Abstract of "Automated Performance Modeling of Multithreaded Programs" by Alexander Tarvo, Ph.D., Brown University, May 2015.

The performance of multithreaded programs is often difficult to understand and predict. Multiple threads use various locking operations, resulting in the parallel execution of some computations and the sequential execution of others. Threads use hardware resources such as a CPU or a hard drive simultaneously, which may lead to their saturation. The result is a complex non-linear dependency between the configuration of a multithreaded 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, 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 work we present an approach for building performance models of multithreaded programs automatically. We employ hierarchical discrete-event models. The higher-level model simulates the data flow through the program using the queueing network. The mid-level model simulates program's threads using probabilistic call graphs. The low-level model simulates program-wide locks and underlying hardware.

We extract information necessary for constructing the model using a combination of static and dynamic analyses of the program under study. 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. The discovered information is translated into the discrete-event model of the program.

In our experiments we successfully generated performance models of a suite of large multithreaded programs. The resulting models predicted performance of these programs across a range of configurations with a reasonable degree of accuracy.

Automated Performance Modeling of Multithreaded Programs

by
Alexander Tarvo
B.S., Chernigov State Technological University, 2002
Sc. M., Chernigov State Technological University, 2004
Sc. M., Brown University, 2011

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2015

This dissertation by Alexander Tarvo is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date _____ _____

Steven Reiss, Director

Recommended to the Graduate Council

Date _____ _____

Ugur Cetintemel, Reader
Brown University

Date _____ _____

Rodrigo Fonseca, Reader
Brown University

Date _____ _____

Eno Thereska, Reader
Microsoft Research UK

Approved by the Graduate Council

Date _____ _____

Peter M. Weber
Dean of the Graduate School

Acknowledgements

First I would like to thank Steven Reiss for being my advisor. He gave me lots of freedom in pursuing my dream of becoming a researcher. At the same time, he was always open to help me during this long journey. His mentorship ranged from giving me invaluable advices on research and teaching to fixing grammar errors in my writing. During the most troublesome times I could always find reassurance and support from Dr. Reiss. I feel lucky to have him as my advisor.

I would also like to thank Ugur Cetintemel for his support and mentorship over these years. Our conversations not only served as an important part of my education here, but always invigorated me to continue my research.

Eno Thereska and his publications on performance models were a source of inspiration for me. Dr. Teherska taught me how to convert my work into accessible papers and how to survive the job search.

I would thank to Rodrigo Fonseca for his feedback on my work and for his suggestions how to make it better.

Shriram Krishnamurthi helped me to avoid many pitfalls on my way to the Ph.D. degree. His involvement in the key moments of my study proved to be essential for completing this journey.

I wish to thank the technical and administrative staffs of the Brown Computer Science department. I am particularly thankful to Lauren Clarke who patiently answered all my questions about the graduate process. I also thank to Eugenia G. deGouveia, Jane McIlmail, Kathy Kirman, Dawn Reed, Phirum Peng, Dorinda Moulton, and Max L. Salvas.

I thank to my professors at the Chernigov State Technological University, where I made my first steps in research and teaching. This experience eventually motivated me to enter the Ph.D. program.

I would never become a Ph.D. student at Brown without help and support from my colleagues at Microsoft. I thank my former managers, Koushik Rajaram and Jacek Czerwonka, for giving me the opportunity to align my research activities with my job. And I thank to researchers at Microsoft Research, who served as role models for me. Nachiappan Nagappan become my first pilot in the world of research. Later on his guidance and advice made my internship at MSR both rewarding and pleasant. Thomas Zimmermann, my co-author, was always a supportive mentor for me. I also thank Brendan Murphy, Christian Bird, Sriram Rajamani, and Aditya Nori for their help and support.

I owe a tremendous debt to my family. My mother, Inna Tarvo, become the first and the most important teacher in my life. She introduced me to the wonderful world of engineering and computers

and gave me the first lessons in programming. All the success I had in my life I owe to her.

I am thankful to my wife, Tanya, for all her patience and support during my Ph.D. study. Like the decemberist's wife, she shared this long and often harsh road with me. And never denied me of her love and support.

I thank my son, Anton, for being patient with his dad, who spent too much of his time in a front of a computer screen. Considering Anton'sinterest in astronomy and cosmology, I think he will become a much better researcher than me.

