ASEJ manuscript No.
(will be inserted by the editor)

Automatic Performance Prediction of Multithreaded Programs: a Simulation Approach

Alexander Tarvo · Steven P. Reiss

the date of receipt and acceptance should be inserted later

Abstract The performance of multithreaded programs is often difficult to understand and predict. Multiple threads engage in synchronization operations and use hardware simultaneously. The result is 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 pogram'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 our experiments we show how the models are constructed and show they accurately predict the performance of various multithreaded programs, including complex industrial applications.

Keywords Program analysis, performance, modeling,

Alexander Tarvo
Google, Kirkland, WA E-mail: alextarvo@gmail.com

Steven P. Reiss
Brown University, Providence, RI E-mail: spr@cs.brown.edu

1 Introduction

Multithreaded programs utilize resources of modern hardware more efficiently. However, behavior of multithreaded programs is significantly more complex than behavior of single-threaded applications. Threads rely on synchronization to enforce 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 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 [41], [19], [51]. For example, a model may be used to find a good configuration for deploying the Tomcat web server. For each combination of configuration parameters, including the number of available CPU cores, the number of Tomcat working threads, or 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 the 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.

Building performance models of such applications is hard. First, it requires discovering queues, threads, and locks in the program; details of their behavior; and 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 the 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 resources and locks 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

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.). Second, in order to be fast and easy to understand the resulting model must be simple and compact. Building compact models requires identifying program constructs that do not have significant impact on performance, and excluding these constructs from the model. 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.

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 [56]. We do not model the performance of individual tasks or requests; instead *we 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.

In our work 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 utilization of hardware resources, such as CPU and hard drive. This lets our model answer "what-if"

questions about the program's performance, detect performance anomalies in the running program, and be used as a decision-making element of a self-configuring data center.

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

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, the retrieving 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 Section 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), 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 [45]. It is assumed that the state of the system does not change between time advances.

Our models are built according to the hierarchical principle [29] 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 middle-level models simulate delays that occur inside the program's threads as they process tasks. 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.

3.1 High-level model

The high-level model is based on a queuing network model [46]. Service nodes of the model $\{tr_1, ..., tr_m\}$ correspond to the program's threads (the full notation used to describe the model is provided in the Table 1). Queues $\{q_1, ..., q_n\}$ correspond to the program's queues and buffers used to exchange the tasks between the different components of the software system. This includes queues and buffers present both in the program itself as well as 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 thread group $Tp_j \in \{Tp_1, ..., Tp_k\}, k \leq m$ 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 one of the most important configuration parameters that can significantly affect 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 the web server. The incoming connections are placed into the OS connection queue q_1, from which they are fetched by the accept thread tr_1. tr_1 forms a task object, which represents the HTTP request, and places that task into the program's task queue q_2. One of the working threads tr_2,...,tr_n fetches the task from the queue q_2 and processes the request.

Our model differs in important ways from the classical queuing networks. First, it does not restrict the structure of the model, the number of service nodes, or distribution families of 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.

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 (CF)*. The special vertex s_0 corresponds to the code fragment executed upon a thread start.

Edges 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

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

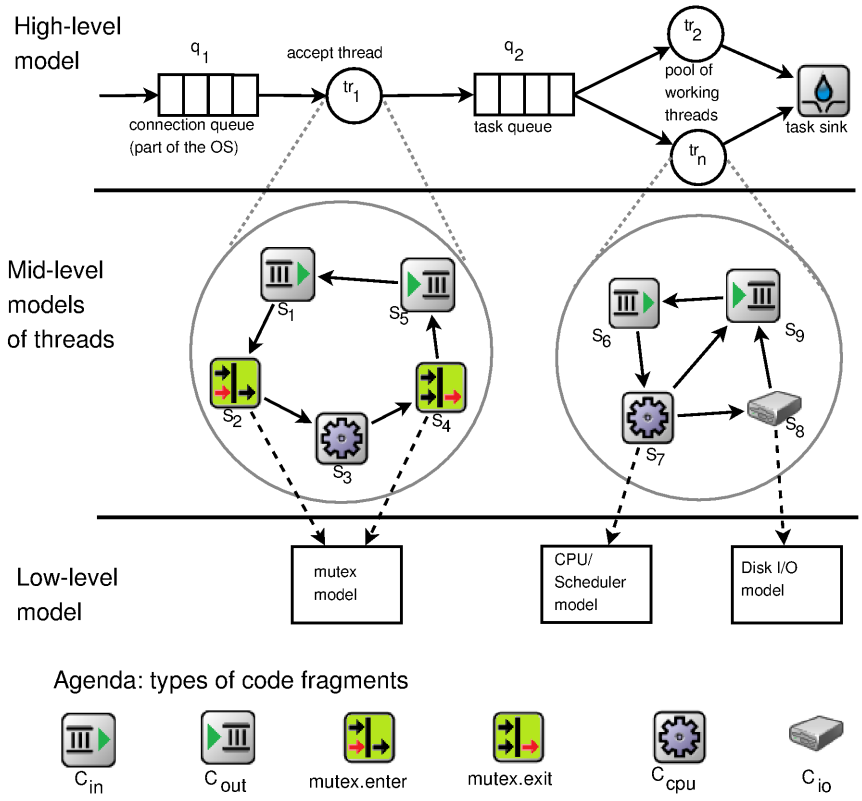

Fig. 1 A model for a web server

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

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 queues.

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

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

Figure 1(middle) deipcts 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. 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 if 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 and the OS thread scheduler, the disk I/O subsystem, and the set $L = \{l_1, ..., l_m\}$ 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 τ_i we describe 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 simulate 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. Below we describe modeling different types of computations in detail.

Modeling CPU computations. CPU computations and memory operations are simulated by the c_{CPU} computation CFs. The parameter of a computation CF $\Pi_{CPU} = \langle \tau_{CPU} \rangle$ 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 τ_{CPU} fluctuates across different executions of s_i, Π_{CPU} is represented as a distribution of CPU times $\mathbb{P}^{\Pi}_{CPU}$.

When the thread model has to compute τ for the computation CF, it samples τ_{CPU} 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 $S_{CPU} = \tau_{CPU}$ 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 $min(\tau_{CPU}, OStimequantum)$. 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.

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 distri-

bution are tuples $\Pi_{disk} = \langle dio_1, ..., dio_k \rangle$ 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 "synchronous read" or "readahead".

The number k of I/O operations allows to implicitly simulate the OS page cache. It was shown [28] 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 $\langle dio_1, ..., dio_k \rangle$ 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.

Disk I/O model is a queuing model whose queue represents the request queue in the actual 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;
- *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 do not explicitly simulate write I/O operations at the moment, 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 $\Pi_{lock} = \langle l_j, optype, \tau_{sync} \rangle$, where

- $l_j \in L$ is the synchronization construct (lock) that is called;
- *optype* is the synchronization operation performed on a lock. Possible values of *optype* depend on the type of the lock. For example, the possible values of *optype* for a mutex are $\{acquire, release\}$, and for a barrier *optype* is $\{await\}$;
- $\tau_{sync} \in (0, ..., \infty)$ is the timeout for synchronization operation. By default timeout is $\tau_{sync} = \infty$, which denotes the infinite timeout. Correspondingly, $\tau_{sync} = 0$ 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.

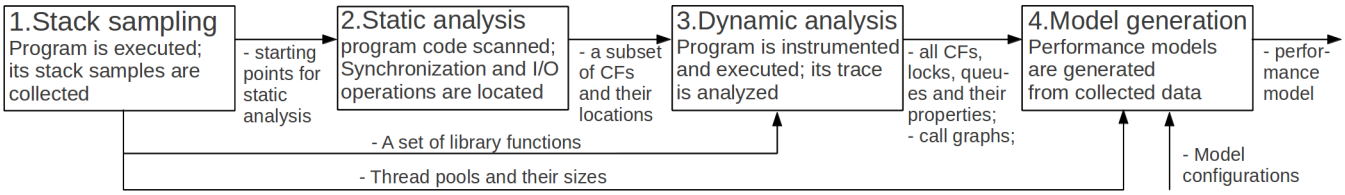

Fig. 2 Model creation stages and intermediate results

The lower-level model explicitly simulates behavior of each lock $\{l_1, ..., l_m\} \in L$ in the program. 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 the barrier indicates the barrier capacity, the parameter of the semaphore is the number of permits, and the mutex has no parameters.

Fetching and sending a task to a queue are simulated by c_{in} and c_{out} code fragments. Their parameters $\Pi_{inout} = \langle qid, optype, \tau_{inout} \rangle$ are the ID of the queue being accessed, type of the operation such as $\{fetch, send\}$, and the optional timeout.

4 Automatic Model Generation

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

- The set $q_1, ..., q_n$ of queues and buffers used to exchange tasks between program's threads. These correspond to the queues in the high-level model;
- The set $tr_1, ..., tr_m$ of threads in the program. Threads correspond to the service nodes of the high-level model;
- The set $Tp_1, ..., Tp_k$ 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 computations, 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 CFs, required to model delays τ;
- The set L of locks in the low-level model, their types, and parameters Π_{lock}.

We collect required data 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 Figure 2). Each stage saves intermediate results into files that are used as input to subsequent stages.

