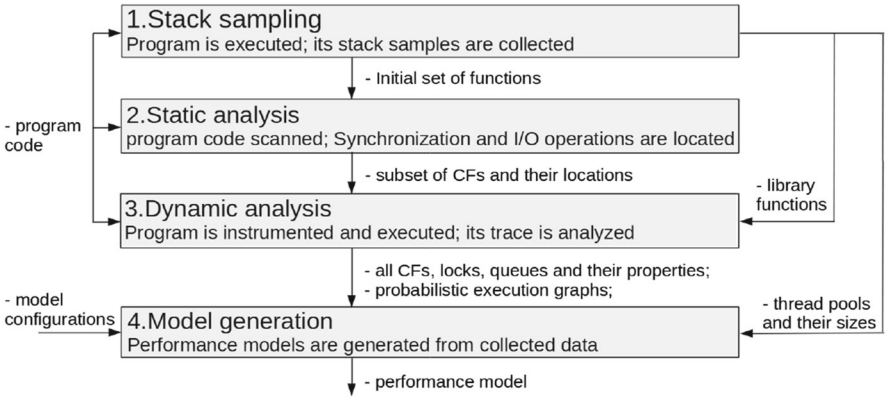

Fig. 2 Model creation stages and intermediate results

The required data are collected using a combination of static and dynamic analysis. During data collection, the program is executed in a single representative configuration, in which $\langle S, \delta \rangle$ and Π would be similar to the $\langle S, \delta \rangle$ and Π of a larger set of configurations for which the program's performance should be predicted. This requires the usage scenario for the program (e.g. the probabilities of accessing particular web pages for a web server or the input dataset for a scientific application) to be similar across the configuration space.

We collect the required data in four stages (see Fig. 2). Each stage saves intermediate results into files that are used as input to subsequent stages.

First, the program is executed and its call stack is sampled. The stack samples are used to detect thread groups and libraries in the program. Second, a static analysis of the program is performed. During this stage c_{sync}, c_{in}, c_{out}, and c_{IO} CFs are detected. Third, the program is instrumented and executed again with the same configuration as the initial run. The instrumentation log is used to detect program-wide locks and queues, properties Π of code fragments, and to build the probabilistic call graphs $\langle S, \delta \rangle$ of the program's threads. Finally, the collected information is used to build a performance model. *All these operations are performed automatically.*

Below we describe these stages in more details.

4.1 Collecting stack samples

During the stack sampling stage our framework finds thread pools, and functions that are called from multiple locations which we call *library functions*. Identifying libraries is essential for generating correct probabilistic call graphs (see Sect. 4.3.1).

As the program is being executed, the framework periodically takes "snapshots" of the call stack of the running program, which are merged to build a *call trie* of the program. In a call trie each leaf node contains the code location being executed, which includes the name of a function or a method and a line number. The non-leaf nodes provide a call stack for that code location. For each leaf the framework maintains the

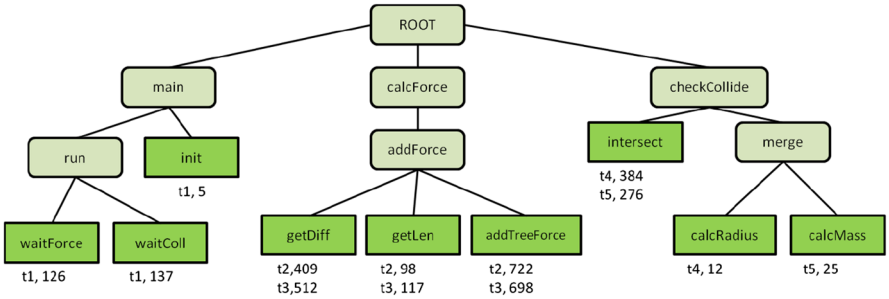

Fig. 3 An example of a call trie

list of pairs $\langle tr_1, ex_1\rangle, \ldots \langle tr_n, ex_n\rangle$, where the ex_i is the number of executions of that code location by the thread tr_i.

An example of the call trie for a multithreaded program is depicted at the Fig. 3. Here the method `waitForce()` was called by the method `run()`, while `run()` itself was called by the method `main()`. The `waitForce()` method was always executed by the thread tr_1; the total number of executions of that method detected during the stack sampling is $ex_1 = 126$. Similarly, the method `getLen()` was executed by threads tr_2 and tr_3 98 and 117 times respectively.

The call trie is used to detect thread pools $Tp_1, \ldots, Tp_k$ in the program. A thread pool $Tp_l = \langle tr_i, \ldots, tr_j\rangle$ is a tuple of one or more threads $tr_i, \ldots, tr_j, j \leq m$ that have same functionality and can process tasks in parallel. $Tp_1, \ldots, Tp_k, k \leq m$ constitute the set of all thread pools present in the program.

We detect thread pools from the call trie in two steps. During the first step our framework forms a set of candidate thread pools $Tp_1, \ldots, Tp_q, , q \geq k$ that can be possibly found in the program. During the second step the framework refines the set of candidate thread pools into the final set of thread pools $Tp_1, \ldots, Tp_k$.

During the first step our framework initially creates a map $\mathbb{T}$ whose keys are the candidate thread pools, and values are the total execution counts for these candidate pools. Then the framework iterates through the leaf nodes of the call trie. For each leaf node the framework retrieves a tuple $Tp_l = \langle tr_i \ldots tr_j\rangle$ of threads that executed the node along with the total number of executions $Ex_l = \sum(ex_i, \ldots, ex_j)$. If $\mathbb{T}$ does not contains the tuple Tp_l, the pair $\langle Tp_l, Ex_l\rangle$ is inserted into $\mathbb{T}$. Otherwise the number of executions for the existing tuple is increased by Ex_l. As a result of the first step, the keys of the map $\mathbb{T}$ form the set $Tp_1, \ldots, Tp_q$ of candidate thread pools.

In our example $\mathbb{T}$ contains following information:

- $Tp_1 = \langle tr_1\rangle$, $Ex_1 = 5 + 126 + 137 = 268$
- $Tp_2 = \langle tr_2, tr_3\rangle$, $Ex_2 = 409 + 98 + 722 + 512 + 117 + 698 = 2556$
- $Tp_3 = \langle tr_4, tr_5\rangle$, $Ex_3 = 384 + 276 = 660$
- $Tp_4 = \langle tr_4\rangle$, $Ex_4 = 12$
- $Tp_5 = \langle tr_5\rangle$, $Ex_5 = 25$

However, the data collected by the stack sampling is not guaranteed to be accurate. It is possible that some of the executions of a method by the thread were not detected during the stack sampling, which results in a number of "spurious" thread

pools detected at the first stage. In our example it is likely that `calcRadius` and `calcMass` methods were also executed by threads tr_5 and tr_4 correspondingly. But these executions were either too infrequent or too short to be detected by the stack sampling. This resulted in detection of "spurious" thread pools Tp_4 and Tp_5.

During the second step spurious thread tuples in $\mathbb{T}$ are detected and merged with the correct ones. The tuple $\langle Tp_1, Ex_1 \rangle$ is considered a spurious one and merged with $\langle Tp_2, Ex_2 \rangle$ if and only if all threads in Tp_2 also present in Tp_1 and $Ex_1 \gg Ex_2$. The resulting tuple is formed as $\langle Tp_1, Ex_1 + Ex_2 \rangle$. After merging, the remaining tuples $Tp_1 \dots Tp_m \in \mathbb{T}$ represent the thread pools detected in the program.

In the example depicted at the Fig. 3, tuples Tp_4 and Tp_5 are merged into Tp_3 because $Ex_3 \gg Ex_4$ and $Ex_3 \gg Ex_5$. The resulting set of thread pools is $Tp_1 = \langle tr_1 \rangle$, $Tp_2 = \langle tr_2, tr_3 \rangle$, $Tp_3 = \langle tr_4, tr_5 \rangle$.

Stack samples are also used to identify program's libraries. The knowledge of libraries is necessary to generate a semantically correct performance model. For every function or method f the framework generates the set of functions $\langle f_1, \dots, f_{ncall} \rangle$ that called f. If the number of callees $ncall > 1$, f is added to the set of *library functions*. Although the stack sampling may not detect some rarely executed library functions, this does not affect correctness of our models.

4.2 Static analysis

During the static analysis our framework scans the code of the program and detects c_{sync}, c_{IO}, c_{in} and c_{out} CFs. It also detects the creation points of locks and queues in the program, as a prerequisite for the dynamic analysis.

The static analyzer represents the program as a dependency graph. The vertices of this graph correspond to functions and methods in the program (both called *"function"* herein) and classes. The edges are code dependencies (e.g. the function A calls the function B) and data dependencies (e.g. the function A refers the class B or creates the instance of B) between these functions. The transitive closure of all the vertices in the dependency graph represents all the code that may be executed by the program.

The static analyzer traverses the dependency graph, starting from the functions discovered during the stack sampling. It scans the code of the functions, searching for the specific constructs that represent c_{sync}, c_{IO}, c_{in} and c_{out} CFs. In the process the analyzer searches for references to other functions and methods, that are subsequently loaded and analyzed.

There are numerous ways to implement synchronization and queue operations in a program. Practically all the modern programming languages such as C, C++ or Java provide low-level primitives to implement threading and synchronization. These primitives are built around the concept of the mutexes and condition variables (Hoare 1974). However, programmers rarely design and think of their programs in the terms of mutexes and condition variables. Instead, programmers design their programs in terms of higher-level locks such as semaphores, barriers, read-write locks, or producer-consumer queues. Similarly, we simulate the semantics of thread interaction in the program in terms of these high-level locks.

Unfortunately, there can be numerous ways to design and implement high-level locks using low-level primitives. As a result, detecting c_{in}, c_{out} and synchronization CFs and determining their operation types $optype$ may require complex analysis that is very hard to automate.

However, manually implementing high-level synchronization constructs is a work-intense and error-prone task for most programmers. The resulting implementations often have inferior performance and are prone to bugs. To facilitate work of developers, most of modern programming languages provide standard libraries of concurrent constructs: semaphores, barriers, synchronization queues, thread pools and other means for thread interaction. Examples of such libraries are the `java.util.concurrent` package for Java, the `System.Threading` namespace in C#, and `boost` threading library in C++. Using standard implementations of locks and queues instead of constructing them from low-level synchronization primitives is a recommended way to developing concurrent applications.[3]

From the standpoint of building performance models, using known implementation of high-level locks and queues greatly simplifies the analysis of the program. Implementing thread interaction using a set of standard constructs allows program analysis to accurately identify queues Q and locks L and in the program, determine their types and parameters Π_{lock}, and discover synchronization operations that involve these locks. Thus in the current study we concentrate on building models of programs that employ standard implementation of locks and queues to implement thread interactions.

The analyzer considers calls to specific functions that perform synchronization operations and access program's queues as c_{sync}, c_{in}, and c_{out} CFs appropriately. Typically, these are the functions that constitute the API of the corresponding thread and locking library. The class of the CF and the type of synchronization operation $optype$ are inferred from the name and the signature of the called function. For example, in a Java program the call to the `Semaphore.acquire(int permits)` is considered as a c_{sync} CF whose type is $optype$="Semaphore_acquire". Similarly, the call to the `Semaphore.release()` method is a c_{sync} CF whose type is $optype$="Semaphore_release".

The analyzer also tracks low-level synchronization primitives, such as monitors, mutexes, and synchronized regions. These constructs are often used to implement simple synchronizations. Our models simulate these constructs explicitly as c_{sync} CFs. However, when the combination of low-level primitives is used to implement a high-level lock, the probabilistic execution graph (PEG) may not be able to capture the deterministic behavior of such lock. Consider a custom implementation of a cyclic barrier that maintains the counter of waiting threads. When the thread calls the barrier, the program checks the value of the counter. If the value of the counter is less than the capacity, the calling thread is suspended; otherwise the program wakes up all the waiting threads. In the PEG this behavior will be reflected as a fork with the probability of lifting the barrier equal to 1/(barrier capacity). As a result, in certain cases the model will lift the barrier prematurely, and in other cases it will not lift the barrier when it is necessary.

[3] http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/overview.html.

c_{in}/c_{out} CFs are detected in the same way as synchronization CFs. The only difference is that the analyzer tracks a different set of API functions or methods, which represent operations on the program's queues. For example, the call to `ArrayBlockingQueue.take()` is a c_{in} CF, and the call to `ArrayBlockingQueue.put()` is a c_{out} CF.

The analyzer also tracks calls to the constructors and initializers of locks and queues. These calls do not directly correspond to the c_{sync} CFs, but they are used to detect queues and locks in the program and retrieve their parameters during the dynamic analysis.

c_{IO} code fragments are discovered in a similar manner. The static analyzer tracks API functions that can perform disk I/O. Calls to the functions that may access the file system metadata, such as `File.exists()` Java method or `stat() libc` function, are considered as I/O CFs. Similarly, the bodies of low-level functions that perform file I/O, such as native methods of the `FileInputStream` Java class, are also considered as I/O CFs.

4.3 Dynamic analysis

The purpose of dynamic analysis is to identify c_{CPU} CFs, the parameters Π of locks and CFs, and the probabilistic call graphs $\langle S, \delta \rangle$ of the program's threads.

The dynamic analyzer instruments the program and runs it again in the same configuration as the initial stack-sampling run. Each CF detected during the static analysis is instrumented with two probes. A *start probe* is inserted immediately before the CF, and an *end probe* is inserted right after the end of the CF. Each probe is identified by the unique numeric identifier (probe ID).

Probes report the timestamp, the probe ID, and the thread ID. For CFs corresponding to a function call, the start probe reports function's arguments, and the end probe reports the return value. For method calls probes also report the reference to the called object, if relevant. This information is used to obtain parameters of c_{sync}, c_{in}, and c_{out} CFs.

During its execution the instrumented program generates the sequence of probe hits on a per-thread basis, which constitute a *trace* of the thread. Two coincident probe hits in the trace form a pair $\langle$*start probe ID, end probe ID*$\rangle$. Every such pair represents an execution of a single code fragment.

The $\langle$*start probe ID, end probe ID*$\rangle$ pairs are "overlapping" in the trace, so the end probe ID of one pair becomes the start probe ID of the next pair. Thus executions of c_{IO}, c_{sync}, c_{in}, and c_{out} CFs in the trace are interleaved with pairs of probe IDs. These pairs, which represent computations performed between executions of c_{IO}, c_{sync}, c_{in}, and c_{out} CFs, correspond to c_{CPU} CFs.

We illustrate the analysis of the program with an example. The Fig. 4 contains a snippet of a Java program that compares images. The program fetches an object `img1` that represents an image (corresponds to a task in the terms of our model) from a shared blocking queue `tasksQueue`. Then the program enters a synchronized region guarded by the `objLock` object, loads a reference image `imgRef` from the file, and exits the region. Next, the program compares `img1` image against the `imgRef` and

```
Line no.
 1        FileInputStream fileRef = new FileInputStream("reference.jpeg");
 2        FileInputStream fileLog = new FileInputStream("compareLog.bin");
 3        MyImage imgRef = null;
 4        MyImage img1 = tasksQueue.poll(100, TimeUnit.MILLISECONDS);
 5        synchronized(objLock) {
 6           byte[] bytesRef = fileRef.readBytes();
 7           imgRef = new MyImage(bytesRef);
 8        }
 9        boolean match = Image.compare(img1, imgRef);
 10       byte[] bytePrevComparisons = fileLog.readBytes();
           ...
```

Fig. 4 A code snippet that performs image comparison

Table 2 A corresponding fragment of the trace

ProbeID	Timestamp (ns)	ObjectID	Arguments/ return value	Comments
10	11345231	7683745	100, 5	#Fetch the img1 object (line 4)
11	11391326	7683745	4387459	
27	11391365	87235467		#Enter a synch. region (line 5)
28	11392132			
10205	11392159	1872565		#Read a reference image (line 6)
10206	19756012	1872565		
6	19873872	87235467		#Exit a synch. region (line 8)
7	19873991			
10205	19923752	32748998		#Read a comparison log (line 10)
10206	25576572	32748998		
...				

loads a comparison log from another file in order to append the result to it (writing the result to the log is not shown in the listing because of space concerns).