I have been lucky to meet lots of good friends at Brown. This experience would never be the same without Alexander Kalinin, Georgy Megrelishvili, Marek Vondrak, Marcelo Martins, Hammurabi Mendes, Alexandra Papoutsaki, Irina Calciu, Dalia Kaulakiene, and other wonderful people whom I met here.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis presents our approach towards building performance models of large multithreaded programs. Our models accurately predict performance of the multithreaded system across a range of configurations, which include variations in the intensity of the workload, configuration options of the program, and some characteristics of the hardware. Our models are built automatically by analyzing the code of the multithreaded program and its behavior in a single configuration. Such abilities of performance models are important for many applications, including autonomous deployment of software systems, answering "what-if" questions about their performance, and detecting performance anomalies.

The following thesis statement describes the ultimate objective of the presented dissertation:

> An accurate and useful performance prediction model of a complex multithreaded program can be built automatically using data collected from executing that program in a single representative configuration.

1.1 Motivation

Multithreaded programs utilize resources of modern hardware more efficiently. However, behavior of multithreaded programs is significantly more complex than behavior of single-threaded applications.

Multiple threads use shared hardware resources, such as the CPU, disks, and the network simultaneously. The behavior and performance characteristics of such system can differ significantly from the behavior of a single-threaded application. Moreover, threads rely on synchronization to ensure semantic correctness of computations (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 some parts of the program's code and the sequential execution of others.

As a result, multithreaded programs exhibit complex non-linear dependency between their configuration and performance. Configurations can reflect the following characteristics of the program

and its environment:

- Parameters of the workload, such as the number of HTTP requests the web server receives in a second;
- The options of the program itself, such as the number of working threads or the size of the internal cache for input-output (I/O) operations;
- Parameters of the Operating System (OS), such as the maximum number of file descriptors available to the program;
- Characteristics of the hardware, such as the number of available CPUs, the amount of random access memory, or the seek time of the hard drive.

Even an expert may be unable to understand the dependency between the configuration and performance of a multithreaded program on a quantitative level. For an example, consider a scientific computing application, where some computations are parallelized across multiple threads, while remaining computations are performed sequentially. A system administrator is trying to achieve a twofold improvement in performance of this application by purchasing a CPU with a larger number of cores. To achieve this goal he must know the exact number of CPU cores necessary to achieve a desired performance.

In this example the system administrator is trying to answer a "what-if" question: what will be the performance of the program for a given number of CPU cores? Answering this question can be difficult. As the program performs a mix of sequential and parallel computations, increasing the number of CPU cores will not result in the proportional increase in performance.

A simple approach to understand dependencies between the configuration and the performance of the program is to run that program in different configurations and measure the performance in each of these configurations. However, such experimentation may require specialized hardware and may not be feasible in practice. Moreover, as the number of different parameters increases, the number of possible configurations grows exponentially. Considering the additional time required for setting up the system for every such experiment, running the system in so many configurations may take too much time and effort to become practical.

A better way to understand dependencies between the configuration and the performance is building a *performance prediction model*. Such model will predict dependency between configuration of the system and its performance without reverting to the costly and time-consuming runs of the actual program.

Performance models are essential for a variety of applications. First, performance prediction model are used for answering "what-if" questions [70] similar to one described above. One variation of "what-if" question is "what will be the performance of the system for a given combination of configuration parameters?" A reverse approach is also true. In particular, answering questions like "what combination of configuration parameters will produce the desired performance?" is essential for efficient deployment of an application.

Second, performance models are employed as a central component for autonomic data centers [27]. Here prediction results are used to dynamically configure the system to achieve high performance. An example of such usage is finding a good configuration for deploying the Tomcat web server. The model will predict response time, throughput and resource utilization by the Tomcat for each combination of configuration parameters, including the number of available CPU cores, the number of Tomcat working threads, and the rate of incoming connections. The configuration that utilizes hardware resources efficiently and also satisfies the service agreement should be chosen for deployment.

A third application of performance models is scheduling the execution of multiple tasks on a cluster [74]. Performance prediction models are used to predict the running time of each task depending on the amount of computations performed, on the configuration options of the program, and on the parameters of the hardware. By leveraging on these predictions the optimization algorithm schedules execution of tasks on different nodes of a cluster. Usually the optimization criteria involves minimizing the overall running time of the tasks while sustaining a desired utilization of hardware resources [58].