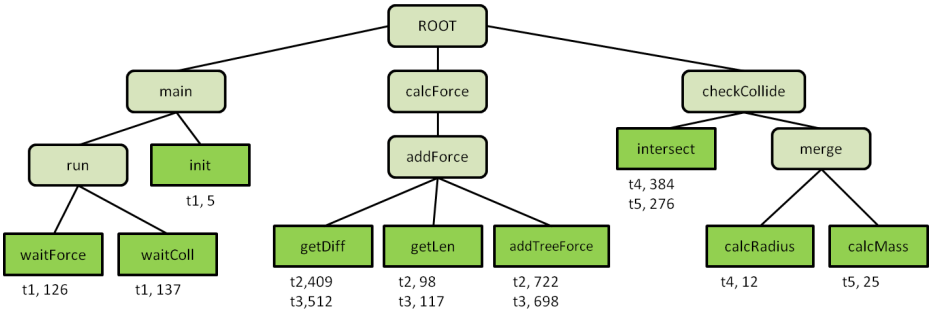

Fig. 3 An example of a call trie

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 we detect c_{sync}, c_{in}, c_{out}, and c_{IO} CFs. Third, the program is instrumented and executed again with the same configuration. 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, frequently called functions and methods in the program, and frequently called library functions. Identifying libraries is essential for generating correct probabilistic call graphs (see Section 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 being executed, and a line number. The non-leaf nodes provide a call stack for that code location. For each leaf the framework maintains the 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 Figure 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.

Thread groups are detected in two steps. First a map $\mathbb{T}$ is created. Its keys are thread tuples discovered by sampling, and its values are execution counts. For each leaf in the trie the framework retrieves a tuple $Tp_i = \langle tr_1, \ldots tr_k \rangle$ of threads that executed the node along with the total number of executions $Ex_i = \sum(ex_1, \ldots, ex_k)$. If $\mathbb{T}$ does not contains the tuple Tp_i, the pair $\langle Tp_i, Ex_i \rangle$ is inserted

into $\mathbb{T}$. Otherwise the number of executions for the existing tuple is increased by Ex_i.

In our example the following tuples are created:

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

The resulting tuples represent the thread pools that can be possibly found in the program. 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 $t5$ and $t4$ 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 the spurious thread tuples in in $\mathbb{T}$ are detected are merged with the correct ones. The tuple $\langle Tp_1, Ex_1 \rangle$ is considered a spurious one and can be 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 Figure 3, the tuple Tp_4 and Tp_5 is 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 a method f the framework generates the set of functions $\langle f_1, ..., f_{ncall} \rangle$ 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). 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 [37].

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. Resulting implementations often had inferior performance and were 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 [1].

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.

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. 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 (probeID).

Probes report the timestamp, the probeID, 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 ⟨*start probe ID, end probe ID*⟩. Every such pair represents an execution of a single code fragment.

The ⟨*start probe ID, end probe ID*⟩ 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.

TODO: should we provide a code example for this trace?

The Figure 4 depicts an example of such trace. Here the CF ⟨10, 11⟩ is a c_{in} CF. The object ID=7683745 recorded by the probe 10 identifies the queue, while the argument value 0 correspond to the timeout of 0 milliseconds. The probe 11

ProbeID	Timestamp	ObjectID	Arguments/ return value
10	11345231	7683745	0
11	11387461	7683745	4387459
27	11391365	87235467	
28	11392132		
10205	11396190	1872565	
10206	19756012	1872565	
6	19873872	87235467	
7	19873991		
10205	19923752	32748998	
10206	25576572	32748998	
...			

Fig. 4 A fragment of the trace for a thread.

reports the return value 4387459, which is an ID of the retrieved object. ⟨27, 28⟩ and ⟨6, 7⟩ are synchronization CFs corresponding to the entry and exit from the synchronized region. The object ID=87235467 and 32748998 identifies the monitor associated with that region. Two instances of ⟨10205, 10206⟩ I/O CF correspond to two (unrelated) file read operations from the disk. Their object IDs 1872565 identify the instances of the corresponding file objects. 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 $s_1 \dots s_n$ of CFs discovered in the trace as the set S of nodes in the PEG. For each node $s_i \in S$ the subset $S_{next} = \{s_k, \dots, s_m\}$ of succeeding nodes is retrieved, along with the numbers of occurrences of the pairs $(s_i, s_k), \dots, (s_i, s_m)$ in the trace. The probability of transition from the node s_i to $s_j, j \in (k \dots m)$ 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 $\delta : S \to P(S)$ 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 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.

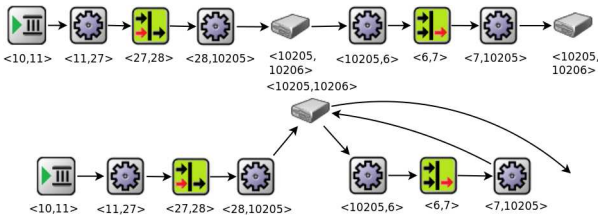

Fig. 5 Top: the ground truth PEG from the thread trace. Bottom: the incorrect PEG generated from the trace that contains a library call.

For example, according to the trace shown on the Figure 4 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 Figure 5, top). However, both I/O operations will eventually call the same read() I/O API that contains an ⟨10205, 10206⟩ I/O CF. As a result, the generated PEG will contain a loop in it (see Figure 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 [59]. 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 Section 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. 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.

TODO: Add an example from the trace we shown above?

This leads to an unnecessary complex PEG, consisting of thousands of CFs (see Table 3). Such complex models have low performance and are hard to analyze. 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 and eliminating those which do not have a noticeable impact on performance.

During the first step the whole timeline 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. In fact, when the program enters the shutdown phase, all the performance information has been already collected and there is no need to further simulate the program.

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 $t = 3 - 5\%$ allows shrinking the PCG by 50-70% without noticeable impact on the accuracy.

Accounting for determinism in the program behavior. Some program behaviors 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. Namely, the program will attempt to fetch the blocking queue and impose a timeout for the operation. Depending on if 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 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. 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 if the loop performed an initial population of 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 millisecond, the average running time of the program will be 25 milliseconds. 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 milliseconds, which corresponds to the mean prediction error $\overline{\varepsilon}(T) = 0.80$.

To address this issue the dynamic analyzer detects loops in the trace using the algorithm [50]. If the loop contains the c_{out} node, the model explicitly simulates it. Otherwise the loop is represented as a cycle in the PEG.

4.3.2 Retrieving parameters of code fragments

The dynamic 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 of these locks and queues, and from their return values. As we mentioned earlier, the lock type *ltype* is inferred from the signature of the constructor/intializer of that lock during the statuc analysis (see Section 4.2). The type-specific parameters *lparam* are retrieved from the values of arguments passed to that constructor. 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, in a Java program the capacity of the barrier is specified by the value `parties` argument of the `CyclicBarrier(int parties)` constructor. Correspondingly, the capacity of the queue is specified as the `capacity` argument of the `ArrayBlockingQueue(int capacity)`. The ID of the object returned by the constructor uniquely identifies the corresponding lock or queue.

Parameters of $\mathbf{c_{sync}}$, $\mathbf{c_{in}}$, and $\mathbf{c_{out}}$ 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* 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 τ_{out} 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 $\mathbb{P}^{\tau}_{CPU}$ of CPU times τ_{CPU}. In a general case the τ_{CPU} for a code fragment can be obtained as a difference $\tau^{end}_{CPU} - \tau^{start}_{CPU}$ between the thread CPU time τ^{start}_{CPU} measured before executing the CF and the thread CPU time τ^{end}_{CPU} measured after executing the CF. Here thread CPU time denotes the amount of time the CPU was executing instructions of that CF.

τ_{CPU} can be accurately measured when the execution time of a thread can be determined. When this is not the case, τ_{CPU} 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.

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 $\{dio_1, ..., dio_k\}$ initiated by that c_{IO} CF. This request-specific data can be retrieved only from the OS kernel. We used the blktrace [2] 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 [64]. 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 $U = \{u(t)_1 \dots u(t)_N\}$ for each user-mode thread obtained from the program trace, and kernel I/O signals $B = \{b(t)_1 \dots b(t)_M\}$ for each kernel-mode thread from the blktrace log. The analyzer discretizes those signals with the sampling interval of one millisecond.

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 $b(t)$.

TODO: fix the notation on the image

The dynamic analyzer matches user to the 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 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 opera-

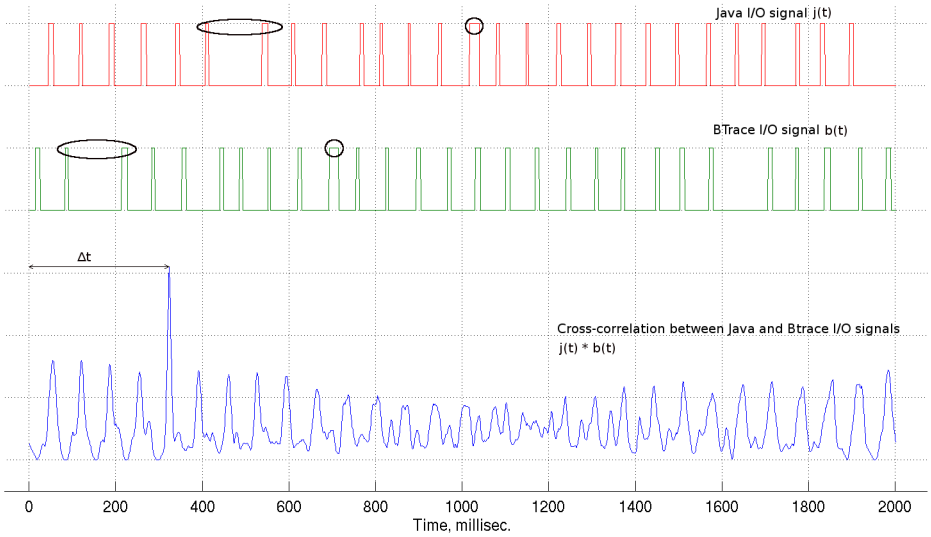

Fig. 6 Cross-correlation between Java and Btrace I/O logs. Distinctive features of the signals are highlighted by circles.

tions 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 $dio_j ... dio_{j+k}$ 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

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}. This information is used to generate the three-tier performance models described in the Section 3. The models are implemented using the OMNeT simulation toolset [3] and can reviewed in the OMNeT IDE.