The trace for a program is shown at the Table 2. Here the CF $\langle 10, 11\rangle$ is a c_{in} CF. The object ID=7683745 recorded by the probe 10 identifies the `tasksQueue` queue, the first argument denotes the timeout of 100 ms, and the second argument denotes the unit of measurement (msec). The probe 11 reports the return value 4387459, which is an ID of the retrieved task `img1`. $\langle 27, 28\rangle$ and $\langle 6, 7\rangle$ are the synchronization CFs corresponding to the entry and exit from the synchronized region. The object ID=87235467 identifies the monitor associated with that region (represented by the `objLock` object in the program). Two instances of $\langle 10205, 10206\rangle$ I/O CF correspond to two unrelated file read operations. Their object IDs 1872565 and 32748998 identify

the instances of file objects `fileRef` and `fileLog` correspondingly. Pairs ⟨11, 27⟩, ⟨28, 10205⟩, ⟨10206, 6⟩, and ⟨7, 10205⟩ are the computation CFs.

4.3.1 Construction of probabilistic execution graphs

A naïve approach to generating the probabilistic execution graph (PEG) for a thread is to treat the set $s_1 \ldots s_n$ of CFs discovered in the trace as the set S of nodes in the PEG. For each node $s_i \in S$ the subset $S_{next} = \{s_k, \ldots, s_m\}$ of succeeding nodes is retrieved, along with the numbers of occurrences of the pairs $(s_i, s_k), \ldots, (s_i, s_m)$ in the trace. The probability of transition from the node s_i to s_j, $j \in (k \ldots m)$ is calculated as

$$p(s_i, s_j) = \frac{count(s_i, s_j)}{\sum_{l=k}^{m} count(s_i, s_l)} \tag{2}$$

Probabilities of transition for every pair of nodes constitute the mapping $\delta : S \rightarrow P(S)$ in the mid-tier model.

However, the naïve approach results in problems when building execution graphs for real-world applications. It may not represent calls to the program's libraries correctly and generates overly complex PEG. To become practical, this approach must be improved.

Correct representation of library calls Distinct execution paths in the program must be represented as non-intersecting paths in the PEG, so that the control flow in the model will not be transferred from one such path to another. However, if these execution paths call a library function containing a code fragment, the instrumentation would emit the same probe IDs for both calls, which correspond to executing the same CF. As a result, distinct execution paths will be connected by the common node in the PEG. During the simulation the thread model may "switch" from one execution path to another unrelated execution path, which is semantically incorrect.

For example, according to the trace shown on the Table 2 the program enters the synchronized region, reads data from a file, exits the synchronized region, and performs another unrelated file read. The "ground truth" call graph has no loops or branches (see Fig. 5, top). However, both I/O operations will eventually call the same readBytes()

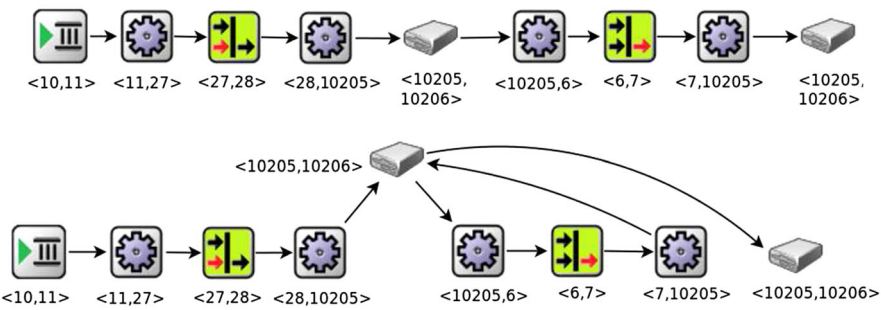

Fig. 5 *Top* The ground truth PEG from the thread trace. *Bottom* The incorrect PEG generated from the trace that contains a library call

I/O API that contains an ⟨10205, 10206⟩ I/O CF. As a result, the generated PEG will contain a loop in it (see Fig. 5, bottom). While simulating this loop the model may not exit the synchronized region, or may attempt exiting it multiple times. In both cases the behavior of the model will be incorrect.

To address this problem the dynamic analyzer represents separate calls to the library CFs as separate PEG nodes using the node splitting technique described in Reiss and Renieris (2001). For every CF located within one of the program's libraries, the analyzer adds a context information describing the origin of the call to that library.

This information is obtained by instrumenting calls to the library functions discovered during the stack sampling (see Sect. 4.1). An entry *library probe* is inserted before every call to a library function; an exit library probe is inserted after such call. As the analyzer scans the trace, it maintains a call stack of library probes. When the entry library probe is encountered in the trace, its ID is added into the stack. This ID is removed from the stack when the corresponding exit probe is detected. When the analyzer detects the CF, it adds the sequence of library probe IDs present in the stack as the prefix of that CF ID. As a result, calls to the library CFs that originate from different locations in the program are represented as separate nodes in the PEG.

For an example, consider that entry/exit library probes 500/501 and 502/503 were inserted into the program, so the resulting sequence of probe IDs in the trace is 10, 11, 27, 28, 500, 10205, 10206, 501, 6, 7, 502, 10205, 10206, 503. The corresponding sequence of CF is ⟨10, 11⟩, ⟨11, 27⟩, ⟨27, 28⟩, ⟨28, 10205⟩, ⟨500, 10205, 10206⟩, ⟨10206, 6⟩, ⟨6, 7⟩, ⟨7, 10205⟩, ⟨502, 10205, 10206⟩, which is consistent with the ground truth PEG.

Reducing the complexity of the model Naïve approach results in an unnecessary complex PEG, consisting of thousands of CFs (see Table 4). Such complex models have low performance and are hard to analyze, so they must be simplified.

According to the naïve approach, all the computations between c_{IO}, c_{sync}, c_{in}, and c_{out} CFs are represented as c_{CPU} CFs, even if their impact on performance is negligible. Similarly, every synchronization region is represented as a pair of CFs, even if it is very short and never becomes contended in practice.

In our example the trace contains four computation CFs. Out of these CFs only two correspond to actual code constructs in the program: ⟨10206, 6⟩ represents creating the `imgRef` object (line 7, execution time 117,860 ns) and ⟨7, 10205⟩ represents image comparison (line 9, execution time 49,761 ns). ⟨11, 27⟩ and ⟨28, 10205⟩ are artifacts of the analysis. Such "spurious" CFs have zero impact on the performance. Their execution times are very low—39 and 27 ns correspondingly.

Overly complex models have major problems. First, simulating spurious CF requires more time than executing them. As a result, performance of the model that has many such CFs will suffer greatly. For example, unoptimized models of Tomcat (see Sect. 5.2) ran slower than the actual program. Second, such model is hard to understand and debug. While implementing PERSIK we often had to analyze and debug the resulting models. Debugging PEGs consisting of thousands of nodes was nearly impossible.

To simplify the model we remove all the *insignificant CFs* that have negligible impact on the program's performance. Model optimization is performed in two steps. The first step is finding phases in the program's execution that do not affect performance

measurements and excluding these phases from modeling. The second step is analysis of the remaining CFs to eliminate those that do not have a noticeable impact on performance.

During the first step the whole time line of the program's execution is split into three phases: the startup phase, when the program doesn't process tasks yet; the work phase, when the program processes tasks; and the shutdown phase, when the program doesn't process tasks any more. Finding phases is easy for programs that handle external requests, such as servers. A timestamp marking the beginning of the work phase is recorded before issuing the first request, and the end timestamp is recorded after the last request is complete. If startup or shutdown phases cannot be easily defined for a program, we assume these phases are absent in the trace.

The model doesn't simulate program's performance during the startup and shutdown phases. Among all CFs executed during the startup phase, only the CFs that are required to build a semantically correct model (c_{in}, c_{out}, and c_{sync} CFs that perform complex synchronization operations, such as awaiting on the barrier) are incorporated into the model. Remaining CFs are considered as insignificant. All the CFs executed during the shutdown phase are considered as insignificant.

During the second step the *insignificant* CFs executed during the work phase are detected and removed from the model. The following CFs are considered as insignificant:

- Non-contended synchronized regions. A synchronized region is non-contended if the mean time required to enter that region is comparable with the instrumentation overhead;
- Computation CFs whose summary CPU times amounts to less than $t\%$ of the overall CPU time for the thread;
- I/O CFs whose total number of I/O operations and summary data transfer amounts to less than $t\%$ of data transferred by the thread.

Setting t between 3 and 5% allows shrinking the PEG by 50–70% without noticeable impact on the accuracy.

Accounting for determinism in the program behavior Some programs express deterministic behavior that is difficult to represent accurately using a probabilistic model. This deterministic behavior must be addressed in the model in order to obtain accurate prediction.

First, the execution flow of a thread may take different paths depending on the availability of the task in the queue. For example the program may attempt to fetch from the blocking queue and impose a timeout for the operation. Depending on whether the request was fetched successfully or if the fetch operation has timed out, the program may execute a different set of code fragments.

To account for this we condition the execution flow in the PEG based on the result of a queuing operation. Namely, the analyzer inserts "virtual" nodes after each c_{in} node in the PEG. The c_{in}^{fetch} virtual node is executed when the c_{in} CF was able to fetch the task from the queue. $c_{in}^{nofetch}$ node is executed if c_{in} did not fetch the task and exited by the timeout.

Second, representing loops as cycles in a PEG may affect the model's accuracy (Koziolek et al. 2006). If a loop that performs exactly n iterations is represented as a

cycle in a PEG, then the number of iterations X for that cycle will not be a constant. It can be shown that X will rather be a random variable that follows a geometric distribution with mean n and a probability mass function $Pr(X = k) = \frac{1}{n} \cdot (1 - \frac{1}{n})^{k-1}$. In most cases this representation has a minor effect on the prediction accuracy. However, if the program's performance y strictly follows the function $y = f(n)$, the predicted performance y' will be a function of a random variable $y' = f(X)$, whose parameters (mean, standard deviation) may differ noticeably from y.

In our experiments such mispredictions occurred when a loop was used to populate the program's queues with tasks. For an example, consider a program with $O(N^2)$ runtime complexity, where N is the number of tasks (the size of the input). Assuming $N = 5$ and length of iteration 1 ms, the average running time of the program will be 25 ms. However, if the loop that populates program's queues with input tasks is modeled as a cycle in the PEG, then the total number of tasks actually generated by the model will follow a geometric distribution with $Pr(N = k) = 0.2 \cdot (0.8)^{k-1}$ and mean $\overline{N} = 5$. The predicted average running time will be 45 ms, which corresponds to the mean prediction error $\overline{\varepsilon}(T) = 0.80$.

To address this issue the dynamic analyzer detects loops that populate task queues using the algorithm in Moseley et al. (2007). The number of iterations in such loops is simulated exactly. This ensures that the model will populate the queue with the right number of tasks.

4.3.2 Retrieving parameters of code fragments

The analyzer retrieves parameters of the model's constructs from the trace.

Parameters of locks and task queues Parameters of locks and queues are obtained from the arguments passed to constructors and intializers corresponding to these locks and queues and from their return values.

The lock type $ltype$ is inferred from the signature of the constructor/intializer of that lock during the static analysis (see Sect. 4.2). The type-specific parameters $lparam$ are retrieved from the values of arguments passed to that constructor. For example, in a Java program the capacity of the barrier is specified by the value `parties` argument of the `CyclicBarrier(int parties)` constructor. Finally, the lock ID lid is obtained from the reference to the lock returned by the constructor; it uniquely identifies each lock $l_i \in L$.

Queues and their parameters are obtained in the same manner. For example, the capacity of the queue is specified as the argument of the `ArrayBlockingQueue (int capacity)` constructor.

Parameters of c_{sync}, c_{in}, and c_{out}CFs Parameters of these CFs are also obtained from the arguments passed to functions and methods operating on locks and queues, and from their return values. The ID of the called lock lid for a c_{sync} CF is obtained from the reference to the lock; it is matched to the lid returned by the lock constructor/initializer. The type of synchronization operation $optype$ was inferred from the signature of the called function earlier during the static analysis. The operation timeout $tmout_{sync}$ is retrieved from the arguments passed to the function. Parameters of the c_{in}/c_{out} CFs are obtained in the same manner.

Some low-level synchronization operations, such as an entry/exit from a synchronized block, might not call functions or methods. $optype$ for such operation is obtained by analyzing the corresponding instruction in the program. lid is obtained from the reference to the associated monitor.

c_{CPU} CFs The parameter of the c_{CPU} CF is the distribution $\mathbb{P}^{\Pi}_{CPU}$ of CPU times $CPUtime$. Generally $CPUtime$ for a code fragment can be obtained as a difference $CPUtime^{end} - CPUtime^{start}$, were $CPUtime^{start}$ is the thread CPU time measured before executing the CF and $CPUtime^{end}$ is the thread CPU time measured after executing the CF. Here thread CPU time denotes the amount of time the CPU was executing instructions of that thread.

$CPUtime$ can be accurately measured when the execution time of a thread can be determined. When this is not the case, $CPUtime$ is measured as the difference between the timestamps of start and end probes of the CF, substituting clock time for CPU time. However, in order to use the latter approach we need to avoid configurations where CPU congestion is likely when defining model parameters.

c_{IO} CFs. The parameters of the c_{IO} CF are the number k and properties (the type of I/O operation and the amount of data transferred) of low-level disk I/O requests $\{dio_1, \ldots, dio_k\}$ initiated by that c_{IO} CF. This request-specific data can be retrieved only from the OS kernel. We used the blktrace[4] to retrieve the log of all kernel-mode disk I/O operations initiated by the program.

Generally, the timestamps and thread IDs in the kernel-mode I/O log might not match the timestamps and thread IDs in the instrumentation log. This makes associating low-level I/O requests with execution of I/O code fragments in the program difficult.

To match blktrace log to the instrumentation log the dynamic analyzer uses cross-correlation—a technique used in signal processing (Stein 2000). The cross-correlation $(f \star g)[t]$ is a measure of similarity between signals f and g, where one of the signals is shifted by the time lag Δt. The result of a cross-correlation is also a signal whose maximum value is achieved at the point $t = \Delta t$. The magnitude of that value depends on similarity between f and g. The more similar are those signals, the higher is the magnitude of $(f \star g)[\Delta t]$.

The analyzer represents sequences of I/O operations obtained from the kernel-mode trace and user-mode trace as signals taking values 0 (no I/O operation at the moment) and 1 (an ongoing I/O). It generates user I/O signals $U = \{u(t)_1 \ldots u(t)_N\}$ for each user-mode thread obtained from the program trace, and kernel I/O signals $B = \{b(t)_1 \ldots b(t)_M\}$ for each kernel-mode thread from the blktrace log. The analyzer discretizes those signals with a sampling interval of 1 ms.

Figure 6 depicts the cross-correlation between signals $u(t)$ and $b(t)$. The cross-correlation signal $(u(t) \star b(t))[t]$ reaches its maximum value at the point $\Delta t = 324$, which means that the user signal $u(t)$ is shifted forwards by $\Delta t = 324$ ms with relation to the kernel signal $b(t)$.

The dynamic analyzer matches user to kernel I/O signals using a greedy iterative procedure. For each pair of signals $\langle u(t)_i \in U, b(t)_j \in B \rangle$ the analyzer

[4] http://linux.die.net/man/8/btrace.

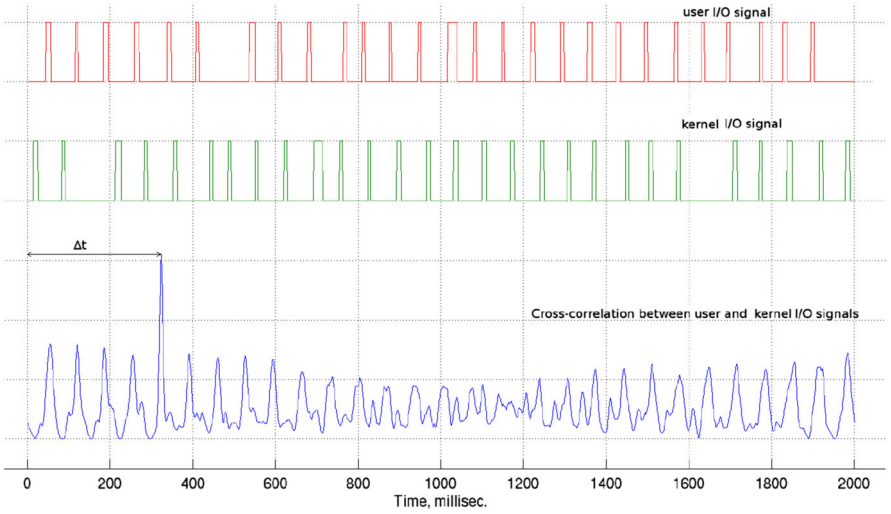

Fig. 6 Cross-correlation between user-mode and kernel-mode I/O logs

computes a cross-correlation signal $xcorr_{ij} = b(t)_i \star u(t)_j$ and the value $\Delta t_{ij} = \arg\max_t(xcorr_{ij})$. The user signal $u(t)_i$ matches the kernel signal $b(t)_j$ if the maximum value of the cross-correlation signal $xcorr_{ij}[\Delta t_{ij}]$ is the highest across the signal pairs.