Finally, the performance prediction models are used to detect performance anomalies in the running software system [94]. The model is deployed alongside the actual system. The performance predicted by the model is compared to the performance measures obtained from the system as it runs. Significant and systematic deviations of measured performance from the predicted performance are considered as manifestations of the system's abnormal behavior. The model can also facilitate localizing the source of abnormal behavior, as long as it is capable of predicting performance of individual components of the system.

Building accurate and useful performance models of modern industrial programs is a challenging problem. These programs are large and complex, and are updated regularly. Building performance models of multithreaded programs is even more difficult. Such performance models must correctly represent the complex locking behavior of the application and concurrent usage of various computational resources such as the CPU and the I/O subsystem.

The section 2 of this thesis provides a detailed overview of existing approaches to the performance modeling. Two techniques traditionally used for building performance models of complex computer programs are statistical modeling and simulation. However, existing applications of these techniques to building performance models of complex multithreaded programs have demonstrated severe limitations.

Statistical models approximate the dependency between the configuration and the performance using a machine learning method, such as a CART tree, a neural network, or a non-linear regression. Statistical models do not need a good knowledge of the system's structure and semantics. However, training these models requires measuring the performance of the system in many different configurations. Moreover, any change to the system may require recollecting this information. As a result, unless the training data is already available or can be easily obtained, statistical models may not be a practical approach to performance prediction.

A viable alternative to statistical method are simulation models. The structure and/or behavior of the such model follows the structure and behavior of the system.

Methodologies traditionally used to build simulation models are queuing networks and Petri nets. Their extensions developed specifically for performance modeling include Layered Queuing Networks (LQN), colored Petri nets, Palladio Component Models (PCM), and others. Although in certain scenarios these models can be solved analytically, in a general case the prediction is obtained using simulation.

Building a simulation model normally does not require running the system in many configurations, but requires deep knowledge of the system's structure and functionality. One approach to obtain the necessary information is manual analysis of the program. However, building performance models manually by using information obtained from developers or by analyzing the code of the program is costly and error-prone. Moreover, the model needs to be updated every time when the program changes. Thus in order to be practical simulation models must be built automatically.

A significant effort has been devoted to automatic building of simulation performance models. Most of work in this area was dedicated to building models of distributed and message-passing systems. Constructing models of such systems is relatively straightforward. Individual computers are naturally considered as components of the model, and network links become the connections between these components. Message passing systems either use a specific API to communicate between hosts (e.g. MPI) or rely on a specific network protocol for this purpose. As a result, interaction between different computers in a message-passing system can be tracked relatively easily.

Unfortunately, methodologies used to automatically build performance models of message passing systems do not capture semantics of complex thread interaction and may not accurately simulate resource contention in multithreaded systems. As a result, these methodologies demonstrate inferior accuracy when used to build models of complex multithreaded programs. For example, the framework described in [104] predicts performance of single-threaded MPI applications with relative error $\varepsilon \in (0.02 \dots 0.3)$. However, its performance decreases to $\varepsilon = 0.5$ if the program performs complex synchronization operations.

Our work attempts to overcome these limitations. **We develop an innovative technique that allows to automatically build the performance model of the multithreaded program by running it in a single representative configuration**. The resulting model can predict performance of the program running in various configurations under the established workload. Our models also predicts utilization of computational resources, which facilitates performance analysis of the system and bottleneck detection.

1.2 Goals

The goal of our work is to develop an approach and a framework for automatic building performance prediction models of real-world multithreaded programs. This includes programs such as servers, multimedia programs such as encoders and 3D renderers, and scientific computing applications. 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. At the same time, we allow the analyst to inspect the generated model, specify the range of configurations in which performance should be predicted, and performance metrics that should be collected.

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. Prediction accuracy is the main quality we seek in our models. For each given configuration the difference between the performance predicted by the model and the actual performance of the system should be minimal.

Generating models by 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 costly, and may not be feasible in a production environment. Instead, we want to generate the model by running a program in a small number of configurations; ideally – in a single representative configuration. In this configuration the behavior of the program and its resource requirements should be very similar to a wider range of configurations.

Predicting performance of programs on the commodity hardware

We build models of programs running on commodity hardware. Potentially our approach can be extended to predicting performance of programs running on cluster and grid systems. However, this would require developing an additional set of hardware models and potentially different approach for program analysis, which is beyond the scope of this work.