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

We implemented our approach as a tool for automatically building models of Java programs. The tool uses ASM [4] framework for bytecode analysis and instrumentation.

In this section we present experimental evaluation of our methodology for automatic generation of performance models. The main evaluation criteria is prediction accuracy of the generated models.

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 [22] 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. The 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. The 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 2 present a summary on these programs and their models. Below we briefly describe these programs, along with results of their simulations.

Table 2 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

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 at a resolution of N x N pixels. The overall time required to render the frame is the most important performance metric of Raytracer. Assuming the constant size of the image, the number of working threads is is a determining factor for 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 Figure 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 Figure 8). These results demonstrate good prediction accuracy for both hardware configurations.

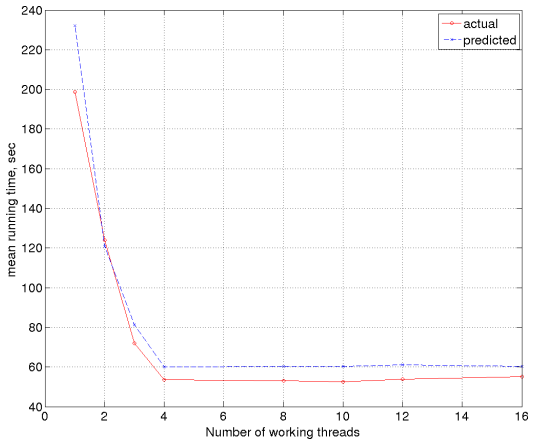

Fig. 7 Predicted and measured running time for Raytracer in hardware Config I

Montecarlo simulates price of marked derivatives based on the prices of the underlying assets. Using historical data on asset prices, the program generates a

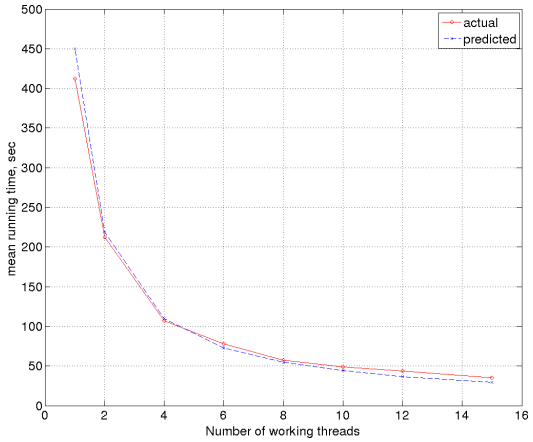

Fig. 8 Predicted and measured running time for Raytracer in hardware Config II

number of time series using 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 Figure 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 Figure 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 Linux *perf* utility. It appeared that the Montecarlo performs a large number 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 can be addressed by more detailed simulation of memory operations, which involve collecting the information on memory accesses by the program and by 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

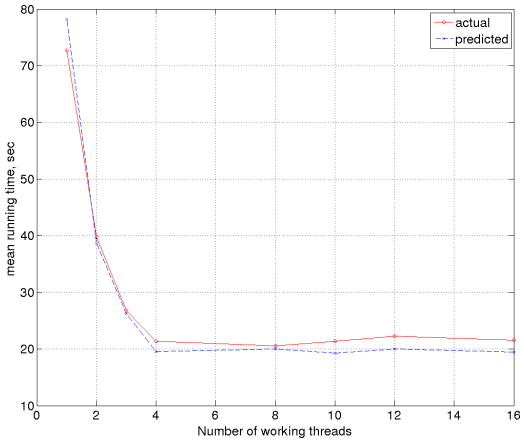

Fig. 9 Predicted and measured running time for Montecarlo in hardware Config I

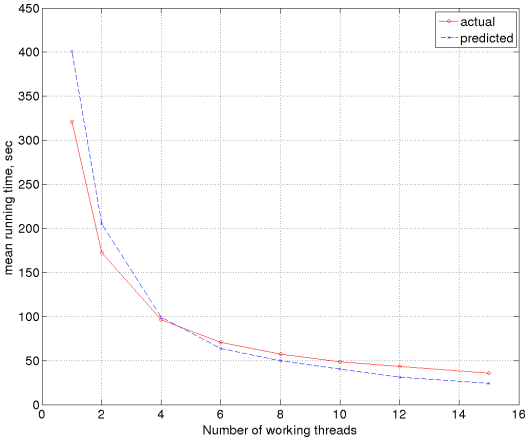

Fig. 10 Predicted and measured running time for Montecarlo in hardware Config II. Contention of the memory bus has some impact on performance.

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 number of working threads in 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 configurations. 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 Figure 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 Figure 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% (5x 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 Section 6.3.

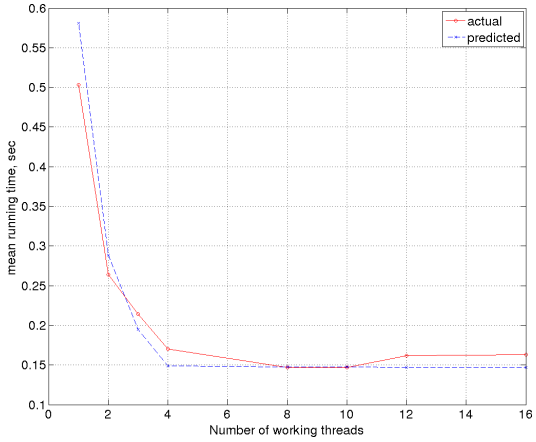

Fig. 11 Predicted and measured iteration length for Moldyn in hardware Config I.

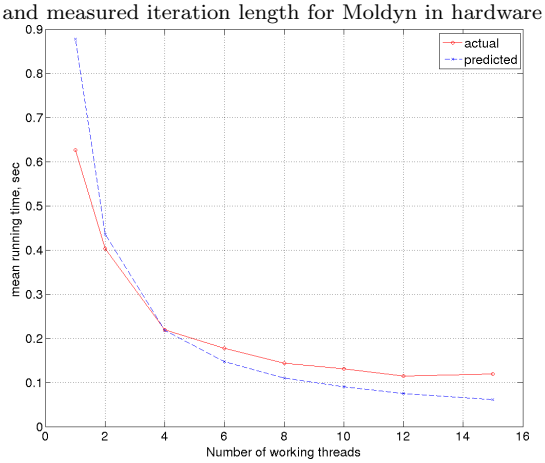

Fig. 12 Predicted and measured iteration length for Moldyn in hardware Config II. Variations in cache miss rate have noticeable influence on performance.

Galaxy simulates the gravitational interaction of celestial bodies using the Barnes-Hut [16] 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 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.

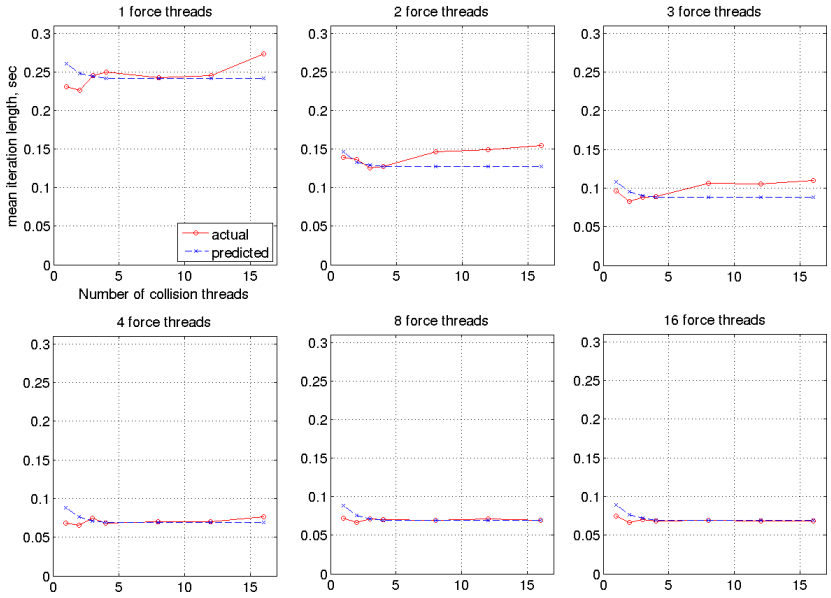

Fig. 13 Predicted and measured iteration length for Galaxy program on a 4-core machine. Impact of collision threads on performance is minimal.