Next the analyzer aligns user and kernel-mode traces by subtracting the Δt from the timestamps of the user-mode trace. Finally, the kernel-mode I/O operations are associated with the user-mode states. Each kernel mode I/O operation dio_j is described as a time interval $[t^b_{start}, t^b_{end}]$ between its start/end timestamps. Similarly, invocations of the user mode I/O CFs are described as time intervals $[t^u_{start}, t^u_{end}]$. The kernel-mode I/O operation dio_j is considered to be caused by the user-mode I/O CF if the amount of intersection between their corresponding time intervals is maximal across all the I/O CFs in the trace. Correspondingly, a sequence $dio_j \ldots dio_{j+k}$ of low-level I/O operations associated with the execution of the user-mode CF are considered to be parameters $\langle dio_1 \cdots dio_k \rangle \in \mathbb{P}^{\Pi}_{disk}$ of that CF. A user-mode I/O CFs that does not intersect any kernel-mode I/O operation is considered as a cache hit ($k = 0$).

4.4 Constructing the performance model

We implemented our methodology as a toolset for automatically building models of Java programs. The toolset consists of two major parts: PERformance SImulation Kit (PERSIK) framework for implementing performance models, and the ModelGen tool for automatically generating the PERSIK model of a given program. The source code of our toolset and examples of generated models are available at.[5]

[5] https://sourceforge.net/projects/persik/.

The PERSIK framework is based on the OMNeT++ toolset for implementing discrete event models.[6] A PERSIK model consists of interconnected *modules* that communicate using messages (Varga and Hornig 2008). *Simple modules* form a set of "building blocks" that is common for models of all the programs. We implemented a variety of simple modules that simulate different types of CFs, queues, locks, hardware; and also modules that perform service functions such as collecting simulation results. Each simple module has a set of parameters, which correspond to parameters Π of the corresponding entity of the formal model. Simple modules are grouped into *compound* modules, which enables constructing hierarchical models.

PERSIK models generally follows the architecture of three-tier models described in the Sect. 3. The main difference is that PERSIK models have two tiers instead of three. The upper-level PERSIK model contains the higher-level queuing model of the system. It also contains models of locks, I/O subsystem, and the CPU/Scheduler model, which occupy the lowest level of the formal model. The lower-level PERSIK model implements the models of working threads, which correspond to the middle tier of the formal model. The architectural differences between the formal and PERSIK models intend to facilitate the actual implementation of the models. These differences are of a cosmetic nature and do not violate the semantics of our formal model.

PERSIK models are automatically generated by ModelGen. ModelGen relies on ASM[7] toolset for bytecode analysis and instrumentation. ModelGen implements program analysis described in Sects. 4.1, 4.2 and 4.3. The result of the program analysis is a set of text and xml files, which contain all the information required to generate the model: the list of threads, thread pools, and queues in the high-level model; the set S of CFs, their classes and properties Π; transition probabilities δ; the set of locks L and their properties Π_{lock}. ModelGen translates this information into the PERSIK model of the program. Elements of the formal model strictly correspond to the modules and and parameters of the PERSIK model, which makes such translation straightforward.

To start using the model the analyst must specify the model's configuration parameters (the numbers of threads in the thread pools, intensity of the workload, sizes of the queues, the numbers of CPU cores etc). The analyst must also specify what performance data should be collected. The model can provide performance data for CFs (execution time τ), for a group of CFs (e.g. a processing time of the task by the thread), or for the whole program (e.g. throughput or a response time). These are the only manual actions performed during the model construction.

5 Model verification

In this section we present experimental evaluation of our methodology for automatic generation of performance models. We used our tool to automatically build performance models of a variety of multithreaded programs and evaluated their prediction accuracy.

[6] http://www.omnetpp.org/.

[7] http://asm.ow2.org/.

To estimate the accuracy of our predictions we built the model of each program from one configuration and used it to predict performance in a set of other configurations. Then we measured actual performance of the non-instrumented program in same configurations. To get reliable measurements we performed three runs of both the actual program and its model in each configuration. The mean values of measured and predicted performance metrics were used to calculated the relative error ε of the model:

$$\varepsilon = \frac{|\overline{measured} - \overline{predicted}|}{\overline{measured}} \tag{3}$$

Performance metrics we predict include program-wide metrics, such as response time or throughput of the program, and also utilization of the computation resources, such as a hard drive or a CPU.

Below we describe our simulations in detail. First, we present results for modeling various small- to medium-size programs. These results demonstrate that our simulation framework is capable of predicting performance of various programs that use different synchronization constructs and hardware resources. Second, we present results for large industrial programs. These results demonstrate that out approach can be used to build accurate models of large, industrial-grade multithreaded programs.

5.1 Modeling small- to medium-size programs

We built models of the following applications: Raytracer (a 3D rendering program), Montecarlo (a financial application), Moldyn and Galaxy (scientific computing applications), and Tornado (a Web server). Raytracer, Montecarlo and Moldyn are parts of the Java Grande benchmark (Bull et al. 1999) suite. Although relatively small in size, these programs express functionalities peculiar to a wide range of multithreaded programs. They implement thread interaction in different ways and use a great variety of synchronization mechanisms to enforce a correct order of computations across multiple threads.

We used two hardware configurations for our experimentation. Configuration Config I is a PC equipped with the Intel Q6600 quad-core CPU, 4GB RAM, and 250 GB HDD. The computer was running Ubuntu Linux OS. Configuration Config II is a PC equipped with 2 eight-core AMD Opteron CPUs (total 16 CPU cores) and 64 GB RAM. The computer was running Debian Linux OS. All the programs, except Tornado, were run in both Config I and Config II configurations. Tornado, as an disk I/O-heavy application, was run only in the Config I configuration.

Table 3 present a summary on these programs and their models. Below we briefly describe these programs, along with results of their simulations.

Raytracer program renders the image at a given resolution using a ray tracing algorithm. The rendering is parallelized across a pool of working threads; each thread renders a given row of pixels and thus corresponds to a "task" in the terms of the formal model. These tasks are stored in a synchronized queue that is initialized upon the start of the program.

In our experiments Raytracer rendered a scene containing 64 spheres. The overall time required to render the frame is the most important performance metric of Ray-

Table 3 Small- to medium-size programs and their models

	Raytracer	Montecarlo	Moldyn	Galaxy	Tornado
Size, lines of code	1468	3207	1006	2480	1705
Number of probes	16	18	30	72	40
Number of CFs	43	17	72	124	88
Number of nodes in the model	25	24	46	59	36

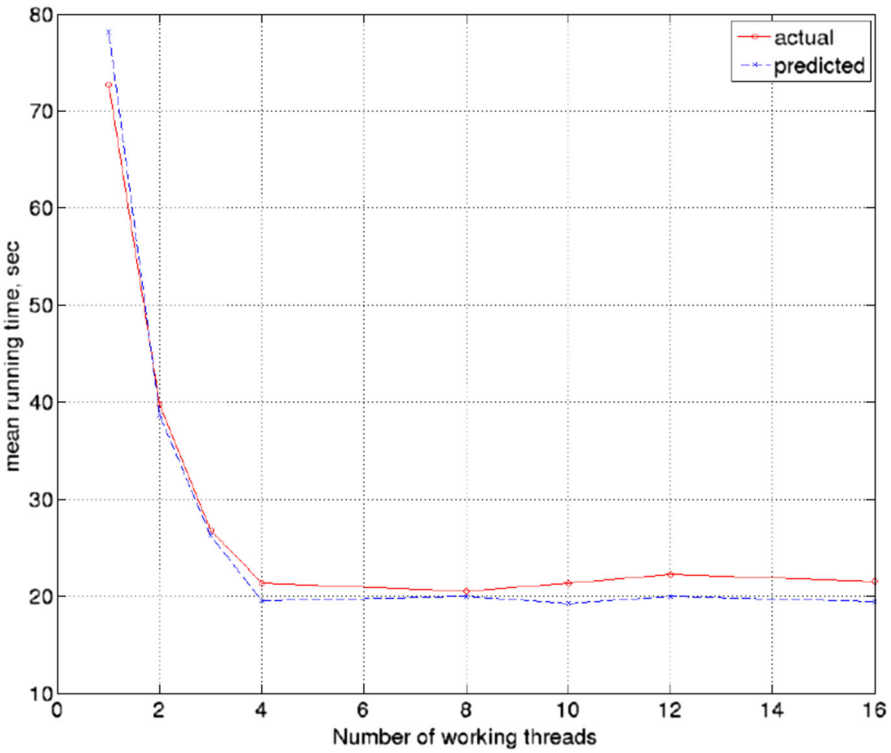

Fig. 7 Predicted and measured running time for Raytracer in hardware Config I

tracer. Assuming a constant size for the image, the number of working threads is a determining factor of the performance of the Raytracer.

We built the model of Raytracer using a configuration with 3 working threads in both Config I and II. Figures 7 and 8 compare the actual and predicted performance of Raytracer in Config I and II correspondingly. We ran Raytracer in the Config I with 1,2,3,4,8,10,12,16 working threads. The relative prediction error in the Config I varied in $\varepsilon \in (0.029, 0.156)$ with the average error measured across all the configurations $\overline{\varepsilon} = 0.117$ (see Fig. 7). Correspondingly, we ran the Raytracer in the Config II with 1,2,4,6,8,10,12,15 working threads. The relative error in the Config II varies in $\varepsilon \in (0.041, 0.173)$ with the average error $\overline{\varepsilon} = 0.086$ (see Fig. 8). These results demonstrate good prediction accuracy for both hardware configurations.

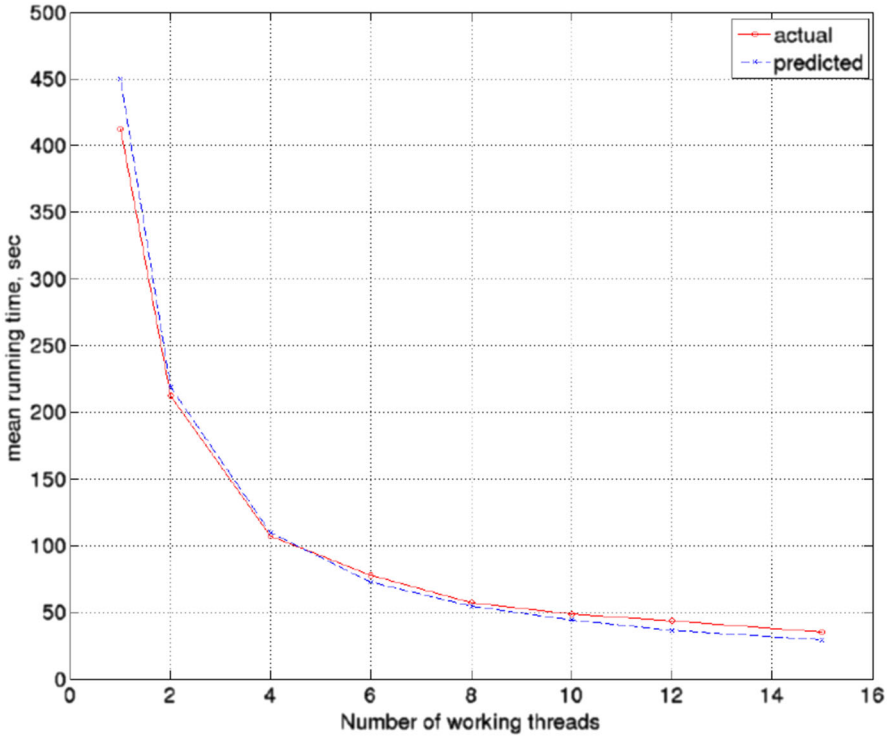

Fig. 8 Predicted and measured running time for Raytracer in hardware Config II

Montecarlo simulates prices of marked derivatives based on the prices of the underlying assets. Using historical data on asset prices, the program generates a number of time series using a Monte Carlo simulation. Each time series is considered as a "task"; time series are generated independently using a pool of working threads. Threads are synchronized using a barrier.

The number of threads is the main factor determining the performance of Montecarlo. The total time required to finish a simulation is the most important performance metric in this case.

In the Config I we built the model of Montecarlo using a configuration with 2 working threads and executed Montecarlo with 1,2,3,4,8,10,12,16 working threads. The relative error in this configuration varied in $\varepsilon \in (0.014, 0.105)$ with $\overline{\varepsilon} = 0.062$ (see Fig. 9). Correspondingly, in the Config II the model of Montecarlo was built using a configuration with 4 working threads. Montecarlo was executed with with 1,2,4,6,8,10,12,15 working threads; the error varied within $\varepsilon \in (0.029, 0.319)$ with $\overline{\varepsilon} = 0.184$ (see Fig. 10).

Although the prediction error remains within the acceptable limits in Config II, the performance of the Montecarlo becomes less linear in relation to the number of threads. To understand the cause of these errors we studied behavior of MonteCarlo using the Linux *perf* utility. It appeared that the Montecarlo performs a large number

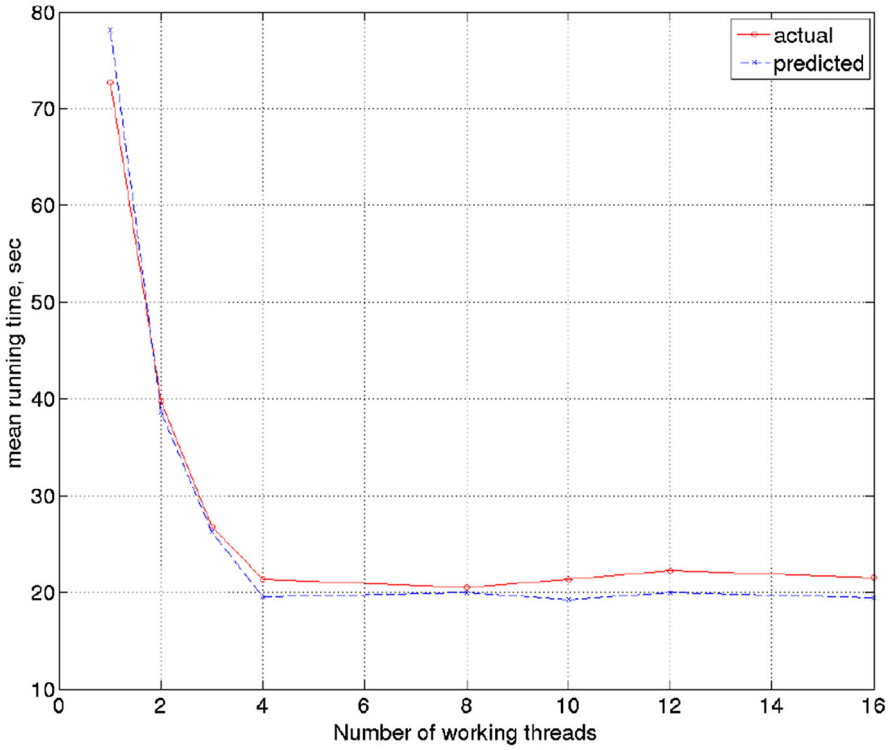

Fig. 9 Predicted and measured running time for Montecarlo in hardware Config I

of memory operations. When executed on a 16-core machine these operations saturate the memory bus, which leads to a performance degradation of the application. These errors could be addressed by a more detailed simulation of memory operations, which would involve collecting the information on memory accesses by the program and developing robust models of a memory subsystem.