Predicting aggregate values of the performance metrics

We do not predict individual measurements of the program's performance metrics. Instead, we predict values of these performance metrics aggregated through many measurements. For example, we do not predict the amount of time required for the web server to process a particular HTTP request. Instead, we predict the mean values of response time and throughput for that web server measured by serving a number of HTTP requests.

1.3 Proposed Approach

Our approach consists of two parts: models specially designed for performance modeling of multithreaded programs and the methodology for automated building of these models. A combination of these two aspects make our approach unique.

We have developed a simulation-based approach to predict performance of multithreaded programs. The choice of simulation over the statistical modeling was dictated by our desire to build the model from a single representative configuration of the program. Components of our model directly correspond to the components of the multithreaded system, which include software, hardware, and

elements of the workload.

We identify and model different sources of delay in a multithreaded program. Similar to existing performance models we simulate delays caused by computations, I/O operations, and queuing. But in contrast to previous methods we also model delays caused by *resource contention*.

Resource contention is a conflict between the program's threads over access to a shared resource, such as CPU, memory, disk, or a synchronization construct [1]. Resource contention is a strong factor affecting performance of the multithreaded program. For example, if the number of 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, the average time the thread must wait to acquire a lock depends on the total number of threads competing for that lock. As the number of working threads increases, the amount of time to finish certain computations will remain the same, while timing of another operations will increase because of resource contention. As a result, *resource contention becomes one of the primary factors contributing to the non-linear dependency between the configuration and performance of the program.*

Accurate prediction of software performance requires modeling different factors that cause delays in the program, including queuing and resource contention. To accurately simulate these factors we developed a hierarchical model consisting of three tiers (see Section 3). At the high level we model the system using a queuing model. Requests in the queuing system correspond to *tasks* - discrete units of work performed by the program's threads. Queues correspond to the buffers and queues in the program and to some OS queues, such as the network connection queue. Service nodes correspond to the program's threads; they simulate delays introduced by threads as they process tasks. The exact amount of time required by the thread to process the task is simulated by the mid-tier model.

The mid-tier model represents threads as probabilistic call graphs. Nodes of such graph correspond to the pieces of the thread's code - code fragments. Edges represent possible transitions of control flow between code fragments. Code fragments (CFs) simulate different operations performed by the program: CPU-bound computations, disk I/O operations, and synchronization. Execution of each CF results in some delay τ. The amount of delay τ depend on the demand for a particular resource accessed by the CF, and on contention of that resource. The exact delay τ is computed by the low-level model.

The low-level model simulates contention of shared resources. It includes models of locks present in the program, models of the CPU and the thread scheduler, and the model of the disk I/O subsystem.

The hierarchy of models allows to accurately simulate different factors affecting the program's performance. The high-level model simulates delays associated with storing the tasks in the queues of the program. The mid-level model simulates delays that occur when threads process individual tasks. The low-level model simulates delays caused by contention of locks and computational resources.

To implement models of the multithreaded programs we have developed a PERSIK (PERformance SImulation Kit) framework (see Section 4). PERSIK can accurately simulate performance

of applications written using different languages and frameworks for multithreaded programming. However, building PERSIK models requires extensive information about the program: threads and thread groups present in the program; locks and queues; semantics of synchronization operations; code fragments, their types and semantics. Obtaining this information manually proved to be overly time-consuming and error prone even for small applications. Manually building models of large applications is not feasible at all.

To make our models practical we developed an methodology for automatic building of performance models (see Section 5). Our methodology involves a combination of static and dynamic analyses to obtain all the necessary information about the program under study. We implemented our methodology into a framework for automatic modeling of multithreaded Java programs.

Our analysis consists of four stages. First, our framework executes the program in a single representative configuration and collects samples of its call stack. These samples are used to identify logical groupings of threads (thread pools) in the program, to discover libraries in the program, and to obtain a set of frequently executed functions and methods that become starting points of the upcoming static analysis.

Second, the static analysis of the program is performed. The framework scans the code of the program. It discovers I/O and synchronization operations in the program, which become code fragments of our thread model. It also discovers initializers of the locks and queues. These are not explicitly represented in the model, but are used during the dynamic analysis of the program.