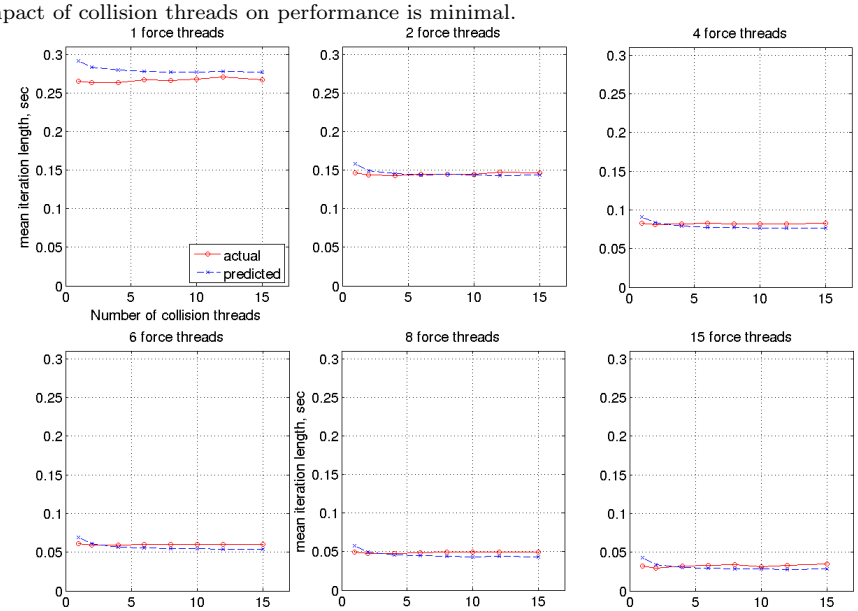

Fig. 14 Predicted and measured iteration length for Galaxy program on a 16-core machine. Increase in the number of working threads doesn't lead to the proportional improvement in performance.

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 Figure 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 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 if 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 [13]. Namely, 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, whose impact on performance becomes noticeable as the number of threads grow.

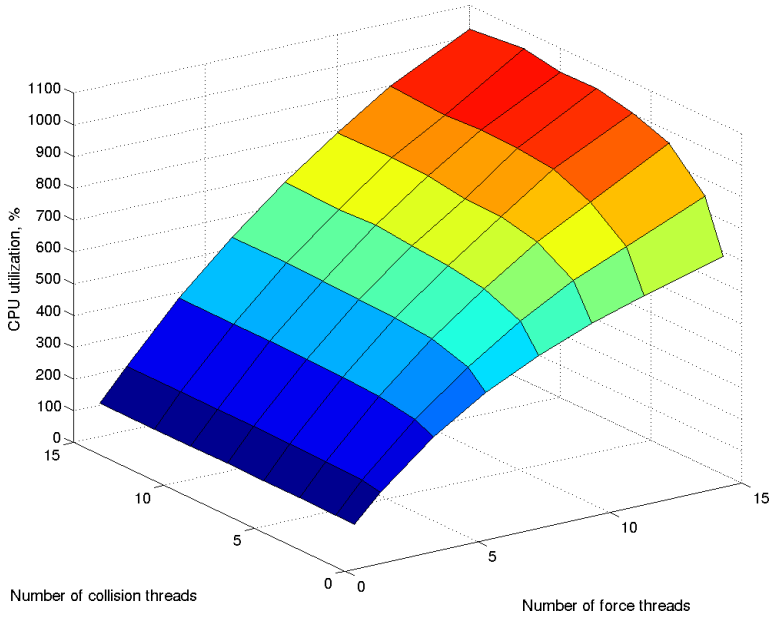

Fig. 15 Predicted CPU utilization for Galaxy in Config II. All CPU cores are never utilized due to the incomplete parallelization of the workload.

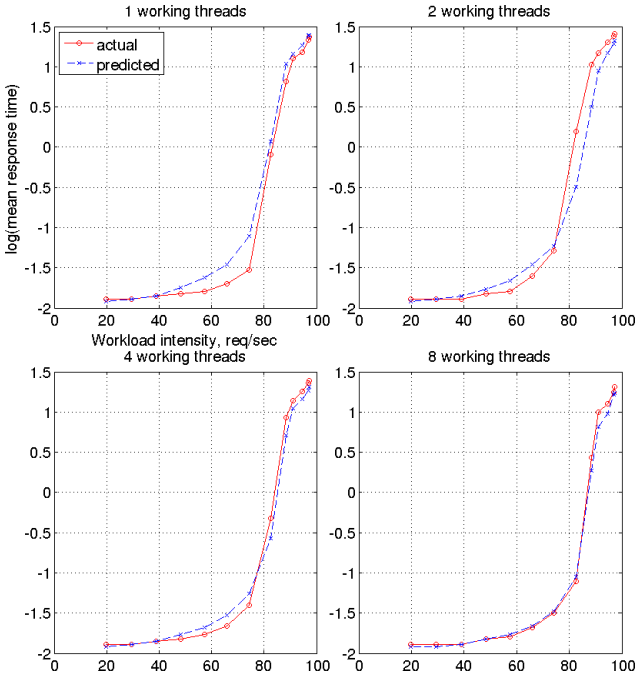

Fig. 16 Predicted and measured response time for Tornado

This analysis is reinforced with the prediction of CPU usage by the Galaxy in Config II (see Figure 15). In particular, we correctly predict that all the CPU cores are never 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 Section 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. In our experiments Tornado was deployed in hardware Config I and used to host about 200000 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 Figure 16. Predictions of the throughput are shown at the Figure 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

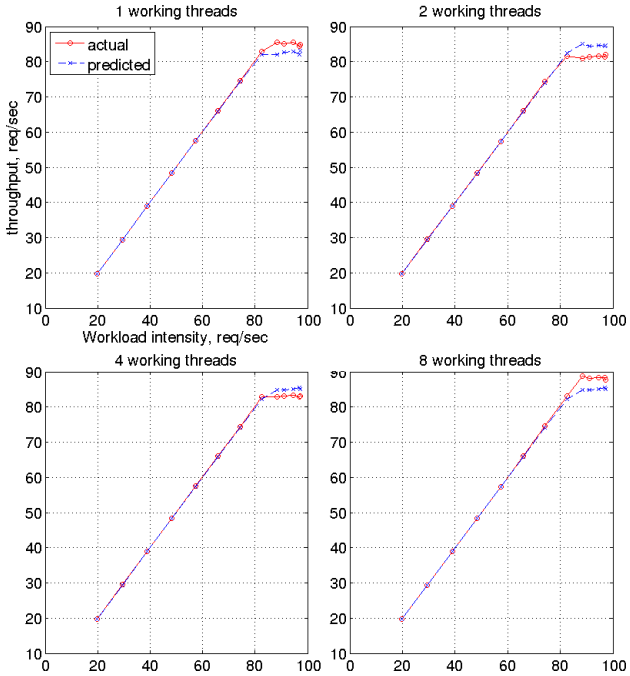

Fig. 17 Predicted and measured throughput for Tornado. The number of working threads has a weak impact on performance due to hard drive contention.

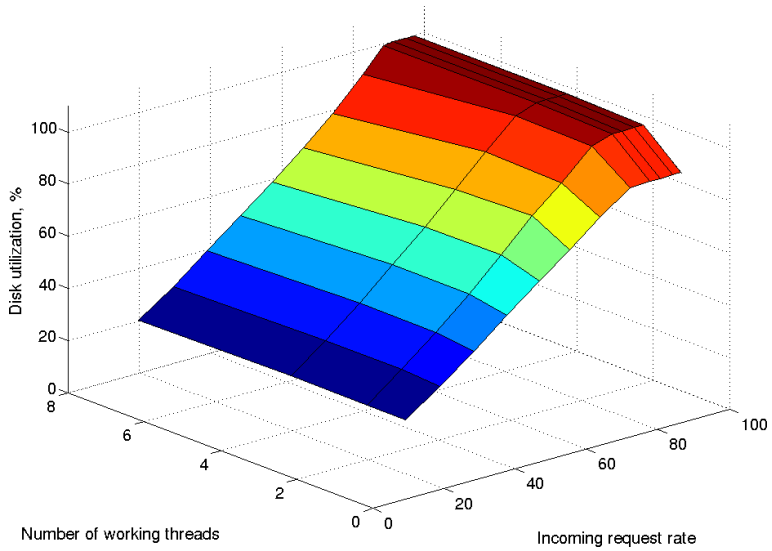

Fig. 18 Predicted utilization of the hard drive by Tornado. A single hard drive becomes a bottleneck in the system.

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 reviews these effects in more details. 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 increase in the average execution time for each I/O request (see Figure 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 programs we have studied in a previous section. To prove the practical value of our methodology we must demonstrate that our framework is capable of building accurate models of large industrial applications. We built models of the following large open-source programs: Sunflow 0.07 3D renderer and Apache Tomcat 7.0 web server.

We predicted the performance of Tomcat in two setups: as a standalone web server hosting static web pages and as a servlet container that hosts an iText library for text conversion. Considering difference in Tomcat functionality over these setups, corresponding models are significantly different. Table 3 provides information on programs and their models.

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

The complexity reduction algorithm eliminated 99% to 99.5% of all CFs as insignificant in 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.