Moldyn simulates motion of argon atoms in a cubic volume. Moldyn discretizes time into small steps (iterations). During each iteration Moldyn computes the force acting on every atom in the pairwise manner, and then updates the positions of the atoms.

Moldyn parallelizes computations across a pool of working threads. Objects that represent atoms are stored in the global synchronized queue. One of these threads (the main thread) coordinates actions of other threads using barriers. During each iteration working threads compute forces acting on atoms, and then the main thread merges forces computed by different threads and calculates updated positions of the atoms.

The length of the iteration is the most important performance metric of the Moldyn. Given the constant number of atoms, the size of the thread pool is the only parameter that determines performance of the Moldyn.

We built the model of Moldyn using a configuration with 2 working threads in both hardware Config I and II. Figures 11 and 12 depict prediction results in these config-

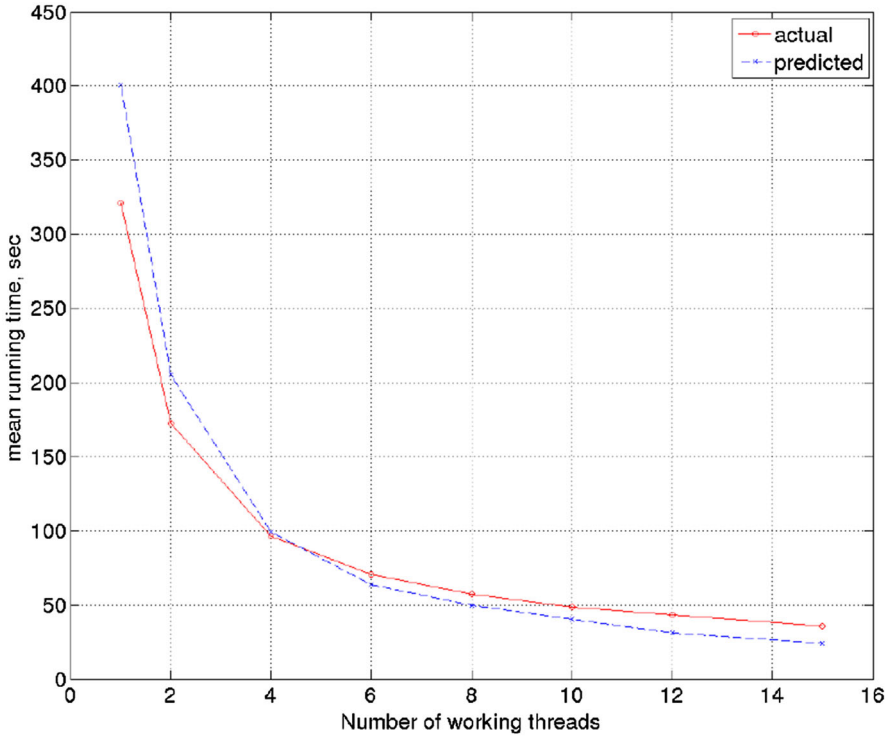

Fig. 10 Predicted and measured running time for Montecarlo in hardware Config II

urations. In Config I we executed Moldyn with 1,2,3,4,8,10,12,16 working threads; the relative varies in $\varepsilon \in (0.013, 0.155)$ with the average error measured across all the configurations $\overline{\varepsilon} = 0.083$ (see Fig. 11). In Config II we executed Moldyn with 1,2,4,6,8,10,12,15 working threads, and the relative error os $\varepsilon \in (0.006, 0.485)$ with the average error $\overline{\varepsilon} = 0.255$ (see Fig. 12).

The model predicts performance of Moldyn on a 16-core machine with lower accuracy than on a 4-core machine. Again, we used `perf` utility to understand the root cause of these errors. We discovered that specifics of data structure used by the Moldyn causes the cache miss rate to increase along with the number of threads. In particular, the miss rate for 1 thread is 0.0063%, while the miss rate for 15 threads is 0.0131% (5× increase). As a result, as the number of threads increases, the CPU time for the CFs increases as well, which leads to the reduction in the accuracy. An accurate model for CPU cache remains a subject of future work. Directions toward developing this model are outlined in the Sect. 6.3.

Galaxy simulates the gravitational interaction of celestial bodies using the Barnes and Hut (1986) algorithm, which relies on an octree data structure to speed up computations. During each iteration the main thread of the Galaxy rebuilds the octree, then the pool of "force threads" computes forces and updates positions of bodies, and, finally, the pool of "collision threads" detects body collisions. Pools communicate

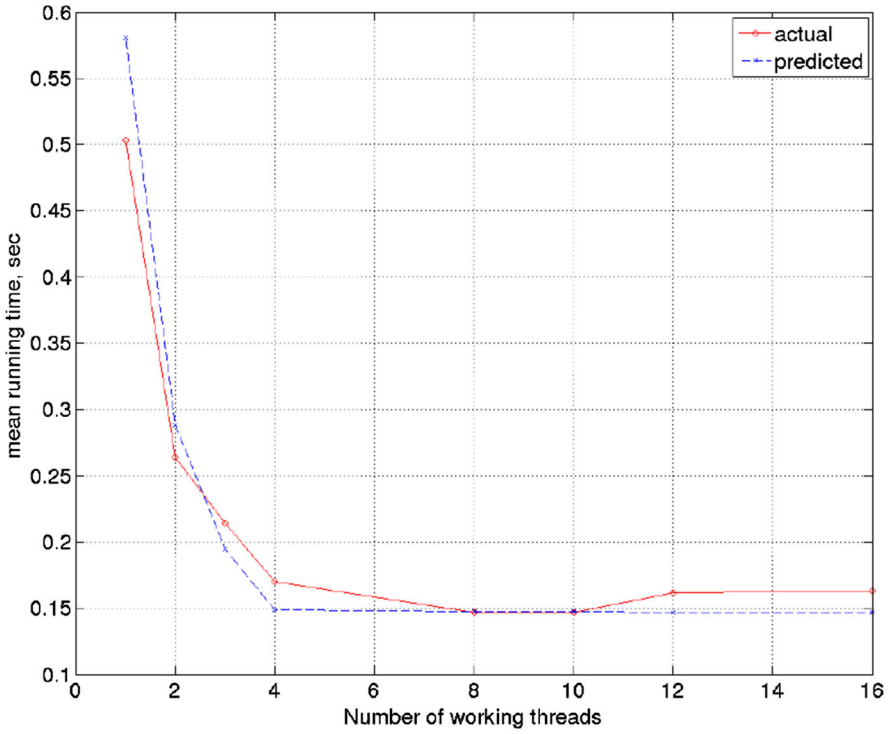

Fig. 11 Predicted and measured iteration length for Moldyn in hardware Config I

through the synchronized queues. The order of computations is enforced by the main thread. The number of force threads and the number of collision threads are the two parameters affecting the performance of the Galaxy. The time taken by an iteration is the most important performance metric of the Galaxy.

In both Config I and Config II we built the model of the Galaxy with 2 force and 2 collision threads. In the Config I we ran the Galaxy with 1,2,3,4,8,12 and 16 "force" and "collision" threads (total 49 combinations). The relative error for Galaxy in the Config I varies in $\varepsilon \in (0.002, 0.291)$ with average error $\overline{\varepsilon} = 0.075$ (see Fig. 13). In the Config II we ran the Galaxy with 1,2,4,6,8,10,12 and 15 "force" and "collision" threads. The relative error in the Config II varies in $\varepsilon \in (0.004, 0.358)$ with $\overline{\varepsilon} = 0.092$ (see Fig. 14), which is almost as accurate as the prediction for 4 CPU cores.

Our model correctly predicts some interesting aspects of the Galaxy performance. First, the model correctly points that the influence of the number of "collision threads" on performance is minimal, as these threads constitute a minor fraction of computations compared to the "force threads". Second, the model predicts the non-linear dependency between the number of "force threads" threads and performance of Galaxy. Increasing the number of "force threads" from 1 to 8 results in 5-fold improvement in performance, while increasing the number of these threads from 8 to 15 improves performance only by 35%. This phenomenon is explained by the Amdahl's law (Amdahl 1967). Namely,

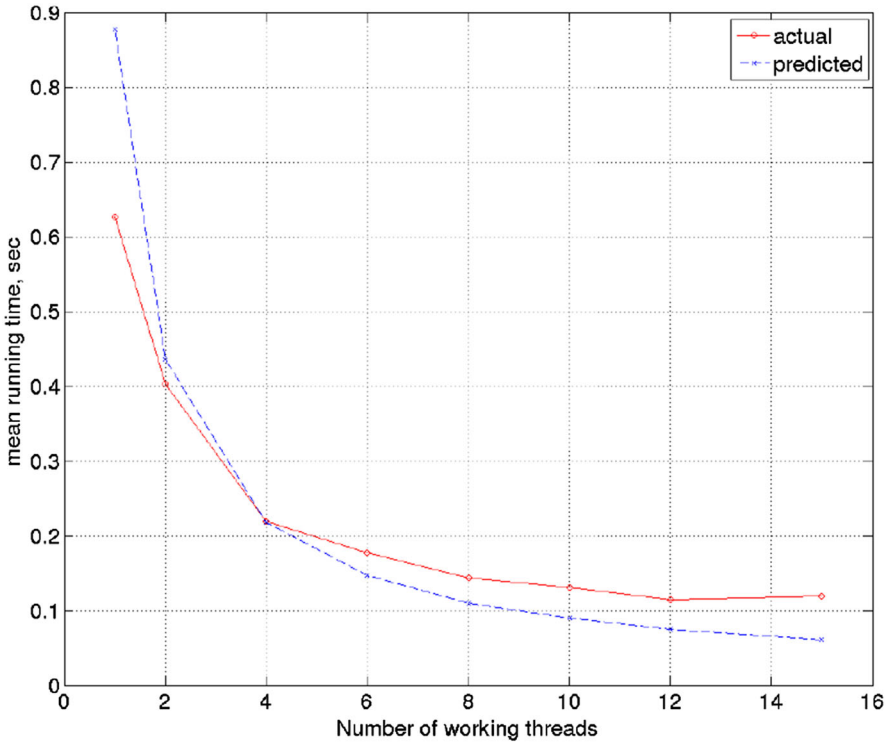

Fig. 12 Predicted and measured iteration length for Moldyn in hardware Config II

rebuilding the octree is not parallelized, and is performed by the main thread. As the number of working thread increases, the time for rebuilding an octree becomes a dominant factor in performance. Furthermore, accessing synchronized queues by the program's threads is also a sequential operation that involves locking, whose impact on performance becomes noticeable as the number of threads grow.

This analysis is reinforced with the prediction of CPU usage by the Galaxy in Config II (see Fig. 15). In particular, we correctly predict that all the CPU cores are never fully utilized. The relative prediction error for CPU utilization varies in $\varepsilon \in (0.004, 0.191)$ $\overline{\varepsilon} = 0.080$.

Tornado is a simple web server, whose structure and behavior are described as an example in the Sect. 3. Unlike Moldyn, Montecarlo, and Galaxy, which engage the CPU-intense computations, Tornado workload is dominated by disk I/O operations. The performance of the web server is influenced by two parameters: the incoming request rate (IRR), which represents the intensity of the workload, and the number of working threads. IRR is measured as the number of requests the web server receives in a time unit. The performance of the web server is characterized by two main metrics: its response time R and throughput T.

Predicting performance of the web server is a more complex problem because it requires simulating not only computations but also the disk and network I/O operations.

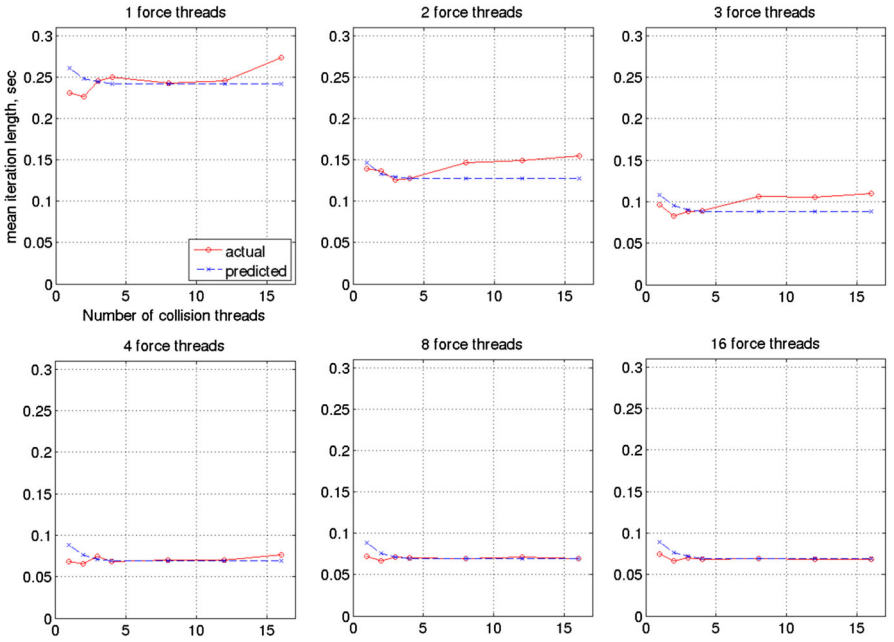

Fig. 13 Predicted and measured iteration length for Galaxy program in Config I

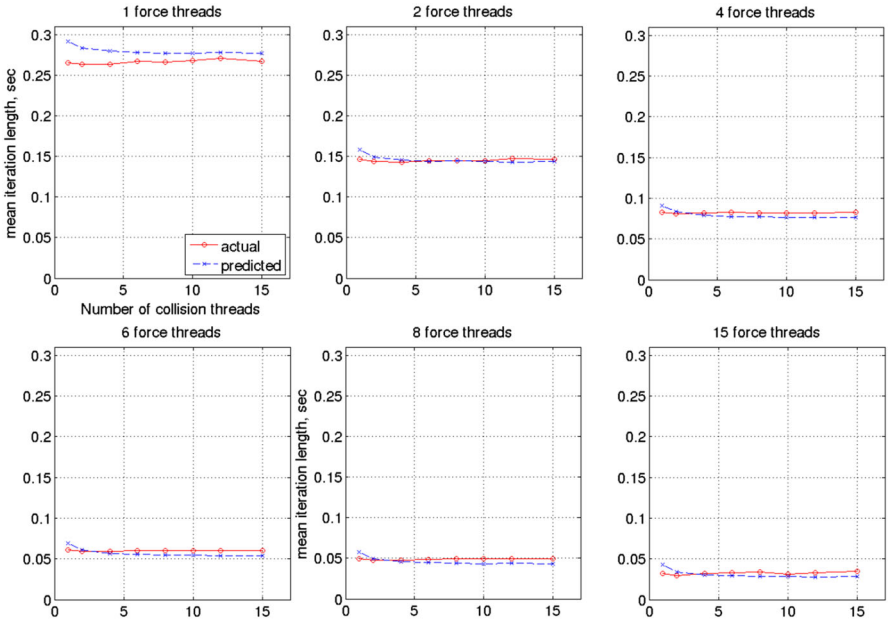

Fig. 14 Predicted and measured iteration length for Galaxy program in Config II

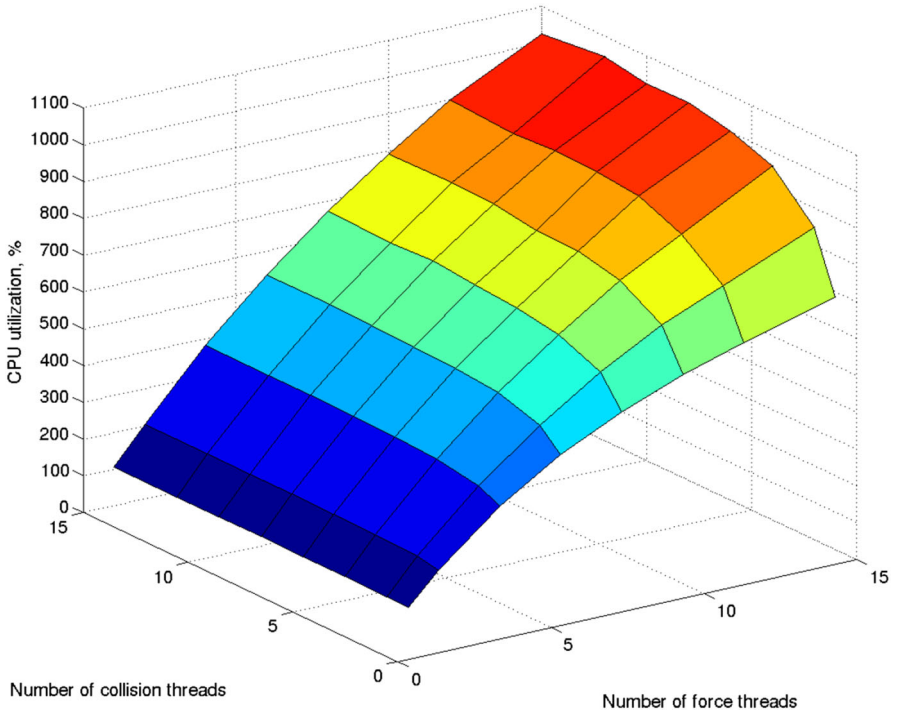

Fig. 15 Predicted CPU utilization for Galaxy in Config II

In our experiments Tornado was deployed in hardware Config I and used to host roughly 200,000 Wikipedia web pages. We used a separate client computer to simulate the incoming connections.