Third, the dynamic analysis of the program is performed. The framework inserts probes at the boundaries of the statements that initialize program's locks, perform I/O and synchronization operations. Then the framework executes the program in the same configuration as during the initial stack sampling run. The trace generated by the instrumented program is used to discover the rest of information necessary for building the model. This includes CPU computations performed by the program, the structure of call graph for threads, as well as demands for computational resources by the code fragments. In addition, the framework discovers characteristics of the program's locks and queues and semantics of their interaction with the program's threads.

Finally, our framework translates information obtained during the previous steps into the PERSIK model of the multithreaded program. Particular attention is devoted to generating a simple and compact model. This is necessary to ensure that the resulting model is easy to understand and has high performance.

1.4 Challenges in Program Analysis and Performance Modeling

Considering the high complexity of multithreaded programs and variety of different frameworks used to implement threading and synchronization, constructing performance models remains a challenging problem. The primary challenges are:

Discovering the semantics of thread interaction. Building the performance model requires

knowledge of the queues and buffers in the program and their interaction with the program's threads, such as which thread reads or writes to a particular queue. It also requires knowledge of locks present in the program and their types, namely, if this particular lock is a semaphore, a mutex, or a barrier. This also includes knowledge of interaction between locks and the program's threads, namely, which thread accesses a particular lock.

However, there are numerous ways to implement locks and queues in the program. Complex locks like semaphores or barriers may be implemented using a combination of a low-level synchronization constructs, such as Java monitors and synchronized regions. Alternatively, the program may use a standard library that provides implementations of such high-level locks, such as `java.util.concurrent` in Java or System.Threading in C#.

Similarly, queues and buffers used to exchange data between the program's threads can be implemented in a variety of ways. These may be regular queues or arrays (potentially, guarded by locks), or blocking queues implemented using the low-level constructs, or specialized implementations of queues for thread interaction. Finally, queues can be encapsulated in a class that implements even a higher-level threading patterns, such as an executor or a thread pool.

Considering a variety of approaches for implementing locks and queues in the program, automatically discovering semantics of thread interaction requires complex program analysis.

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. However, the behavior of modern operating systems and hardware can be increasingly complex. For example, after completing a disk I/O operation the modern I/O scheduler may wait for another I/O operation issued by the same thread. Such implementation improves both throughput and response time of the disk I/O subsystem in case of multiple sequential disk accesses, but results in more complex behavior of the thread scheduler. Similarly, the modern thread scheduler may bind a thread to a particular CPU in a hope of improving a cache hit rate, and thus improve overall performance of the system.

Considering such complex behavior of the OS and hardware, building accurate and compact models of OS, hardware and locks remains a challenging problem.

Understanding the structure of the program. Accurate modeling of resource contention requires knowledge of the internal structure of the program's threads. This includes information on operations carried out by the threads, be they CPU computations, I/O activities, or locking; and the sequence in which these operations are performed.

Different hardware allows different degree of parallelism. Modern CPUs have multiple cores and can run simultaneously multiple threads without causing resource contention. At the same time, the hard drive usually carries out a single I/O operation at a time. Thus the degree of contention experienced by the program's threads strongly depends on type of hardware resources they consume and thus on the types of operations they perform. As a result, knowledge of which operations are performed by the program's threads (CPU computations, I/O activities, or locking) is essential to

model resource contention accurately.

The sequence of operations is also important, especially for programs that are engaged in synchronization. For example, consider a group of threads that acquire a mutex and also perform the CPU-bound computations. If the mutex is released before the computations begin, the performance of the program will be determined by the amount of CPU contention introduced by the computations. However, if the mutex is released only after the computations are complete, the performance will be determined by the contention of the mutex. As a result, knowledge of the sequence in which thread performs these operations is essential for the accurate performance modeling. This knowledge becomes even more important for threads that use different types of hardware resources and are engaged in complex synchronization operations.

Detecting different operations performed by the program, determining their types, and understanding their sequence of execution requires sophisticated static and dynamic analysis of the program.

Discovering parameters of the program's components. Performance of the program depends on parameters of its components, such as locks and queues, and on the resource demands of its threads.

For example, the amount of time the thread has to wait to for a semaphore depends on a number of permits available for that semaphore. Similarly, the amount of time the program spends on the disk I/O depends on the amount of data it has to transfer. And the amount of time required to perform computations by the method depends on the number of operations performed by that thread.