Our models run 8-1000 times faster than the actual program (see Table 3). 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.

Table 3 Large programs and their models

	Tomcat (web server)	Tomcat+iText (servlet container)	Sunflow
Program size (LOC)	182810	283143	21987
Number of probes	3178	3926	380
Mean instrumentation overhead	7.3%	2.4%	5.7%
Number of CFs	11206	9993	209
Total number of nodes in the model	82	49	42
Simulation speedup	8-26	37-110	1050

All the experiments with large applications were performed in the hardware Config I, with a quad-core CPU. 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 [5]. 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 $\overline{\varepsilon} = 0.032$.

Our framework could accurately predict performance of Sunflow across different hardware configurations. 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. Nevertheless, it opens a path for such a prediction because CF timing can be estimated by an analytic CPU model (e.g. by applying scaling coefficients to execution times of CFs) or by using microbenchmarks on the target architecture.

Apache Tomcat as a web server. Apache Tomcat is a web server and Java servlet container [6]. Thanks to its reliability, flexibility, and high performance Tomcat is widely used in industry; more than a half of Fortune 500 companies reportedly use Tomcat in their business [7]. However, these Tomcat features come at the cost of the high internal complexity. Tomcat consists of over 200000 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,

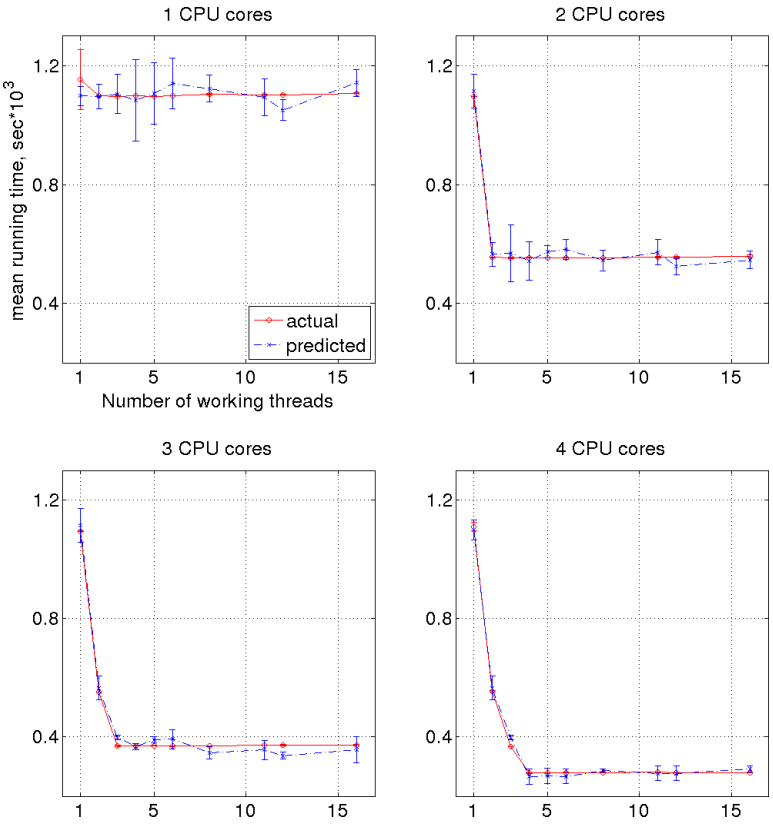

Fig. 19 Predicted and measured performance of Sunflow. Good accuracy for configurations involving under- and over-utilization of resources

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 about 600000 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 with and workload intensity ranging from 48.3 to 156.2 req/s. During each run 10000 requests were issued.

The prediction results for R and T are depicted at the Figures 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 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 $\overline{\varepsilon}(R) = 0.269$. Similarly 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 a configuration space of Tomcat. According to our measurements, mean $\overline{k} = 0.755$ with 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.

To verify our assumption about 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.

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 in a web server setup Tomcat. 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 using the iText [8] 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 number of working threads ranging from 1 o 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 Figure 22, and results for T are depicted at the Figure 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 τ_{CPU} fluctuates less than the demand for I/O bandwidth, which leads to the lower prediction error 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

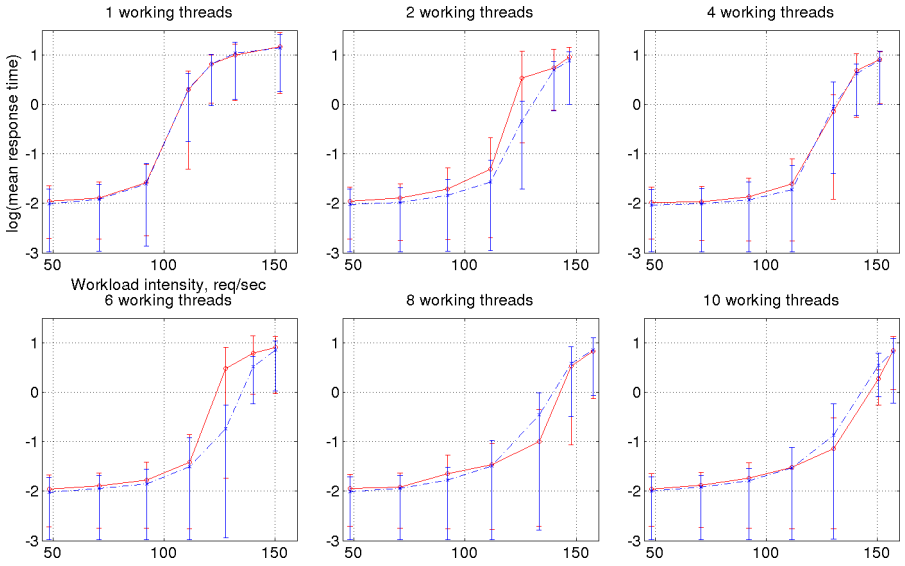

Fig. 20 Response time of Tomcat in a web server setup. Small variation in demand for I/O bandwidth lead to large changes in the response time.

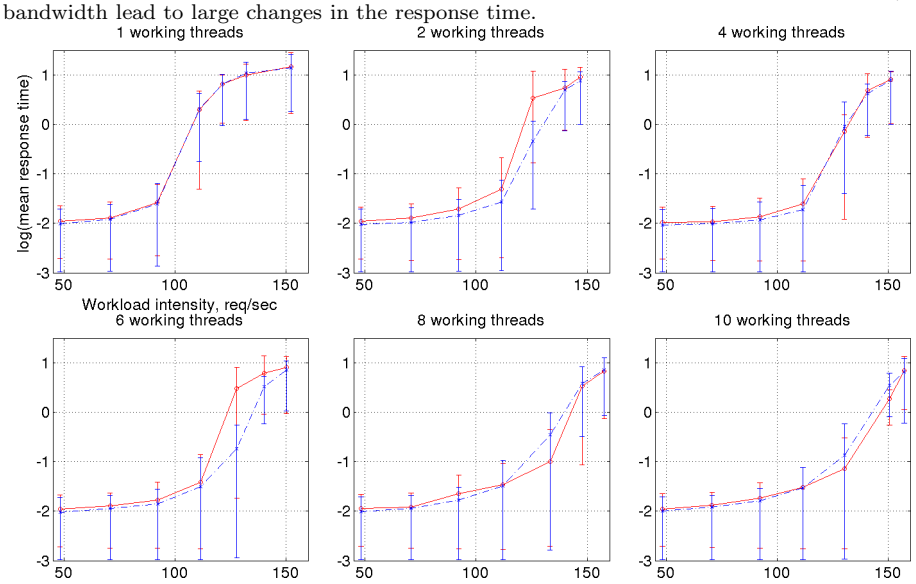

Fig. 21 Throughput of Tomcat in a web server setup. Configurations leading to server saturation are detected accurately.

ranges from 21.4 req/sec for a configuration with 1 thread to 85.5 req/sec for 8 threads.

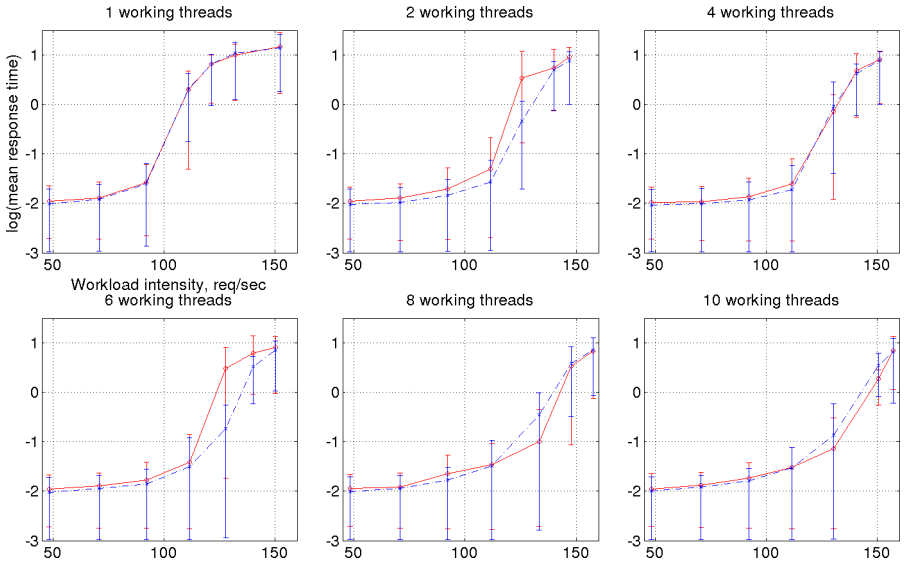

Fig. 22 Response time of Tomcat in a servlet container setup. Consistent demand for the CPU time leads to an accurate prediction.