In our experiments we ran Tornado with with 1,2,4 and 8 working threads and IRR ranging from 19.75 to 97.2 requests per second (rps), measured at the server side. The model of the web server was built using a configuration with IRR=57.30 requests per second (RPS) and 1 working thread.

The prediction of the response time is shown at the Fig. 16. Predictions of the throughput are shown at the Fig. 17. The relative prediction error for response time R is in $\varepsilon(R) \in (0.017, 1.583)$ with $\overline{\varepsilon}(R) = 0.249$. Prediction for throughput T and hard drive utilization U_{disk} are considerably more accurate. The relative error for T is $\varepsilon(T) \in (0.000, 0.051)$ and $\overline{\varepsilon}(T) = 0.012$; the error for hard drive utilization $\varepsilon(U) \in (0.000, 0.077)$, while $\overline{\varepsilon}(U) = 0.025$.

One cause for the relatively high error terms for R is the variance in page cache hit rate; the next section of the paper describes these effects in more detail. Another cause is the simplistic model of networking operations, which are currently simulated as CPU computations.

The model correctly predicts that the number of working threads has a weak influence on the performance of Tornado. The single hard drive becomes a bottleneck, so any increase in the number of parallel I/O operations is negated by the proportional

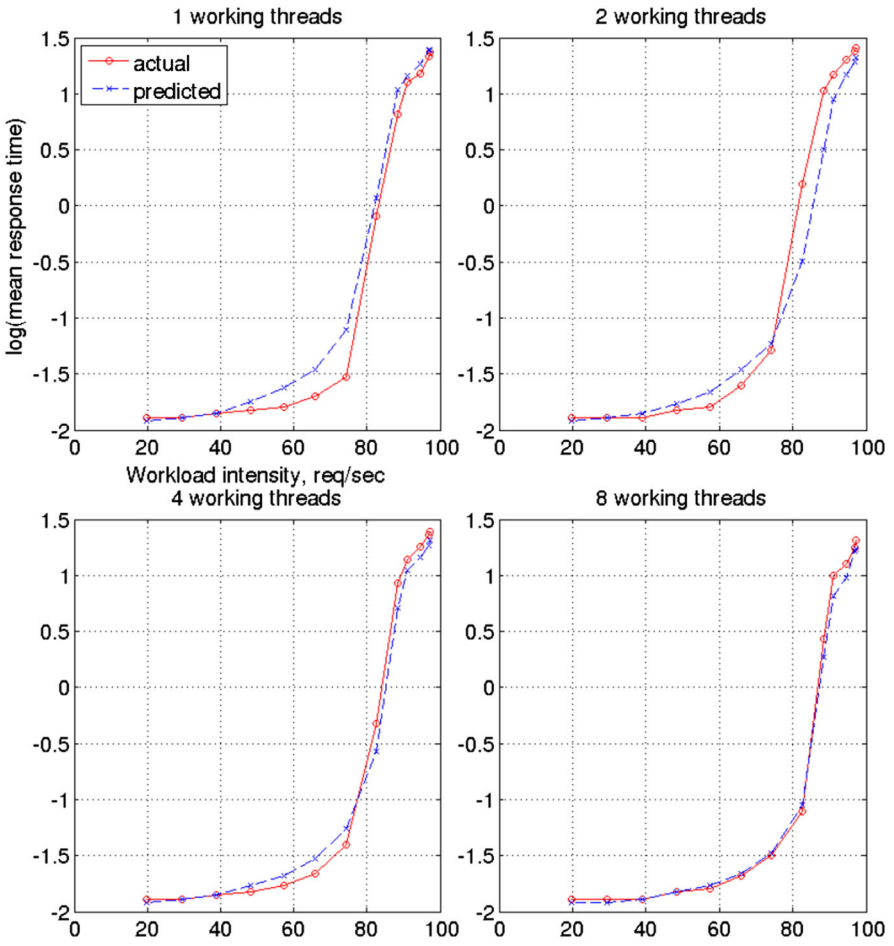

Fig. 16 Predicted and measured response time for Tornado

increase in the average execution time for each I/O request (see Fig. 18 for predicted hard drive utilization). We believe this example demonstrates the necessity of proper simulation of I/O operations in multithreaded programs because they often become a determining factor in the program's performance.

5.2 Modeling large industrial applications

Modern multithreaded applications are significantly larger and more complex than the programs studied in the previous section. To prove the practical value of our methodology we must demonstrate that our framework is capable of building accurate models of industrial-scale applications. We built models of the following large open-source programs: Sunflow 0.07 3D renderer and Apache Tomcat 7.0 web server.

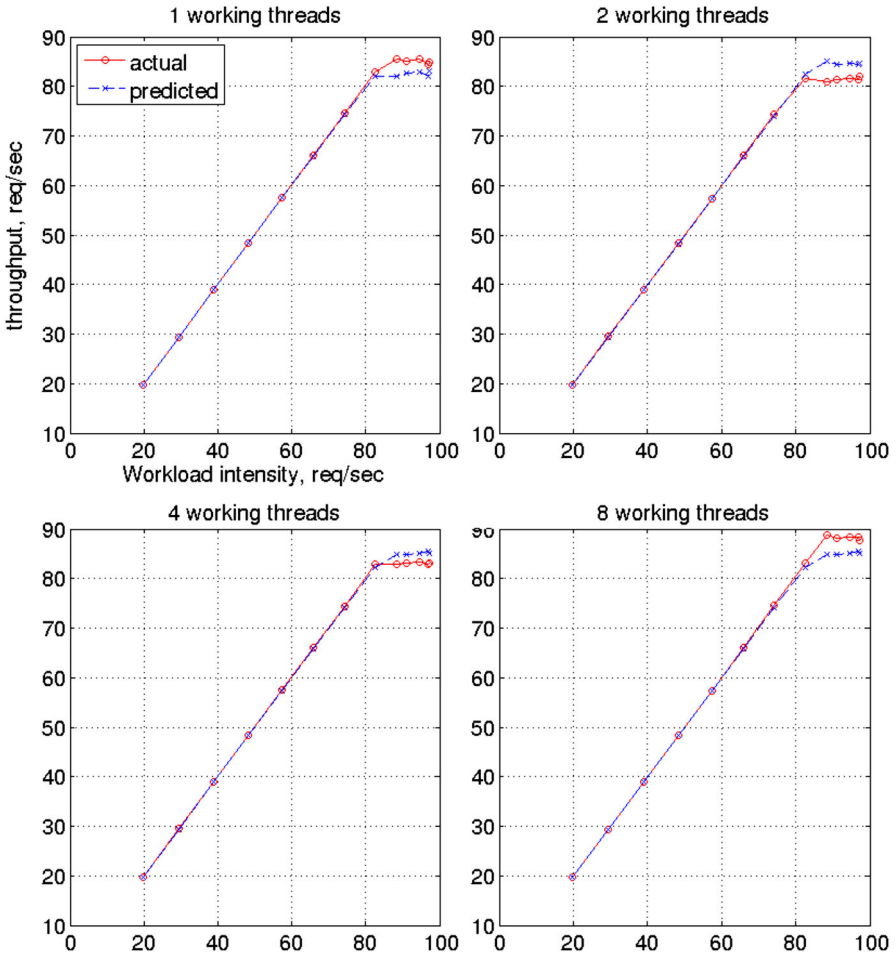

Fig. 17 Predicted and measured throughput for Tornado

We predicted the performance of Tomcat in two setups: as a standalone web server that hosts static web pages and as a servlet container that hosts an iText library for text conversion. Considering difference in Tomcat functionality over these setups, the corresponding models are significantly different. Table 4 provides information on large programs and their models.

Instrumentation did not alter semantics of these programs, but introduced some overhead. The overhead, measured as a relative increase in the task processing time by an instrumented program, ranged from 2.5 to 7.6%.

The complexity reduction algorithm eliminated 99–99.5% of all CFs as insignificant in the Tomcat and Tomcat+iText models correspondingly. Most of insignificant CFs were detected during the startup or shutdown phases. No startup or shutdown phases were detected in the Sunflow, and only 80% of its CFs were eliminated as insignificant.

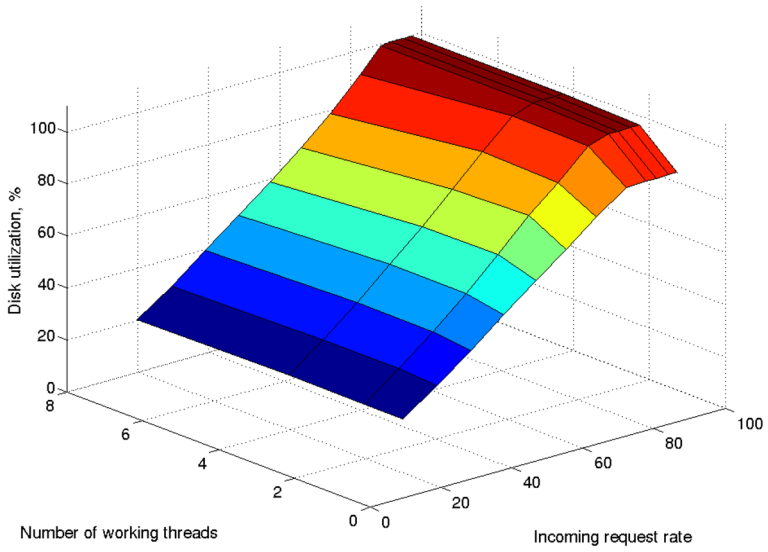

Fig. 18 Predicted utilization of the hard drive by Tornado

Table 4 Large programs and their models

	Tomcat (web server)	Tomcat+iText (servlet container)	Sunflow
Program size (LOC)	182,810	283,143	21,987
Number of probes	3178	3926	380
Mean instrumentation overhead	7.3%	2.4%	5.7%
Number of CFs	11,206	9993	209
Total number of nodes in the model	82	49	42
Simulation speedup	8–26	37–110	1050

Our models run 8-1000 times faster than the actual program (see Table 4). The actual speedup depends not on the size of a program, but on a ratio between the times required to simulate CFs by the model and times required to execute these CFs by the program. Simulating a CF requires a (roughly) constant amount of computations, regardless of its execution time. Thus models that invoke many CFs with short execution times or simulate intense locking operations tend to run slower than models that execute few long-running CFs. As a result, eliminating insignificant CFs is essential for achieving a high performance of the model.

Using performance models offers two additional sources of speedup over benchmarking. First, multiple instances of a model can run simultaneously on a multicore computer. Second, the model does not require a time-consuming process of setting up the live system for experimentation.

All the experiments with large applications were performed using configuration Config I, on a computer with a 2.4 GHz Intel Q6600 quad-core CPU and 4 GB RAM.

Below we briefly describe architecture of our large-size testing applications and discuss the result of our simulations.

Sunflow 3D renderer Sunflow is a 3D renderering program for photo-realistic image synthesis. The program features an extensible object-oriented design that allows for extending and customizing the ray tracing core (See Footnote 2). The Sunflow offers a wide range of features including various types of cameras, surface shaders and modifiers, light sources and image filters, and various file formats for importing and exporting data.

Upon the start of the Sunflow the main thread reads a scene specification from the disk, splits the frame into multiple tiles that correspond to "tasks" in our model, and stores tile coordinates in the queue. Then the main thread starts working threads. The pool of working threads reads tile coordinates from the queue, renders the image tiles, and synthesizes the resulting image.

Given the constant size of the image, the number of working threads and the number of CPU cores are two main factors that determine the performance of the Sunflow. The time required to render the image is the main performance metric.

We predicted Sunflow performance with 1,2,3,4,5,6,8,11,12 and 16 working threads and with 1,2,3 and 4 active CPU cores. Figure 19 compares predicted and measured rendering times in each of these configurations. The relative error varies in $\varepsilon \in (0.003, 0.097)$ with the average error across all the configurations $\overline{\varepsilon} = 0.032$.

Apache Tomcat as a web server Apache Tomcat is a web server and Java servlet container (see Footnote 1). Thanks to its reliability, flexibility, and high performance Tomcat is widely used in industry[8] However, these Tomcat features come at the cost of the high internal complexity. Tomcat consists of over 200,000 lines of Java code and hundreds of Java classes. Tomcat uses up to 10 different thread pools to start up and shut down the program, to accept incoming connections, to process timeouts, to serve incoming HTTP requests, and for other purposes. Web applications hosted by the Tomcat can perform synchronization and start additional threads, further increasing complexity of the system.

We used Tomcat to host over 600,000 Wikipedia web pages. In our experiments Tomcat relies on a single blocking queue to pass incoming HTTP requests to a fixed-size thread pool. The performance of the Tomcat was influenced by the size of the thread pool and by the the workload intensity (the number of requests the server receives in a second, req/s). The performance metrics are response time R and throughput T.

The model of the web server was built using a configuration with workload intensity 92 requests per second (req/s) and 1 working thread. We predicted performance of Tomcat with the number of working threads ranging from 1 to 10, and workload intensity ranging from 48.3 to 156.2 req/s. During each run 10,000 requests were issued.

The prediction results for R and T are depicted at the Figs. 20 and 21 respectively. The relative prediction error $\varepsilon(T) \in (0.001, 0.087)$ with average error $\overline{\varepsilon}(T) = 0.0121$. In non-saturated configurations throughput is roughly equal to the incoming request

[8] http://tomcat.apache.org/.

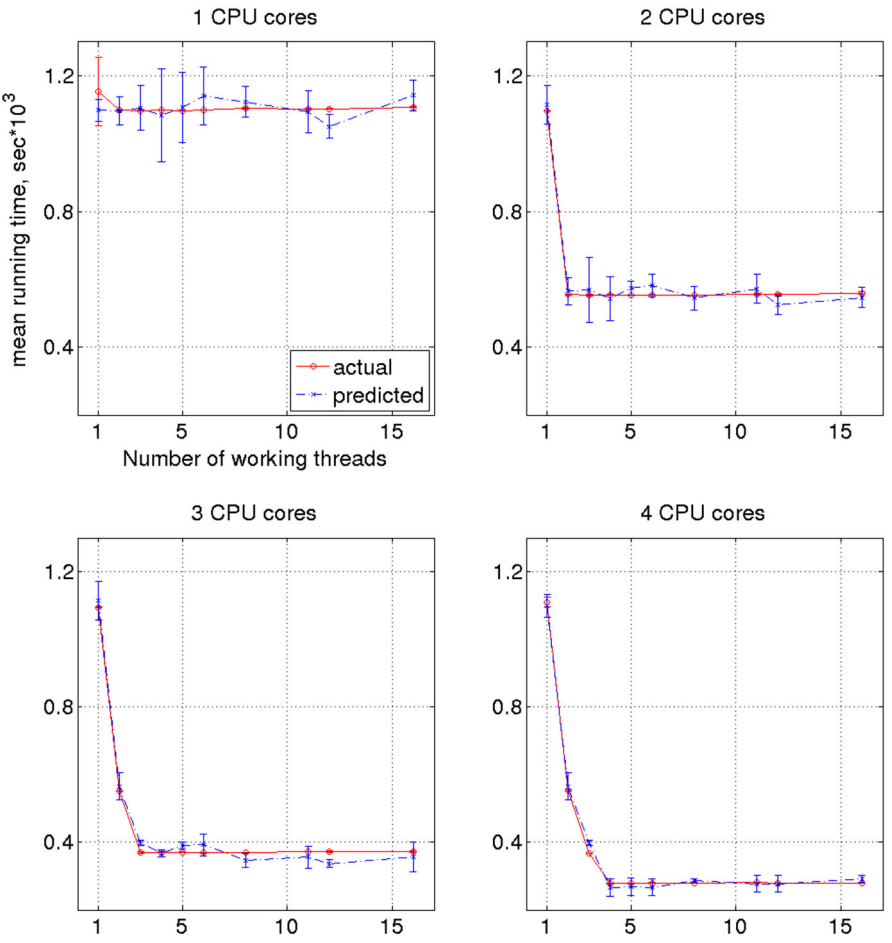

Fig. 19 Predicted and measured performance of Sunflow

rate, thus the relative error for saturated configurations is a more informative accuracy metric: $\overline{\varepsilon}(T_{sat}) = 0.027$.