Discovering program's parameters require further analysis of the program. Retrieving parameters of locks and queues may require additional semantical analysis of these constructs. Obtaining demands for computational and I/O resources may require sophisticated program instrumentation. Furthermore, certain information, such as the amount of data transferred from the hard drive might be reliably obtained only from the OS kernel.

1.4.1 Addressing Challenges

We address the challenges described above by developing a sophisticated methodology for program analysis.

Understanding semantics of thread interaction is a very hard problem. A general-purpose solution to this problem would require a very sophisticated and potentially inaccurate program analysis. However, we discovered that such analysis can be greatly simplified if the program relies on standard implementation of locks and queues. By leveraging on this knowledge it is possible to accurately detect locks and queues in the program, determine their types, and detect operations on these locks and queues performed by the program's threads.

To accurately simulate resource contention we developed elaborate models of the system's shared resources. This includes models of locks present in the program, models of elements of the operating system, such as the thread scheduler, the I/O scheduler and the page cache, and models of hardware,

such as the CPU and the hard drive. Our models are relatively simple and do not require information specific to the particular execution of the program, such as the numbers of hard drive sectors being accessed. Nevertheless, they achieve prediction accuracy comparable to other studies.

We identify computations performed by the program's threads using a combination of static and dynamic analyses. Static analysis identifies synchronization, I/O operations, and accesses to the shared queues by the program. The rest of the code is considered as CPU-bound computations and is detected during the dynamic analysis. Dynamic analysis also determines sequence of these operations and represents them as a probabilistic call graph.

We developed different approaches for retrieving parameters of different computations in the program. Parameters of the synchronization and queuing operations are discovered by instrumenting calls to corresponding locks and queues. Resource demands for CPU computations are discovered from timestamps in the program trace. Resource demands for I/O operations are obtained by collecting kernel-mode trace of low-level I/O requests and associating them with user-mode I/O operations in the program using signal processing technique.

1.5 Contributions

In this work we have developed a methodology for building accurate performance models of multithreaded programs. Our models can be built automatically by running the program in a single representative configuration and can predict performance across a range of configurations. We summarize our contributions below:

- **A three-tier discrete-event model for performance prediction.** We have introduced a model specially designed for predicting performance of multithreaded programs. Our model simulates performance the system at multiple levels. At the high level it simulates flow of data through the threads and queues of the program. At the middle level the model simulates delays that occur in the program's threads. At the low level it simulates contention of locks and hardware resources. Such architecture allows accurate simulation of all the factors that can influence performance of a multithreaded program. The detailed description of our models is presented in the Section 3 of this thesis.

- **A framework for simulating performance of multithreaded programs.** We developed a framework for constructing performance models of multithreaded programs. Our framework is implemented as a collection of components that can be assembled into the model of multithreaded system. This includes components that represent queuing, synchronization operations, computations, and I/O activities in the program; and components that simulate behavior of operating system and underlying hardware. The framework also defines protocols for implementing interaction between these components. The framework is described in the Section 4.

- **A methodology for automatic construction of performance models.** We propose

a combination of a static and dynamic analyses to understand semantics of multithreaded programs that use standard patterns for threading and synchronization. Our analyses identify thread and thread groups in the program; locks, queues, and their parameters; semantics of interaction between the program's threads; computation performed by the program's threads, their sequence of execution, and demand for hardware resources. Results of the analysis are automatically translated into performance prediction models of multithreaded programs. our methodology for automated model construction is presented in the Section 5.

- **Verification of our approach by building models of various multithreaded programs.** We implemented our methodology as a tool for automatic building performance models of Java programs. We validated our approach by building models of various open source programs, including web servers, multimedia programs and scientific applications. Our models successfully predicted performance of these programs and utilization of hardware resources. The relative error of our models is (0.062 ... 0.134) for predicting performance of CPU-bound programs and (0.027 ... 0.269) for I/O-bound programs, which is comparable to the accuracy reported by other models [44],[64],[103],[73]. A detailed description of our experiments is presented in the Section 6.

Chapter 2

Related work

In this section we review the existing body of work on performance modeling and on automatic program analysis. First we present three basic paradigms for predicting performance of computer systems: analytical, statistical, and simulation models. Next, we review the existing approaches toward automatic building of performance models. Since automatic building of performance models requires an elaborate analysis of the program, we also describe the state of the art in the areas of analysis and understanding of computer programs.