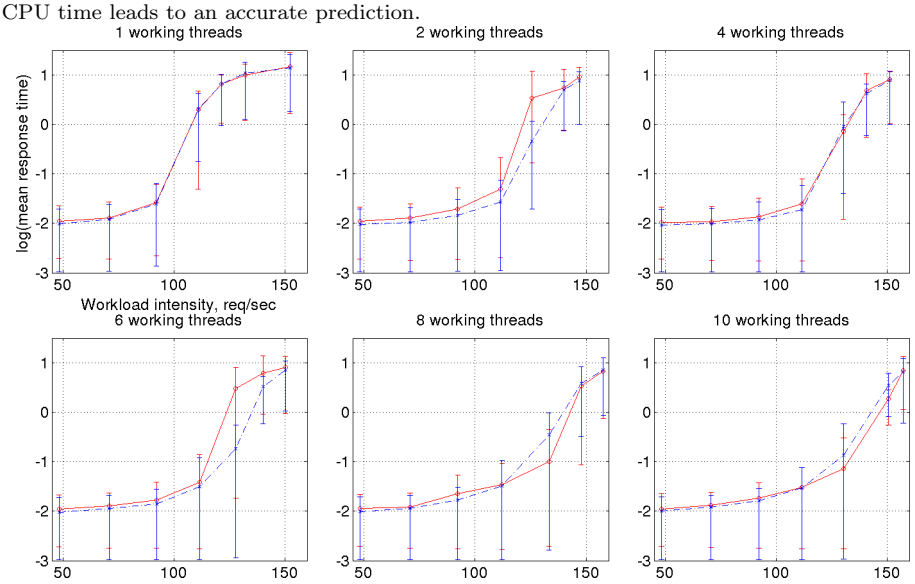

Fig. 23 Throughput of Tomcat in a servlet container setup.

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 would result 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 semantics of thread interaction can be a hard problem in a 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 demands for the elements of the program. Small errors in measuring resource demands may lead to large prediction errors. However, obtaining precise measurements of resource demands without introducing a significant overhead is difficult. Moreover, resource demands can vary in time, leading to decrease in prediction accuracy.

We found that in order to be fast and easy to understand the resulting model must be simple and compact. 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 actual performance. Although we use a simple step-by-step procedure for locating bugs in models, developing tools and approaches 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.

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.

During data collection the program is executed in a single representative configuration, where the transition probabilities δ and CF parameters Π would be similar to δ and Π of a larger set of configurations. This requires the usage patterns for the program, such as the image resolution in Sunflow or probabilities of accessing individual web pages in Tomcat, to remain similar across the configuration space. Changing usage patterns may require reconstructing the model.

Our framework in its present state can 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 [1]. Moreover, semantics of locks implemented using low-level constructs can be discovered using analysis described in [57]. 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.

Our models do not explicitly simulate calls made by the program to other systems, such as Web services or SQL databases.

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 an image size in Sunflow or the number of particles in Moldyn are described 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 other 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 $\Pi_{CPU} = \tau_{CPU}$ for a computational CF could be defined as $\tau_{CPU} = f(X_{CPU}, X_{cf})$, 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 $\tau_{CPU} = f(X_{CPU}, X_{cf})$ 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 [25]. 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. In a cloud environment programs are executed in a variety of configurations, which naturally provides data 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 case when 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 allows to 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 [52] [34], 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, they 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 [9] or NS3 [10] 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.

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 $y = f(\mathbf{x})$: analytical models, statistical models, and simulation.

Analytic models explicitly represent the dependency $y = f(\mathbf{x})$ using a set of equations. Narayanan, Thereska and Ailamaki used analytic model to predict the dependency between the size of the DBMS cache and its response time R and throughput T [51]; the reported relative errors are $\varepsilon(T) \leq 0.1$ and $\varepsilon(R) \in (0.33...0.68)$.

Herodotou and Babu developed a set of analytical performance models to predict the running time of MapReduce tasks [36]. Authors reconstruct the profile of the task using dynamic analysis and pass the profile to the "what-if" prediction engine. The prediction engine relies on simulation and analytical models to evaluate performance of the task for the given configuration.

Chen, John and Kaseridis used such model to predict utilization of the L2 cache and memory bandwidth for a given program on a multiprocessor system. Their model report average error in (0.09 to 0.13). Bennani and Menasce [19] developed the analytical model of a transaction processing system to detect configurations resulting in its high performance. The model was used as a central element of the autonomic data center Strebelow at al. employed analytic models to study performance of certain multithreaded design patterns [65].

Analytical models are compact and expressive; however they require knowledge of system's functionality and a substantial mathematical skill to formalize this functionality using a set of equations. In particular, complex behavior is difficult to express with the analytical model. Moreover, analytically modeling even a simple multithreaded system is challenging [48]. Nevertheless, analytical models can be used as a part of the larger model to predict performance for some of the system's components. For example, Thereska and Narayanan [70] uses analytical models as a part of the larger model to simulate individual components of the distributed system, such as network and disk.

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

Statistical models are a popular approach to predicting the performance of SQL databases. Ganapathi et al used a statistical model to predict the running time of SQL queries [32]. Authors construct the x vector from the DBMS query plan and use a k-NN technique to predict performance for a given query. The correlation between the actual and predicted execution times $R^2 \in (0.55 \dots 0.95)$. Authors further extend this technique to predict the running time of Hadoop tasks [31]. The x vector included metrics such as the number of bytes written during different phases of the task. This study reports correlation $R^2 \in (0.87 \dots 0.93)$

Akdere et al used a similar approach to predict performance of the SQL queries running in isolation [12]. The x vector can be built from individual operators of SQL query, which allows to train models on-line. Authors developed an iterative procedure to select relevant queries for training. The resulting model have relative error in a range of $(0.05 \dots 0.1)$. Finally, Duggan et al predict individual running

time of a mix of concurrently running queries [27] using a multivariate linear regression. The resulting model has accuracy $\varepsilon \in (0.14 \dots 0.27)$.

Apart for predicting performance of queries, statistical models are used to predict performance for a wide range of systems. Happe et al employed a non-linear regression for predicting dependency between the response time in the message-passing middleware software and the size of the message [35]. Lee et al used linear regression and neural networks to predict the running time of scientific computing applications on a large grid system [47]. Their feature vector x included both parameters of the task and the configuration of the program. The relative prediction error varied in $\varepsilon \in (0.01, ..., 0.25)$.

Although statistical models do not require knowledge of system's internals, they have limitations. Building statistical models require running the system in many configurations, which is time-consuming and costly. Any change to the software or hardware of the system requires re-training the whole model [68]. Finally, the accuracy of a statistical model strongly depends on the representativeness of the training dataset. Cheung et al demonstrated that although his statistical model based on a non-linear regression has good accuracy in extrapolating the performance of the system (predicting performance within the ranges of configuration parameters used for model training), the interpolation accuracy (predicting performance for a point outside the training dataset) can be very low [24].

There are attempts to alleviate these shortcomings. Statistical models can be built when large amounts of data are already available, e.g. from the prior runs in the cloud environment or from a large user base [69]. Sophisticated program analysis and machine learning techniques can help reducing the amount of training data. In particular, Chun et al. [25] use internal program features such as values of variables, loop and branch counts as metrics $\mathbf{x}$, which allows to reduce the size of training set by 50%. Westermann et al developed a methodology for iterative selection of points into the training set [73]. Their approach allows to reduce the size of the set, although both error terms and the number of points strongly depend on the selection algorithm

Finally, statistical models can be successfully employed for those scenarios when the training dataset can be collected relatively quickly, e.g. by benchmarking. Thus statistical models can become a feasible approach for modeling individual components of the large system, such as disk I/O subsystem. Huang et. al. used CART trees to predict performance of the SSD disk with $\varepsilon \in (0.17 \dots 0.25)$ [40]. Wang et al built a regression tree model of a traditional hard drive with $\varepsilon \in (0.17 \dots 0.38)$ [72]. Anderson used k-NN algorithm to predict running time of disk I/O operations; the accuracy of the model is $\varepsilon \in (0.02 \dots 0.2)$ [14]. The size of the request, its sequentiality, and length of the I/O sheduler queue are used as predictor $\mathbf{X}$

Simulation models, such as queuing networks, Petri nets, and their extensions mimic the behavior and/or structure of the system. These models proved to be the most flexible and capable techniques for modeling complex systems.

Although some of these models can be solved analytically, simulation remains the main tool for predicting performance using simulation models. Simulation can represent complex behavior of the system. Building a simulation model does not require running the system in many configurations. However, constructing simulation models require knowledge of the components of the system and their properties.

A variety of formal methods for building simulations have been developed. The first such methodology was queuing networks [46]. In particular, van der Mei et. al. used queuing networks to model impact of networking parameters at the performance of the web server [71]. 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 [75] [62]. In a LQN the queue and the server are united in a single node. The nodes can represent different computers (e.g. the client and the server), software components of the system, as well as the hardware components, such as disk and CPU. The role of the node determines its position within the network. In particular, nodes representing hardware components are placed at the lowest level of the network.