The error for R is $\varepsilon(R) \in (0.003, 2.452)$ and $\overline{\varepsilon}(R) = 0.269$. Similar to results for the Tornado web server, the prediction error for the response time was relatively high. We investigated this phenomena and concluded that increase in error terms is attributed to fluctuations of the page cache hit rate k across the configuration space used for Tomcat, which, according to our measurements, varied with mean $\overline{k} = 0.755$ and standard deviation $\sigma(k) = 0.046$. In statistical terms this means that in 95% of cases the true value of k will vary between $(0.663, 0.847)$ across different configurations. These variations in the page cache hit rate cause proportional variations in the request processing time by the working threads. However, in saturated configurations, when the HTTP requests start to accumulate in the queue, even small variations in the request processing time result in large variations in the response time R.

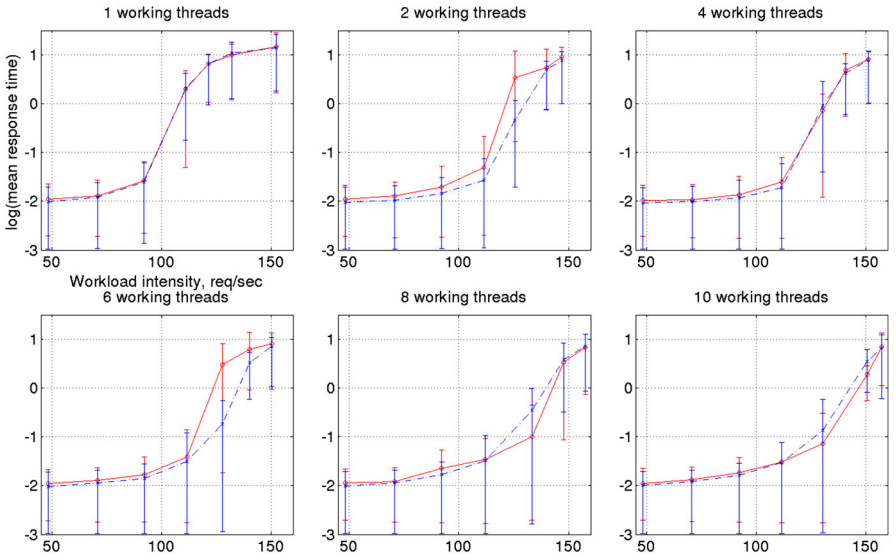

Fig. 20 Response time of Tomcat in a web server setup

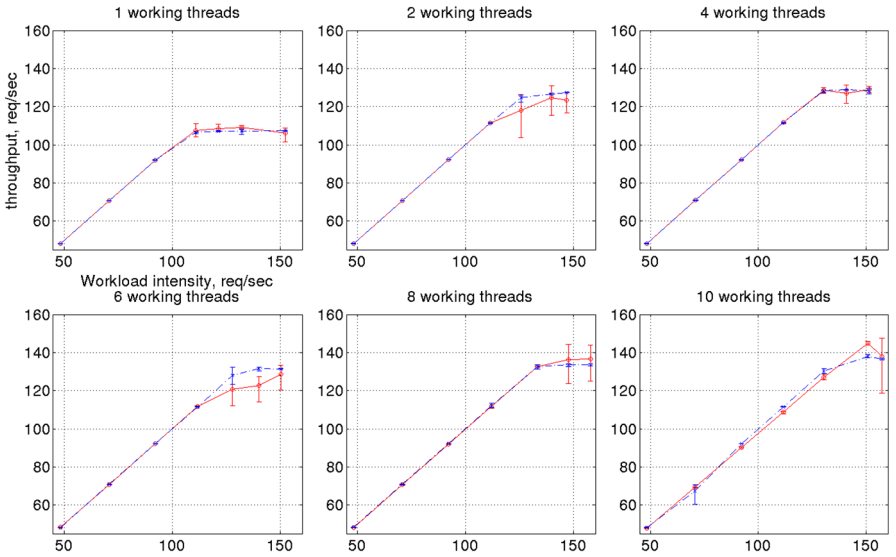

Fig. 21 Throughput of Tomcat in a web server setup

To verify our assumption about the cause of inaccuracies we introduced a 15% artificial bias in k. This resulted in increasing the relative error to $\varepsilon(R) \in (0.015, 3.109)$ with $\overline{\varepsilon}(R) = 0.882$. We believe this experiment demonstrates the difficulties in predicting the inherently variable disk I/O operations. Moreover, it emphasizes the importance of precise data collection for accurate performance prediction because even a small bias in data collection results in a large prediction error.

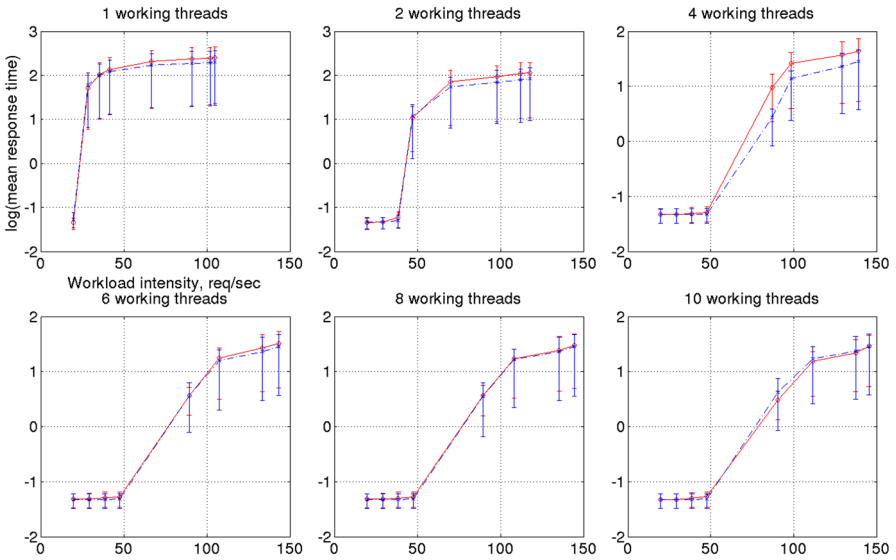

Fig. 22 Response time of Tomcat in a servlet container setup

Our model correctly predicts that the number of working threads has a minor impact on performance of Tomcat in this setup. This can be attributed to a mixed behavior of Tomcat in a web server setup. 81% of computational resources consumed during processing the HTTP request is the I/O bandwidth, and 19% is CPU time. As a result, the single hard drive becomes the bottleneck that prevents performance from growing significantly as the number of working thread increases. At the same time, remaining CPU computations are parallelized across four CPU cores, resulting in small but noticeable performance improvement.

Apache Tomcat as a servlet container Tomcat is more frequently used as a servlet container. We used Tomcat to host a web application that reads a random passage from the King James bible, formats it, and converts into the PDF document using the iText.[9] library.

The model of the web server was built using a configuration with workload intensity 57.30 requests per second (req/s) and 1 working thread. We predicted performance of Tomcat with the number of working threads ranging from 1 to 10 and workload intensity ranging from 19.67 to 132.68 requests per second. During each run 10000 requests were issued. The prediction results for R is depicted at the Fig. 22, and results for T are depicted at the Fig. 23.

The relative prediction error for response time across all the configurations $\varepsilon(R) \in (0.000, 0.716)$ with the average error $\overline{\varepsilon}(R) = 0.134$. The CPU time $CPUtime$ fluctuates less than the demand for I/O bandwidth, which leads to the lower prediction error in a servlet container setup.

[9] http://itextpdf.com/.

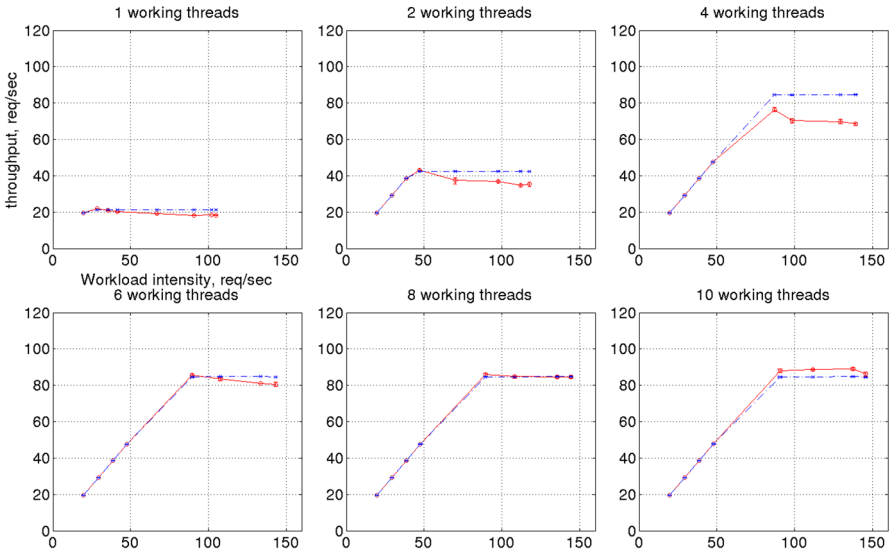

Fig. 23 Throughput of Tomcat in a servlet container setup

The prediction error for throughput across all configurations $\varepsilon(T) \in (0.000, 0.236)$, while the mean error $\overline{\varepsilon}(T) = 0.053$. For saturated configurations, $\varepsilon(T) \in (0.000, 0.356)$ and $\overline{\varepsilon}(T_{sat}) = 0.099$.

The model correctly predicts the workload intensity at which the server saturates. PDF conversion is a CPU-heavy task, thus performance of the server is bounded by the number and performance of CPU cores. Since there are four CPU cores available, the actual saturation point depends on the number of threads. It ranges from 21.4 req/sec for a configuration with 1 thread to 85.5 req/sec for 8 threads.

6 Discussion and future work

As we experimented with our models, we made some interesting findings about our approach, discovered its limitations, and laid ground for the future work. Below we discuss these aspects of our work in detail.

6.1 Findings

We found that modeling locks and synchronization operations is essential for an accurate and semantically correct model of the multithreaded system. Locks not just influence performance of the system. They often form a "skeleton" of the program, which coordinates work of all the program's threads. Failure to simulate these locks results in a non-functional model of the program.

We learned that building simulation models that can handle a broad range of multithreaded programs is difficult. In particular, different programs use various approaches to implement threading, so discovering the semantics of thread interaction can be dif-

ficult in the general case. However, the analysis of the program is greatly simplified if that program uses a specific implementation of high-level locks and queues. Models of such programs can be built automatically.

Proper modeling of hardware is essential for an accurate simulation. This includes modeling of CPU computations, disk I/O, CPU cache, memory and network. However, it is challenging to construct models of hardware that are both accurate and fast.

Accurate performance prediction requires precise measures of resource demand for the elements of the program. Small errors in measuring resource demand may lead to large prediction errors. However, obtaining precise measurements of resource demand without introducing a significant overhead is difficult. Moreover, resource demand can vary in time, leading to a decrease in prediction accuracy.

We found that in order to be fast and easy to understand the resulting model must be simple and compact. A large and complex model can run slower than the actual program, which defeats the whole purpose of performance prediction. Building compact models requires identifying program constructs that do not have significant impact on performance, and excluding these constructs from the model.

Finally, debugging performance models is difficult. Often the only manifestation of the bug is the deviation between the predicted and the actual performance. Although we use a simple step-by-step procedure for locating bugs in models, developing tools and methods for debugging performance models may be a prerequisite for their practical use.

6.2 Limitations

Although our framework is capable of building performance models automatically, it imposes certain limitations on the programs we can model. Below we discuss limitations in our approach. The next section outlines our plans to address those limitations.

Our high-level models represent computations as task processing. Although this approach does not cover all possible programs, it allows simulating most programs of interest for performance purposes. Moreover, our models do not simulate performance characteristics of individual requests but rather predict average performance of the system for a given workload.

Currently PERSIK can not model distributed systems. In particular, PERSIK does not explicitly simulate calls made by the program to other systems, such as Web services or SQL databases. Representing these calls as computations or I/O operations would result in an inaccurate prediction.

The PERSIK model may become inaccurate if usage patterns for the program change significantly. Examples of such patterns are the image resolution in Sunflow or probabilities of accessing individual web pages in Tomcat. Changes in these patterns may result in changes in behavior and resource usage of the program, which will lead to changes in performance. In terms of PERSIK, changes in usage patterns lead to changes in transition probabilities δ and CF parameters Π of the model. Updating δ and Π may require reconstructing the model.

Currently PERSIK cannot model programs that use low-level constructs, such as Java monitors, to implement high-level locks like barriers or blocking queues. Our

framework can automatically build models of only those programs that implement multithreading using the well-defined synchronization operations. We do not see it as a major limitation as modern programming frameworks offer rich libraries of locks, and programmers are encouraged to use these instead of developing their own implementations of locks (See Footnote 3). Moreover, semantics of locks implemented using low-level constructs can be discovered using analysis described in Reiss and Tarvo (2013). However, programs that implement "custom" locks that cannot be assigned to one of existing lock types (semaphore, barrier, mutex etc), cannot be modeled at this moment.

Our framework can handle some changes in hardware, such as the different number of CPU cores. However, this does not yet translate into an accurate prediction of the program running on a totally different hardware. Differences in characteristics of CPU, memory, and cache will result in different execution times for individual CFs.

PERSIK simulation framework does not include models of network, RAM, and CPU cache. This prevents our framework from accurately modeling some aspects of the system's performance, such as of memory bus contention, network contention, and cache coherence. As a result, the modeling accuracy can decrease for certain workloads and hardware platforms.

6.3 Future work

We plan to address limitations outlined above and to extend the scope of our approach, so it could predict performance for a wider range of programs and workloads. A special attention should be given to predicting performance of programs running on different hardware and having a wide variety of usage patterns. Examples are predicting performance of Sunflow image renderer with different image sizes, or predicting performance of Tomcat on different hardware. These predictions may require developing new modeling architectures and new approaches towards automatic building of these models.

One approach that would allow modeling changes in both usage patterns and hardware is a hybrid of statistical and simulation models. In a hybrid model the usage patterns such as the image size in a 3D renderer or the number of particles in molecular simulator are defined using metrics X_{pat}. The dependency $(\delta, \Pi) = f(X_{pat})$ between the structure of the execution graph δ and resource demands Π on the one side and the usage patterns X_{pat} on the another side would be approximated statistically. The resource demands Π_{CPU} and Π_{disk} for CFs running on different hardware would be modeled in a similar manner. For example, the amount of CPU time $\Pi_{CPU} = CPUtime$ for a computational CF could be defined as $CPUtime = f(X_{CPU}, X_{cf})$, where X_{CPU} are metrics that describe a CPU (e.g. microarchitecture and clock rate), and X_{cf} are metrics that describe the mix of CPU instructions executed by that CF.

Data required to approximate $(\delta, \Pi) = f(X_{pat})$ will be collected by running the program with different usage patterns. Similarly, the dependency $CPUtime = f(X_{CPU}, X_{cf})$ can be approximated using a library of microbenchmarks. Microbenchmarks that are representative over a variety of CF types will be executed on different hardware platforms. Performance of those microbenchmarks will be measured, providing information for building a variety of models.

Although building the hybrid model would require multiple runs of the program, we expect the number of these runs to be significantly smaller than if the pure statistical model was used (Chun et al, 2010). One reason for that is that parameters X_{CPU}, X_{cf} and X_{pat} are likely to be conditionally independent given δ, Π.

We expect that hybrid models would be particularly useful in a cloud setting. Cloud-based programs are executed in a variety of configurations, and data on these executions can be collected in a centralized manner. This data can be used to approximate dependencies such as like $(\delta, \Pi) = f(X_{pat})$. Furthermore, cloud providers usually offer a limited variety of hardware, which simplifies modeling of different hardware configurations.

In a scenario where multiple runs are undesirable, changes in usage patterns can be tackled by recollecting δ and Π directly from the running program and updating the model on-line. This approach can account for usage patterns that were previously unseen. However, it would require developing techniques for low-overhead program analysis that can be enabled and disabled dynamically during the program's execution.

If measuring hardware performance through microbenchmarks is not possible, then network, memory, and cache operations should be modeled explicitly. Although models for predicting memory and cache performance are known (Nethercote et al. 2006; Gulur et al. 2014), these models either require data specific to a particular execution of the program or work significantly slower than the program itself. Developing accurate and robust models for predicting performance of memory and cache is a challenging area.

Another direction for the future work is adopting PERSIK for modeling distributed systems. Modern server-side applications are usually distributed. These programs issue calls to remote applications running on different machines, such as databases or cache services. As a result, the performance of such program is often determined by the timing of these calls.

PERSIK models in their current form cannot simulate such distributed systems. However, PERSIK can be extended by introducing another layer in the hierarchy of models. This layer will represent the topology of the distributed system, where nodes represents individual hosts and links between these nodes are the network connections. The topological layer of the model can be built using INET[10] or NS3[11] simulators. The topological layer will predict the performance of the distributed system at the global scale by modeling delays caused by network communication between its individual hosts. Subsequently, performance of each individual host will be simulated using a corresponding PERSIK model.

6.4 Threats to validity

In our experiments we often relied on artificial benchmarks. We believe this is not a major concern, as difference in workload would simply manifest as differences

[10] http://www.inet.omnetpp.org/.

[11] http://www.nsnam.org/.

in structure δ and parameters Π of the model. A more elaborate validation would require collecting actual workload of the production system, and replaying it in our experimental setup.

We parameterized our model with only three variables: the number of working threads, the number of CPU cores, and the workload intensity. Adding more parameters to the model, such as characteristics of the workload or hardware, would require extending PERSIK framework as discussed in Sect. 6.3.

The precision of our experiments was limited by the precision of tools to measure performance of the system. We measured wall-clock time using `System.nanoTime` Java call, which has precision of 1 ns. Measurements of CPU time, used to compute CPU utilization, had lower precision of 10 msec. To ensure sufficient accuracy we ran each experiment for hundreds of seconds, so effect of measurement inaccuracies become negligible.

To evaluate accuracy of our models we repeated our experiments three times in each configuration and compared mean values of performance metrics. Increasing the number of executions to ten in few selected configurations didn't alter our results significantly. Thus we do not see the low number of repetitions as a potential threat to the statistical validity of our results.

We evaluated PERSIK by building models of 8 different programs. Although our set of programs was diverse, it may not be representative over all the existing multithreaded programs, their workloads, hardware platforms and programming languages. This threatens the external validity of our work and calls for a more thorough evaluation of PERSIK with a wider range of programs and workloads.

We built models of only programs written using Java programming language. PERSIK can successfully model C/C++ programs (Tarvo and Reiss 2012), but automatic building of models written in languages other than Java is not yet implemented. Similarly, we used only two hardware setups in our experiments. Thus it is important to validate PERSIK on a wider range of hardware platforms, including different CPUs or different storage system architectures.

7 Related work

We divide the related work into two categories: performance modeling, and automated program analysis and model construction.

7.1 Performance modeling of computer programs

At the high level the performance of the system can be represented as a function $y = f(\mathbf{x})$, where $\mathbf{x}$ are metrics describing the configuration and workload of the system, and y is a measure of the system's performance. Existing approaches to performance modeling can be divided into three classes based on their representation of the dependency $y = f(\mathbf{x})$: analytic models, statistical models, and simulation. Below we review these model classes and compare them to PERSIK, which is a simulation model.

Analytic models explicitly represent the dependency $y = f(\mathbf{x})$ using a set of equations. They were used for a variety of tasks.

An analytic model was used to predict the dependency between the size of the DBMS cache and its response time R and throughput T (Narayanan et al. 2005); the relative errors are $\varepsilon(T) \leq 0.1$ and $\varepsilon(R) \in (0.33 \ldots 0.68)$. Analytic models were employed to predict the running time of MapReduce job (Herodotou and Babu 2011); the profile of a job was retrieved using dynamic analysis. An analytic model was used to predict the utilization of the L2 cache and memory bandwidth for a given program on a multiprocessor system with the average error in (0.09–0.13) (Chen et al. 2011). Analytic models were also employed to study performance of certain multithreaded design patterns (Strebelow et al. 2012).

In Chen et al. (2005) an analytic model was used to predict the optimum size of the thread pool for a component-based server-side application. Similarly to PERSIK, parameters of the model are estimated by benchmarking the program. The accuracy of the model was $\varepsilon(R) \in (0.001 \ldots 0.27)$; the errors were highest in for configurations where the size of the thread pool exceeded the number of simultaneous client connections. Furthermore, it is not clear if the program under the study performed any synchronization operations and what were the characteristics on the workload.

Analytic models are compact and expressive. They usually work faster than simulation models, such as PERSIK. However building analytic models require knowledge of the system's functionality and a substantial mathematical skill to formalize this functionality using a set of equations. Unlike PERSIK, analytic models have difficulties expressing a complex behavior of a multithreaded application. In particular, analytically predicting performance of a single thread pool requires development of a complex mathematical model that must be solved iteratively (Menasce and Bennani 2006).

Nevertheless, analytic models can be used as a part of the larger model to predict performance for some of the system's components. For example, analytic models were used as a part of the larger model to simulate individual components of the distributed system, such as network and disk (Thereska and Ganger 2008).

Statistical models tend to overcome some drawbacks of analytic models. They do not explicitly formulate the function $y = f(\mathbf{x})$. Instead, the system is executed in a number of configurations $\mathbf{x}^1, \ldots, \mathbf{x}^n \in X$, where performance measurements $y^1, \ldots, y^n \in Y$ are collected. Then a statistical method is used to approximate the dependency $Y = f(X)$.

Statistical models can predict performance of a wide range of systems. A k-NN technique was used to predict the running time of SQL queries based on the features of the DBMS query plan (Ganapathi et al. 2009). The correlation between the actual and predicted execution times $R^2 \in (0.55 \ldots 0.95)$. This technique was further extended to predict the running time of Hadoop tasks (Ganapathi et al. 2010) with $R^2 \in (0.87 \ldots 0.93)$ A similar approach was used to predict performance of the SQL queries running in isolation with $\varepsilon \in (0.05 \ldots 0.1)$ (Akdere et al. 2012). Here the x vector can be built from individual operators of SQL query, which allows to train models on-line. This approach was further developed to predict the running time of a mix of concurrently running queries with $\varepsilon \in (0.14 \ldots 0.27)$ (Duggan et al. 2011).

Similarly, a non-linear regression was used to predict the response time in the message-passing middleware software (Happe et al. 2010). A combination of a linear

regression and neural networks predicted the running time of scientific computing applications on a grid with $\varepsilon \in (0.01, \ldots, 0.25)$ (Lee et al. 2007).

Statistical models do not require knowledge of system's internals, which is a major advantage compared to PERSIK and simulation models in general. But unlike PERSIK, building statistical models require running the system in many configurations, which is time-consuming and costly. Furthermore, any change to the system requires re-training the whole model (Thakkar et al. 2008). Finally, the accuracy of a statistical model strongly depends on the representativeness of the training dataset (Cheung et al. 2011).

There are attempts to alleviate these shortcomings. Statistical models can be built when large amounts of data are already available, e.g. from the from a large user base (Thereska et al. 2010) or from prior runs of the system in the cloud environment. In particular, the work Chen et al. (2014) presents the StressCloud framework to automatically generate workloads for cloud-based programs. StressCloud is used to analyze system's performance and energy consumption; however, same approach can be employed to speed up constructing of performance models.

The size of the training set can be further reduced by the sophisticated program analysis (Chun et al, 2010) or by special algorithms for composing a training set (Westermann et al. 2012). Finally, statistical models can be used successfully when the training dataset can be collected relatively quickly, e.g. by benchmarking. This allows modeling individual components of a large system, such as the disk I/O subsystem. CART trees were used to predict performance of the SSD disk with $\varepsilon \in (0.17 \ldots 0.25)$ (Huang et al. 2011), a regression tree predicted performance of the traditional hard drive with $\varepsilon \in (0.17 \ldots 0.38)$ (Wang et al. 2004), and the k-NN algorithm predicted the running time of disk I/O operations with $\varepsilon \in (0.02 \ldots 0.2)$ (Anderson 2001).

Simulation models, such as queuing networks, Petri nets, and their extensions mimic the behavior and/or structure of the system. These models are very flexible and capable of modeling complex systems.

Some simulation models can be solved analytically, which speeds up prediction greatly. However, simulation remains the main tool for predicting performance using these models because simulation can represent complex behavior of the system. Building a simulation model does not require running the system in many configurations but requires knowledge of the components of the system, their properties and interactions.

A variety of formal methods for building simulations have been developed. The first such methodology was queuing networks (Lazowska et al. 1984). In particular, queuing networks were used to model impact of networking parameters at the performance of the web server (Mei et al. 2001). However, queuing networks in their classical form can be too restrictive for simulating complex systems. As a result, a number of extensions have been developed.

Layered queuing networks (LQN) extend traditional queuing networks by adding the hierarchy of model components (Woodside et al. 1995). LQNs can be solved analytically and are particularly useful for simulation of distributed systems. LQNs were used to models performance of simple CORBA applications and web services with $\varepsilon \in (0.02 \ldots 0.05)$ (Hoecke et al. 2005) and to predict performance of the CPU-bound ERP application with $\varepsilon = 0.15$, although the application did not carry out any I/O or synchronization activities (Rolia et al. 2009). However, analytic solution

of LQN models that simulate complex threading behavior is challenging (Franks and Woodside 1998).

Another simulation methodology is Petri nets and their extensions, such as colored Petri nets (CPN) (Kristensen et al. 1998). CPN allow assigning values (denoted as colors) to the tokens. CPN were capable of simulating the complex locking constructs in a program (Roy et al. 2008). CPN predicted performance of a parallel file system with $\varepsilon \in (0.2 \ldots 0.4)$ (Nguyen and Apon 2012). CPN was used to build a model of the Linux Ext3 file system capable of simulating read and write operations, system's page cache, and the filesystem journal. The model predicted the average throughput of the filesystem with $\varepsilon \in (0.12 \ldots 0.34)$ (Nguyen and Apon 2011). This study was extended to allow simulation of the parallel file system with $\varepsilon \in (0.2 \ldots 0.4)$ (Nguyen and Apon 2012). Queuing Petri Nets (QPN) extend the Colored Petri nets by adding queuing and timing aspects into the model (Bause 1993). QPN was used to simulate distributed component-based and event-based systems (Kounev et al. 2012).

Simulation models can be used in combination with other model types. The work Liu et al. (2005) describes a combination of a queuing and an analytic models specifically designed to predict performance of EJB system. Similarly to PERSIK, the model is hierarchical. The queuing model simulates the flow of a task, and every service node contains an analytic model that estimates a resource demand for that node. The model predicts performance of EJB applications with $\varepsilon \in (0.05-0.14)$. Like PERSIK, the model Liu et al. (2005) relies on a single representative run of the system to obtain values of some parameters. But unlike PERSIK, the model is constructed manually. It does not simulate contention of hardware and locks; it is unclear if the model can handle programs other than EJB applications.

IRONModel also uses a combination of a queuing network and analytic model to simulate a distributed system (Thereska and Ganger 2008). Finally, PACE framework employs hierarchical approach to model MPI applications (Jarvis et al. 2008). PACE predicted the execution time of the `nreg` image processing application with $\varepsilon <= 0.1$ (Jarvis et al. 2008).

Simulation models are more flexible than analytic or statistical models. As a result, CPN and other methodologies were successful in simulating some aspects of multithreaded applications. However, we are not aware of any framework capable of simulating both locks and simultaneous hardware usage. We address this by developing performance models that can simulate both complex synchronization operations and simultaneous usage of hardware. This allows PERSIK to handle a larger variety of multithreaded programs.

7.2 Automatic analysis and performance modeling of computer programs

Although simulation models are more flexible, their construction is significantly more difficult because these models require extensive information on the system's internals and functionality. This information can be retrieved manually, as in Woodside et al. (1995), Mei et al. (2001), Hoecke et al. (2005), and Xu et al. (2005). However, the manual analysis of the software system is time-consuming and error-prone. Thus the problem of automatically analyzing computer programs and building their per-

formance models has gained significant attention. Techniques used for constructing performance prediction models are often used for other performance tools such as intelligent profilers and descriptive models. As a result, the boundary between program analysis techniques used to build predictive models and other performance tools is often vague.

Program analysis have been extensively used to understand the structure of multithreaded programs. A CHET tool extracts specifications from the running parallel programs and represents them as automata (Reiss 2004). This work was further extended to identify synchronization mechanisms in a multithreaded program and to determine their types through a dynamic analysis (Reiss and Tarvo 2013).

Similarly, the Magpie tool facilitates understanding the characteristics of the system's workload by inferring the flow of request from the sequence of API calls (Barham et al. 2004). The THOR tool relies on a sophisticated combination of kernel and user-mode instrumentations in order to understand and visualize relations between the Java threads and locks (Teng et al. 2010).

Perfume tool (Ohmann et al. 2014) represents the behavior of the system as a finite state machine (FSM). FSM actions correspond to the program's states, and transitions are conditioned on the resource demands of a state, such as memory or a CPU time. Perfume models are built from existing logs, thus FSM states in Perfume correspond to the system-specific concepts, e.g. TCP connection states or HTML page accesses. As a result, the main application of Perfume is not to predict, but to describe behavior of the system.

We believe Perfume is readily capable of building FSMs of multiple threads. However, representing semantic interactions between threads requires additional logic, which in PERSIK is taken by the high-level model. PERSIK also has a very limited ability to condition transitions in the probabilistic execution graph (PEG) on external factors, such as presence of a request in the queue. Implementing a full Perfume-like support for external conditions in PERSIK would enable simulating complex behaviors, such as modeling determinism in a program.

Program analysis has been also used to understand performance of the program. Input-sensitive profiling automatically measures how the input size of the program's functions affects the running time of these functions (Coppa et al. 2012). Similarly, in Zaparanuks and Hauswirth (2012) the tool automatically deduces the runtime cost of the algorithm based on the size of the data structures. A similar approach was used to measure the computational complexity of the application (Goldsmith et al. 2007). A stack sampling technique was used to identify parallel idleness and parallel overhead in the multithreaded program (Tallent and Mellor-Crummey 2009).

Coz causal profiler (Curtsinger et al. 2015) locates constructs that could be a suitable candidates for optimization. For every such construct Coz slows down the rest of the program by a predefined rate, thus virtually "speeding up" the candidate. Unlike PERSIK, Coz doesn't analyzes the semantics of the program in order to build a simulation of the system. Instead, Coz evaluates each candidate construct in a separate experiment. It discovers the best candidates by measuring relative change in the program's performance.

Finally, the paper Brünink and Rosenblum (2016) presents an approach to detect distinct performance patterns for the program's functions and then to infer call paths

that correspond to these patterns. This information is used to build models that describe the performance of the system. But unlike PERSIK, the resulting model does not predict the performance in absolute terms. Instead, it is used to produce performance assertions to facilitate regression testing of the program.