LQNs can be solved analytically and are particularly useful for simulation of distributed systems. Van Hoecke et al relied on the LQN to build models of simple CORBA applications and web services [38] with $\varepsilon \in (0.02 \dots 0.05)$. Rolia et al use LQN to predict performance of the CPU-bound ERP application with accuracy $\varepsilon = 0.15$, although their system did not carry out any I/O or synchronization activities [60]. However, analytic modeling of complex threading behavior with LQN [30] may be challenging.

Similarly, Palladio Component Models (PCM) is a novel approach to simulation, where the system is divided into a number of interconnected components [18].

Another well-known simulation methodology is Petri nets and their extensions. One of the most widely used extensions is the colored Petri nets [44] that allow assigning values (denoted as colors) to the tokens. Roy et al used colored Petri nets to model performance of a simple multithreaded scientific computing application [63].

In [54] Colored Petri Net (CPN) predicted performance of a parallel file system with $\varepsilon \in (0.2 \dots 0.4)$, and in [63] CPN was used to simulating the complex locking constructs in a program. Nguen and Apon [53] used colored Petri nets to build the model of a Linux Ext3 file system. Their implementation simulates read and write operations, system's page cache, and the filesystem journal. The model predict the average throughput of the system with $\varepsilon \in (0.12 \dots 0.34)$. This study was extended [54] to allow simulation of the parallel file system with $\varepsilon \in (0.2 \dots 0.4)$.

Queuing Petri Nets (QPN) extend the Colored Petri nets by adding queuing and timing aspects into the model [17]. Kounev, Spinner, and Meier used Queuing Petri Nets to simulate distributed component-based and event-based systems [43].

In addition to the models that use some formal method or its extension, certain approaches rely on a combination of different formal methods or propose their own modeling paradigms.

For example, IRONModel by Thereska and Ganger [70] relies on a queuing network to simulate the flow of the request through a distributed system, and uses analytical models to simulate performance of certain components, such as the network and hard drive. PACE framework by Jarvis et al employs hierarchical approach for building models of MPI appilcations [42]. In PACE the high-level model represents the program as a whole, the middle-level models represent code templates within the program, and lower-level models represent underlying hard-

ware. PACE was used to predict the execution time of the `nreg` medical image processing application with the $\varepsilon <= 0.1$ [42].

Simulation models are more flexible then analytical 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 our model to handle a larger variety of multithreaded programs.

7.2 Automatic analysis and performance modeling of computer programs

Simulation models for performance prediciton are more flexible than analytical or statistical models. Their construction is significantly more difficult though, mostly because they require extensive information on the system's internals and functionality. This information can be retrieved manually, as in [75], [71], [38], [76]. However, the manual analysis of the software system is time-consuming and error-prone. Furthermore, any change to the system will require recollecting necessary information and rebuilding the model or some of its parts. These shortcomings of the manual model building are apparent. Thus the problem of automated analysis of multithreaded programs and building their performance models gained a significant attention in the research community.

Automatic construction of simulation models requires understanding the structure of the program, its semantics, and resource demands. This can be done using the thorough and sophisticated analysis of the program. Below we will review the main directions towards automatic program analysis and their application to the automatic construction of performance models.

Program analysis have been extensively used to understand structure of multithreaded programs. In particular, Reiss proposed CHET - a tool for extracting specifications from the parallel program itself and represnnting them in a form of automata [58]. By further extending work in this area, Reiss produced a system to identify locks (synchronization mechanisms) in a multithreaded program and to determine their types [57]. Author uses program instrumentation to collect information about interactions of threads in a program and then relies on heuristics to assign each lock into a corresponding category, be it a semaphore, a read-write lock, a mutex etc.

Similarly, Burnim and Sen discovers deterministic specification in the multithreaded programs [23]. Authors define pre- and post-conditions for the block of a program's code that must hold for all the inputs in the multithreaded program. Finally, Barham et al develop Magpie - a tool for understanding the characteristics of the system's workload [15]. Magpie relies on the user-supplied schema to infer the flow of request the from the sequence of API calls.

Program analysis has been used extensively to collect detailed information about the performance of the program. Teng et al developed THOR - a tool for performance analysis of parallel Java applications [67]. THOR relies on a sophisticated combination of kernel and user-mode instrumentations in order to to understand and visualize relations between the Java threads and locks.

Coppa, Demetrescu, and Finocchi present the idea of input-sensitive profiling [26]. Their profiler automatically measures how the input size of the program's function affects the running time of that function. Similarly, Zaparanuks and Hauswirth develop a tool that automatically deduce the cost of the algorithm based on the size of the supplied data structures [78]. Authors rely on the combination of static and dynamic analyses to produce a trace of the program, and then approximate a dependency between the input size and the number of iterations by the program. A similar approach was taken by Goldsmith et al to measure the computational complexity of the application [33].

Stack sampling has become a popular technique to reduce the overhead of dynamic analysis. Tallent and Mellor-Crummey use stack sampling to identify parallel idleness and parallel overhead in the multithreaded program [66]. Their work allows to discover areas of the code that contribute to non-linear performance characteristics of the program. Mitchell and Sweeney applied a similar approach towards predicting performance of multithreaded programs [49].

Program analysis techniques similar to ones described above were used to automatically construct performance models. Hrischuk et al conducted an initial study on automatic generation of LQN models from the system's trace [39]. The Although this information allows generating the skeleton of the LQN model automatically, it is not clear if the instrumentation or parameterization of the model is automated as well.

Similarly, Israr et al [41] automatically build LQN models of message-passing programs from their traces. In their work, authors concentrate on semantical correctness of the resulting model. Woodside et al [74] propose building LQN models using information about the system available during its design phase. Authors implement a prototype tool capable of generating the LQN model and extracting its parameters, such as resource demand. Authors verify their approach by building the model of the document distribution service application. The resulting model have accuracy $\varepsilon = 0.3$.

Brosig, Huber, and Kounev [20] automatically generate the Palladio Component Model (PCM) of the distributed EJB application from its traces. Authors simulated dependency between the intensity of the workload and the performance of the SPECj Enterprise2010 Java benchmark. Their predictions of CPU utilization and the response time are accurate within $\varepsilon \in (0.1 \dots 0.3)$.

Nudd et al proposed a PACE framework [55] to automate building performance models of MPI/PVM message-passing programs. The skeletons of the PACE models are built by the means of static code analysis, while model parameters can be specified either manually or by benchmarking.

Xu and Subhlok [77] automatically build models of MPI applications from their traces. Normally accuracy of their models is ($\varepsilon \leq 0.15$). However the accuracy drops to $\varepsilon \in (0.3 \dots 0.5)$ in configurations where nodes of the system are involved in synchronization operations.

Resource demands for the program are usually discovered by instrumenting the program and measuring the resource demands of its individual components [15]. If direct measurement is not possible, the resource demands can be inferred. In particular, Rolia et al rely on a least square approach to infer resource demand from higher-level measurements such as execution count [61]. Similarly, Brosig et al rely on a Service Demand Law [21] to infer resource demand for PCM-based models.

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.

We address this limitation by proposing novel static and dynamic analyses for building performance models of multithreaded programs. Our analyses automatically discover resource demands and semantics of thread interaction and translate this information into a model of the multithreaded system.

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 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 betweeh 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 $(0.032 \dots 0.134)$ for CPU-intense and $(0.262, 0.269)$ for I/O-intense workloads, which is comparable to results reported by other studies [31] [27][40] [72] [76][54]. At the same time, our framework builds program models automatically and does not require running the program in many configurations. Source code of our framework and generated models is available at [11].

Our next steps will be improving the flexibility of our framework, as discussed in Section 6. These improvements 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

1. http://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/overview.html.
2. http://linux.die.net/man/8/btrace.
3. http://www.omnetpp.org/.
4. http://asm.ow2.org/.
5. http://sunflow.sourceforge.net/.
6. http://tomcat.apache.org/.
7. http://tomcat.apache.org/.
8. http://itextpdf.com/.
9. http://www.inet.omnetpp.org/.
10. http://www.nsnam.org/.
11. https://sourceforge.net/projects/persik/.
12. M. Akdere, U. etintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In A. Kementsietsidis and M. A. V. Salles, editors, *ICDE*, pages 390–401. IEEE, 2012.
13. G. M. Amdahl. 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), pages 483–485, New York, NY, USA, 1967. ACM.
14. E. Anderson. Hpl–ssp–2001–4: Simple table-based modeling of storage devices, 2001.
15. P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proc. of the Symposium on Opearting Systems Design & Implementation*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.
16. J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324:446–449, 1986.
17. F. Bause. Queueing petri nets - a formalism for the combined qualitative and quantitative analysis of systems. In *In Proc. of the 5th International Workshop on Petri nets and Performance Models. IEEE*, pages 14–23. IEEE, 1993.
18. S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proc. of the 6th international workshop on Software and performance*, WOSP '07, pages 54–65, New York, NY, USA, 2007. ACM.
19. M. Bennani and D. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *Proc. of International Conference on Automatic Computing*, pages 229–240, Washington, DC, USA, 2005. IEEE.
20. F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *Proc. International Conference on Automated Software Engineering*, ASE '11, pages 183–192, Washington, DC, USA, 2011. IEEE.
21. F. Brosig, S. Kounev, and K. Krogmann. Automated extraction of palladio component models from running enterprise java applications. In *Proc. of the 1st International Workshop on Run-time models for Self-managing Systems and Applications*, ROSSA'09. ACM, New York, NY, USA, Oct. 2009.
22. J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance java. In *Proc. OF ACM Java Grande Conference*, pages 81–88. ACM, 1999.
23. J. Burnim and K. Sen. Determin: Inferring likely deterministic specifications of multithreaded programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 415–424, New York, NY, USA, 2010. ACM.
24. L. Cheung, L. Golubchik, and F. Sha. A study of web services performance prediction: A client's perspective. In *Proc. of the 19th Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 75–84, Washington, DC, USA, 2011. IEEE.
25. B.-G. Chun, L. Huang, S. Lee, P. Maniatis, and M. Naik. Mantis: Predicting system performance through program analysis and modeling. *CoRR*, abs/1010.0019, 2010.
26. E. Coppa, C. Demetrescu, and I. Finocchi. Input-sensitive profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'12, pages 89–98, New York, NY, USA, 2012. ACM.