Program analysis techniques similar to those described above were used to automatically construct performance models. It was shown that the semantically correct LQN model of the message-passing program can be built from its trace automatically (Israr et al. 2005). Similarly, an automatically constructed LQN model predicted the performance of a document distribution service application with $\varepsilon = 0.3$ (Woodside et al. 2001). A Palladio Component Model model was constructed from the trace of the distributed EJB application; it could predict the CPU utilization and the response time with $\varepsilon \in (0.1 \ldots 0.3)$ (Brosig et al. 2011). Finally, a model of MPI application was built from its trace and demonstrated prediction accuracy ($\varepsilon \leq 0.15$). However its the accuracy dropped to $\varepsilon \in (0.3 \ldots 0.5)$ in configurations where nodes of the system are involved in synchronization operations. Resource demands for such models are usually discovered by instrumenting the program and measuring the resource demands of its individual components (Barham et al. 2004). If direct measurement is not possible, the resource demands can be inferred using a Service Demand Law (Brosig et al. 2009).

Despite a great variety in techniques for automated modeling of computer programs, they share one common feature: most of them are designed to model distributed message-based systems. These techniques do not capture complex thread interaction patterns and resource contention in the multithreaded systems. Consequently, they cannot generate accurate performance models of multithreaded programs.

8 Summary

In this paper we presented a methodology for automatic modeling of complex multithreaded programs. We developed hierarchical models, where different model tiers simulate different factors that affect performance of the program, and interaction between the tiers simulates joint influence of these factors on the performance. This unique architecture allows our models to accurately predict performance of a wide range of multithreaded programs. To implement our models we have developed a PERSIK framework – a discrete-event simulator written using a C++ language.

Building a simulation model of an arbitrary multithreaded program is hard. However, we discovered that analysis of a program is greatly simplified if that program relies on a well-defined implementation of high-level locks and queues. Based on this finding we developed a four-stage methodology to generate performance models automatically. Our methodology relies on a combination of a static and dynamic analyses to discover threads and thread pools in the program, interactions between these threads, operations performed by each thread, and their resource demands. The discovered information is automatically translated into the PERSIK model of the multithreaded program.

We verified our approach by building models of various Java applications, including large industrial programs such as a 3D renderer and a web server. Our models

have average prediction error in (0.032 . . . 0.134) for CPU-intense and (0.262, 0.269) for I/O-intense workloads, which is comparable to results reported by other studies (Ganapathi et al. 2010; Duggan et al. 2011; Huang et al. 2011; Wang et al. 2004; Xu et al. 2005; Nguyen and Apon 2012). At the same time, our framework builds program models automatically and does not require running the program in many configurations. The source code of our framework and generated models is available at (See Footenote 5).

Our next steps will be improving the flexibility of our framework, which will allow predicting performance for a wider range of applications and workloads.

Acknowledgements We thank Dr. Eno Thereska for his insightful comments and feedback on the paper. This work is supported by the National Science Foundation through Grant CCF1130822.

References

Akdere, M., etintemel, U., Riondato, M., Upfal, E., Zdonik, S.B.: Learning-based query performance modeling and prediction. In: Kementsietsidis, A., Salles, M.A.V. (eds.) ICDE, pp. 390–401. IEEE (2012)

Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), pp. 483–485, New York, NY. ACM (1967)

Anderson, E.: Hpl–ssp–2001–4: simple table-based modeling of storage devices (2001)

Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using magpie for request extraction and workload modelling. In: Proceedings of the Symposium on Opearting Systems Design and Implementation, pp. 18–18, Berkeley, CA, USA. USENIX Association (2004)

Barnes, J., Hut, P.: A hierarchical o(n log n) force-calculation algorithm. Nature **324**, 446–449 (1986)

Bause, F.: Queueing petri nets—a formalism for the combined qualitative and quantitative analysis of systems. In: Proceedings of the 5th International Workshop on Petri nets and Performance Models. IEEE, pp. 14–23. IEEE (1993)

Bennani, M., Menasce, D.: Resource allocation for autonomic data centers using analytic performance models. In: Proceedings of International Conference on Automatic Computing, pp. 229–240, Washington, DC, USA. IEEE (2005)

Brosig, F., Huber, N., Kounev, S.: Automated extraction of architecture-level performance models of distributed component-based systems. In: Proceedings of International Conference on Automated Software Engineering, ASE '11, pp. 183–192, Washington, DC, USA. IEEE (2011)

Brosig, F., Kounev, S., Krogmann, K.: Automated extraction of palladio component models from running enterprise java applications. In: Proceedings of the 1st International Workshop on Run-Time Models for Self-managing Systems and Applications, ROSSA'09. ACM, New York, NY, USA (2009)

Brünink, M., Rosenblum, D.S.: Mining performance specifications. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 39–49, New York, NY, USA. ACM (2016)

Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S., Davey, R.A.: A benchmark suite for high performance java. In: Proceedings of ACM Java Grande Conference, pp. 81–88. ACM (1999)

Chen, F., Grundy, J., Schneider, J.-G., Yang, Y., He, Q.: Automated analysis of performance and energy consumption for cloud applications. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ICPE '14, pp. 39–50, New York, NY, USA. ACM (2014)

Chen, J., John, L.K., Kaseridis, D.: Modeling program resource demand using inherent program characteristics. SIGMETRICS Perform. Eval. Rev. **39**(1), 1–12 (2011)

Chen, S., Liu, Y., Gorton, I., Liu, A.: Performance prediction of component-based applications. J. Syst. Softw. **74**(1), 35–43 (2005)

Cheung, L., Golubchik, L., Sha, F.: A study of web services performance prediction: a client's perspective. In: Proceedings of the 19th Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '11, pp. 75–84, Washington, DC, USA. IEEE (2011)

Chun, B.-G., Huang, L., Lee, S., Maniatis, P., Naik, M.: Mantis: predicting system performance through program analysis and modeling. CoRR, abs/1010.0019 (2010)

Coppa, E., Demetrescu, C., Finocchi, I.: Input-sensitive profiling. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'12, pp. 89–98, New York, NY, USA. ACM (2012)

Curtsinger, C., Berger, E.D.: Coz: Finding code that counts with causal profiling. In: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, pp. 184–197, New York, NY, USA. ACM (2015)

Duggan, J., Cetintemel, U., Papaemmanouil, O., Upfal, E.: Performance prediction for concurrent database workloads. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, SIGMOD '11, pp. 337–348, New York, NY, USA. ACM (2011)

Feng, W., Zhang, Y.: A birth-death model for web cache systems: numerical solutions and simulation. In: Proceedings of International Conference on Hybrid Systems and Applications, pp. 272–284 (2008)

Ferrari, D., Serazzi, G., Zeigner, A.: Measurement and Tuning of Computer Systems. Prentice-Hall, Upper Saddle River (1983)

Franks, G., Woodside, M.: Performance of multi-level client-server systems with parallel service operations. In: Proceedings of the 1st International Workshop on Software and Performance, WOSP '98, pp. 120–130, New York, NY, USA. ACM (1998)

Ganapathi, A., Chen, Y., Fox, A., Katz, R., Patterson, D.: Statistics-driven workload modeling for the cloud. In: Proceedings of International Conference on Data Engineering Workshops, pp. 87–92 (2010)

Ganapathi, A., Kuno, H., Dayal, U., Wiener, J. L., Fox, A., Jordan, M., Patterson, D.: Predicting multiple metrics for queries: better decisions enabled by machine learning. In: Proceedings of International Conference on Data Engineering, pp. 592–603, Washington, DC, USA. IEEE (2009)

Goldsmith, S.F., Aiken, A.S., Wilkerson, D.S.: Measuring empirical computational complexity. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE '07, pp. 395–404, New York, NY, USA. ACM (2007)

Gulur, N., Mehendale, M., Manikantan, R., Govindarajan, R.: Anatomy: an analytical model of memory system performance. In: The 2014 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '14, pp. 505–517, New York, NY, USA. ACM (2014)

Happe, J., Westermann, D., Sachs, K., Kapov, L.: Statistical inference of software performance models for parametric performance completions. In: QoSA'10, pp. 20–35 (2010)

Herodotou, H., Babu, S.: Profiling, what-if analysis, and cost-based optimization of mapreduce programs. PVLDB **4**(11), 1111–1122 (2011)

Hoare, C.A.R.: Monitors: an operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974)

Hoecke, S.V., Verdickt, T., Dhoedt, B., Gielen, F., Demeester, P.: Modelling the performance of the web service platform using layered queuing networks. In: Proceedings of International Conference on Software Engineering Research and Practice (2005)

Huang, H., Li, S., Szalay, A., Terzis, A.: Performance modeling and analysis of flash-based storage devices. In: Symposium on Mass Storage Systems and Technologies (MSST), pp. 1 –11 (2011)

Israr, T.A., Lau, D.H., Franks, G., Woodside, M.: Automatic generation of layered queuing software performance models from commonly available traces. In: Proceedings of International Workshop on Software and Performance, WOSP '05, pp. 147–158, New York, NY, USA. ACM (2005)

Jarvis, S.A., Foley, B.P., Isitt, P.J., Spooner, D.P., Rueckert, D., Nudd, G.R.: Performance prediction for a code with data-dependent runtimes. Concurr. Comput. Pract. Exp. **20**, 195–206 (2008)

Kounev, S., Spinner, S., Meier, P.: Introduction to queueing petri nets: modeling formalism, tool support and case studies. In: Proceedings of the Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE '12, pp. 9–18, New York, NY, USA. ACM (2012)

Koziolek, H., Firus, V.: Parametric performance contracts: non-Markovian loop modelling and an experimental evaluation. In: Kuester-Filipe, J., Poernomo, I.H., Reussner, R.H. (eds.) Proceedings of the 5th International Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA'06), volume 176(2) of *ENTCS*, pp. 69–87. Elsevier (2006)

Kristensen, L.M., Christensen, S., Jensen, K.: The practitioner's guide to coloured petri nets. Int. J. Softw. Tools Technol. Transfer **2**, 98–132 (1998)

Law, A.M., Kelton, W.D.: Simulation Modeling and Analysis, 2nd edn. McGraw-Hill Higher Education, New York (1997)

Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative System Performance. Computer System Analysis Using Queuing Network Models. Prentice Hall, Upper Saddle River (1984)

Lee, B.C., Brooks, D.M., de Supinski, B.R., Schulz, M., Singh, K., McKee, S.A.: Methods of inference and learning for performance modeling of parallel applications. In: Proceedings of SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '07, pp. 249–258, New York, NY, USA. ACM (2007)

Liu, Y., Fekete, A., Gorton, I.: Design-level performance prediction of component-based applications. IEEE Trans. Softw. Eng. **31**(11), 928–941 (2005)

Menasce, D.A., Bennani, M.N.: Analytic performance models for single class and multiple class multithreaded software servers. In: International CMG Conference (2006)

Moseley, T., Connors, D.A., Grunwald, D., Peri, R.: Identifying potential parallelism via loop-centric profiling. In: Proceedings of the 4th International Conference on Computing Frontiers, CF '07, pp. 143–152, New York, NY, USA. ACM (2007)

Narayanan, D., Thereska, E., Ailamaki, A.: Continuous resource monitoring for self-predicting DBMS. In: Proceedings of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 239–248, Washington, DC, USA. IEEE (2005)

Nethercote, N., Walsh, R., Fitzhardinge, J.: Building workload characterization tools with valgrind. Invited tutorial, October (2006)

Nguyen, H.Q., Apon, A.: Hierarchical performance measurement and modeling of the linux file system. In: Proceedings of International Conference on Performance Engineering, ICPE '11, pp. 73–84, New York, NY, USA. ACM (2011)

Nguyen, H.Q., Apon, A.: Parallel file system measurement and modeling using colored petri nets. In: Proceedings of International Conference on Performance Engineering, ICPE '12, pp. 229–240, New York, NY, USA. ACM (2012)

Ohmann, T., Herzberg, M., Fiss, S., Halbert, A., Palyart, M., Beschastnikh, I., Brun, Y.: Behavioral resource-aware model inference. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pp. 19–30, New York, NY, USA. ACM (2014)

Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley Professional, Boston (2005)

Reiss, S., Tarvo, A.: Automatic categorization and visualization of lock behavior. In: Proceedings of the First IEEE Working Conference on Software Visualization, VISSOFT '13. IEEE (2013)

Reiss, S.P.: Chet: A system for checking dynamic specifications. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, pages 302–305, Washington, DC, USA, 2004. IEEE Computer Society

Reiss, S.P., Renieris, M.: Encoding program executions. In: Proceedings of the 23rd International Conference on Software Engineering, ICSE '01, pp. 221–230, Washington, DC, USA. IEEE (2001)

Rolia, J., Casale, G., Krishnamurthy, D., Dawson, S., Kraft, S.: Predictive modelling of SAP ERP applications: challenges and solutions. In: Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools, VALUETOOLS '09, pp. 9:1–9:9, ICST, Brussels, Belgium. ICST (2009)

Roy, N., Dabholkar, A., Hamm, N., Dowdy, L.W., Schmidt, D.C.: Modeling software contention using colored petri nets. In: Miller, E.L., Williamson, C.L. (eds.) *MASCOTS*, pp. 317–324. IEEE (2008)

Stein, J.Y.: Digital Signal Processing: A Computer Science Perspective. Wiley, New York (2000)

Strebelow, R., Tribastone, M., Prehofer, C.: Performance modeling of design patterns for distributed computation. In: International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS '12, pp. 251–258, Washington, DC, USA. IEEE (2012)

Tallent, N.R., Mellor-Crummey, J.M.: Effective performance measurement and analysis of multithreaded applications. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '09, pp. 229–240, New York, NY, USA. ACM (2009)

Tarvo, A., Reiss, S.P.: Using computer simulation to predict the performance of multithreaded programs. In: Proceedings of the Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE '12, pp. 217–228, New York, NY, USA. ACM (2012)

Tarvo, A., Reiss, S.P.: Automated analysis of multithreaded programs for performance modeling. In: Proceedings of the 29th International Conference on Automated Software Engineering, ASE '14, pp. 7–18, New York, NY, USA. ACM (2014)

Teng, Q.M., Wang, H.C., Xiao, Z., Sweeney, P.F., Duesterwald, E.: Thor: a performance analysis tool for java applications running on multicore systems. IBM J. Res. Dev. **54**(5), 456–472 (2010)

Thakkar, D., Hassan, A.E., Hamann, G., Flora, P.: A framework for measurement based performance modeling. In: Proceedings of International Workshop on Software and Performance, WOSP '08, pp. 55–66, New York, NY, USA. ACM (2008)

Thereska, E., Doebel, B., Zheng, A.X., Nobel, P.: Practical performance models for complex, popular applications. In: Proceedings of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, SIGMETRICS '10, pp. 1–12, New York, NY, USA. ACM (2010)

Thereska, E., Ganger, G.R.: Ironmodel: robust performance models in the wild. In: Proceedings of International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '08, pp. 253–264, New York, NY, USA. ACM (2008)

van der Mei, R., Hariharan, R., Reeser, P.: Web server performance modeling. Telecommun. Syst. **16**, 361–378 (2001). doi:10.1023/A:1016667027983

Varga, A., Hornig, R.: An overview of the OMNeT++ simulation environment. In: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems and Workshops, Simutools '08, pp. 60:1–60:10, ICST, Brussels, Belgium, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2008)

Wang, M., Au, K., Ailamaki, A., Brockwell, A., Faloutsos, C., Ganger, G.R.: Storage device performance prediction with cart models. In: Proceedings of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS '04, pp. 588–595, Washington, DC, USA. IEEE (2004)

Westermann, D., Happe, J., Krebs, R., Farahbod, R.: Automated inference of goal-oriented performance prediction functions. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, pp. 190–199, New York, NY, USA. ACM (2012)

Woodside, C.M., Hrischuk, C.E., Selic, B., Bayarov, S.: Automated performance modeling of software generated by a design environment. Perform. Eval. **45**(2–3), 107–123 (2001)

Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S.: The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. IEEE Trans. Comput. **44**(1), 20–34 (1995)

Xu, J., Oufimtsev, A., Woodside, M., Murphy, L.: Performance modeling and prediction of enterprise javabeans with layered queuing network templates. In: Proceedings of Conference on Specification and Verification of Component-based Systems, SAVCBS '05, New York, NY, USA. ACM (2005)

Zaparanuks, D., Hauswirth, M.: Algorithmic profiling. ACM SIGPLAN Not. **47**(6), 67–76 (2012)