27. J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *Proc. of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 337–348, New York, NY, USA, 2011. ACM.
28. W. Feng and Y. Zhang. A birth-death model for web cache systems: Numerical solutions and simulation. In *Proc. of International Conference on Hybrid Systems and Applications*, pages 272–284, 2008.
29. D. Ferrari, G. Serazzi, and A. Zeigner. *Measurement and tuning of computer systems*. Prentice-Hall, 1983.
30. G. Franks and M. Woodside. Performance of multi-level client-server systems with parallel service operations. In *Proc. of the 1st International Workshop on Software and Performance*, WOSP '98, pages 120–130, New York, NY, USA, 1998. ACM.
31. A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Proc. of International Conference on Data Engineering Workshops*, pages 87–92, 2010.
32. A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proc. of International Conference on Data Engineering*, pages 592–603, Washington, DC, USA, 2009. IEEE.
33. S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson. 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, pages 395–404, New York, NY, USA, 2007. ACM.
34. N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan. Anatomy: An analytical model of memory system performance. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 505–517, New York, NY, USA, 2014. ACM.
35. J. Happe, D. Westermann, K. Sachs, and L. Kapov. Statistical inference of software performance models for parametric performance completions. In *QoSA'10*, pages 20–35, 2010.
36. H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.
37. C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.
38. S. V. Hoecke, T. Verdickt, B. Dhoedt, F. Gielen, and P. Demeester. Modelling the performance of the web service platform using layered queuing networks. In *Proc. of International Conference on Software Engineering Research and Practice*, 2005.
39. C. E. Hrischuk, J. A. Rolia, and C. M. Woodside. Automatic generation of a software performance model using an object-oriented prototype. In *Proc. of the 3rd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '95, pages 399–409, Washington, DC, USA, 1995. IEEE.
40. H. Huang, S. Li, A. Szalay, and A. Terzis. Performance modeling and analysis of flash-based storage devices. In *Symposium on Mass Storage Systems and Technologies (MSST),*, pages 1 –11, may 2011.
41. T. A. Israr, D. H. Lau, G. Franks, and M. Woodside. Automatic generation of layered queuing software performance models from commonly available traces. In *Proc. of International Workshop on Software and Performance*, WOSP '05, pages 147–158, New York, NY, USA, 2005. ACM.
42. S. A. Jarvis, B. P. Foley, P. J. Isitt, D. P. Spooner, D. Rueckert, and G. R. Nudd. Performance prediction for a code with data-dependent runtimes. *Concurr. Comput. : Pract. Exper.*, 20:195–206, March 2008.
43. S. Kounev, S. Spinner, and P. Meier. Introduction to queueing petri nets: modeling formalism, tool support and case studies. In *Proc. of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE '12, pages 9–18, New York, NY, USA, 2012. ACM.
44. L. M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.
45. A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Higher Education, 2nd edition, 1997.

46. E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance, Computer System Analysis Using Queuing Network Models*. Prentice Hall, 1984.
47. B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proc. of SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM.
48. D. A. Menasce and M. N. Bennani. Analytic performance models for single class and multiple class multithreaded software servers. In *Int. CMG Conference*, 2006.
49. N. Mitchell and P. F. Sweeney. On-the-fly capacity planning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 849–866, New York, NY, USA, 2013. ACM.
50. T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proc. of the 4th international conference on Computing frontiers*, CF '07, pages 143–152, New York, NY, USA, 2007. ACM.
51. D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting dbms. In *Proc. of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 239–248, Washington, DC, USA, 2005. IEEE.
52. N. Nethercote, R. Walsh, and J. Fitzhardinge. Building workload characterization tools with valgrind. Invited tutorial, October 2006.
53. H. Q. Nguyen and A. Apon. Hierarchical performance measurement and modeling of the linux file system. In *Proc. of International Conference on Performance Engineering*, ICPE '11, pages 73–84, New York, NY, USA, 2011. ACM.
54. H. Q. Nguyen and A. Apon. Parallel file system measurement and modeling using colored petri nets. In *Proc. of International Conference on Performance Engineering*, ICPE '12, pages 229–240, New York, NY, USA, 2012. ACM.
55. G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace–a toolset for the performance prediction of parallel and distributed systems. *Int. J. High Perform. Comput. Appl.*, 14:228–251, August 2000.
56. T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
57. S. Reiss and A. Tarvo. Automatic categorization and visualization of lock behavior. In *Proc. of the first IEEE Working Conference on Software Visualization*, VISSOFT '13. IEEE, 2013.
58. S. P. Reiss. 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.
59. S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. of the 23rd International Conference on Software Engineering*, ICSE '01, pages 221–230, Washington, DC, USA, 2001. IEEE.
60. J. Rolia, G. Casale, D. Krishnamurthy, S. Dawson, and S. Kraft. Predictive modelling of sap erp applications: Challenges and solutions. In *Proc. of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '09, pages 9:1–9:9, ICST, Brussels, Belgium, Belgium, 2009. ICST.
61. J. Rolia, A. Kalbasi, D. Krishnamurthy, and S. Dawson. Resource demand modeling for multi-tier services. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, pages 207–216, New York, NY, USA, 2010. ACM.
62. J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Softw. Eng.*, 21(8):689–700, Aug. 1995.
63. N. Roy, A. Dabholkar, N. Hamm, L. W. Dowdy, and D. C. Schmidt. Modeling software contention using colored petri nets. In E. L. Miller and C. L. Williamson, editors, *MASCOTS*, pages 317–324. IEEE, 2008.
64. J. Y. Stein. *Digital Signal Processing: A Computer Science Perspective*. John Wiley & Sons, Inc., New York, NY, USA, 2000.
65. R. Strebelow, M. Tribastone, and C. Prehofer. Performance modeling of design patterns for distributed computation. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 251–258, Washington, DC, USA, 2012. IEEE.

66. N. R. Tallent and J. M. Mellor-Crummey. 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, pages 229–240, New York, NY, USA, 2009. ACM.
67. Q. M. Teng, H. C. Wang, Z. Xiao, P. F. Sweeney, and E. Duesterwald. Thor: A performance analysis tool for java applications running on multicore systems. *IBM J. Res. Dev.*, 54(5):456–472, Sept. 2010.
68. D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora. A framework for measurement based performance modeling. In *Proc. of International Workshop on Software and Performance*, WOSP '08, pages 55–66, New York, NY, USA, 2008. ACM.
69. E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. In *Proc. of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, SIGMETRICS '10, pages 1–12, New York, NY, USA, 2010. ACM.
70. E. Thereska and G. R. Ganger. Ironmodel: robust performance models in the wild. In *Pro. of International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, pages 253–264, New York, NY, USA, 2008. ACM.
71. R. van der Mei, R. Hariharan, and P. Reeser. Web server performance modeling. *Telecommunication Systems*, 16:361–378, 2001. 10.1023/A:1016667027983.
72. M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with cart models. In *Proc. of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '04, pages 588–595, Washington, DC, USA, 2004. IEEE.
73. D. Westermann, J. Happe, R. Krebs, and R. Farahbod. Automated inference of goal-oriented performance prediction functions. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 190–199, New York, NY, USA, 2012. ACM.
74. C. M. Woodside, C. E. Hrischuk, B. Selic, and S. Bayarov. Automated performance modeling of software generated by a design environment. *Perform. Eval.*, 45(2-3):107–123, 2001.
75. C. M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Trans. Comput.*, 44(1):20–34, Jan. 1995.
76. J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy. Performance modeling and prediction of enterprise javabeans with layered queuing network templates. In *Proc. of Conference on Specification and Verification of Component-based Systems*, SAVCBS '05, New York, NY, USA, 2005. ACM.
77. Q. Xu and J. Subhlok. Construction and evaluation of coordinated performance skeletons. In *Proc. of International Conference on High performance computing*, HiPC'08, pages 73–86, Berlin, Heidelberg, 2008. Springer-Verlag.
78. D. Zaparanuks and M. Hauswirth. Algorithmic profiling. *SIGPLAN Not.*, 47(6):67–76, June 2012.