2.1 Performance models of computer programs

At the high level the performance of the system can be represented as a function $y = f(x)$, where 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 $y = f(x)$: analytical models, statistical models, and simulation.

2.1.1 Analytical models

Analytical models are, probrably, the historically first class of models used for performance prediction. These models explicitly represent the dependency $y = f(x)$ as an equation or a set of equations.

Analytical models were used by Narayanan, Thereska and Ailamaki to predict performance of the DBMS [70]. Authors instrument the program and collect data on resource requirements from the resulting trace. Analytical model use this data to predict dependency between the size of the DBMS cache and performance of the system. The model predicts throughput with the relative error $\varepsilon \leq 0.1$ and response time with $\varepsilon \in (0.33, 0.68)$.

Herodotou and Babu developed a set of analytical performance models to predict the running time of MapReduce tasks [52]. 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. It is used by the cost-based optimizer to find those configurations of the task that yield high performance. Using such model-based optimizer increases performance of MapReduce jobs by 50% compared to the default rule-based optimizer implemented in Hadoop.

Similarly, Bennani and Menasce [27] 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 comprising of n servers processing different types of transactions.

Chen, John and Kaseridis used an analytical model to predict the demand on computation resources in the multiprocessor system [33]. In particular, they predicted utilization of the L2 cache, memory bandwidth, and hardware branch predictor for a given program. Their model report average error in a range of 0.09 to 0.13.

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. Furthermore, complex behavior is difficult to express with the analytical model. For example, analytically predicting performance of a single thread pool, which does not perform any synchronization or simultaneous usage of hardware resources, requires development of a complex mathematical model that must be solved iteratively [67].

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 [94] uses analytical models as a part of the larger model to simulate individual components of the distributed system, such as network and disk.

2.1.2 Statistical models

Statistical models tend to overcome some drawbacks of analytical models. They do not explicitly formulate the function $y = f(x)$. Instead, they approximate it using some statistical method. Generally this requires executing the program in a number of configurations in order to collect a representative set of points (X, Y). The dataset (X, Y) is used to train the model.

Statistical models are widely used to predict the performance of SQL databases. Ganapathi et al used a statistical model to predict the running time of SQL queries [45]. Authors construct the x vector from the query plan generated by the DBMS and select its relevant features using Kernel Canonical Correlation Analysis. During the prediction they use a k-NN technique to detect queries with performance features similar to a given one. The correlation between the actual and predicted execution times $R^2 \in (0.55 \ldots 0.95)$.

Authors further extend this technique to predict the running time of Hadoop tasks [44]. The x vector included metrics such as the number of bytes written during different phases of the task. This study reports correlation $R^2 \in (0.87 \ldots 0.93)$

Akdere et al used a similar approach to predict performance of the SQL queries running in

isolation [19]. The x vector can be built from the query plan for the complete SQL query or its individual operators. This modularity allows on-line training of the model. 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 [40]. They use the I/O latency as an indicator of query performance, and predict it 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 developed a statistical model for predicting performance and resource utilization in the message-passing middleware software [49]. They used non-linear regression to predict the relation between the size of the message and performance metrics, which include the response time, utilization of the CPU and the disk subsystem.

Lee et al used linear regression and neural networks to predict the running time of scientific computing applications on a large grid system [64]. Their feature vector x included both parameters of the task and the configuration of the program. The resulting error varied in $\varepsilon \in (0.02 \dots 0.12)$.

Although statistical models do not require knowledge of system's internals, they have severe limitations. Building statistical models require running the system in a large number of configurations, which is time-consuming and costly. Also such experimentation might decrease the availability of the system. Thus it may not be performed in the production environment, when the software is already deployed and is being used by the customers. Moreover, any change to the software or hardware of the system requires re-training the whole model [92].

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 [36].

There are attempts to overcome these shortcomings, or, at least, make them less restrictive. In particular, statistical models can be built when large amounts of data are already available. Thereska et al propose retrieving performance data from a large user base [93]. The released version of the application is instrumented, so it will report its configuration and performance to the centralized database. The analyst can use a k-NN method to search through the database and locate the configuration closest to the given one. This approach can handle complex applications, but its success depends on the popularity of the application and wiliness of users to share their performance data.

Sophisticated program analysis and machine learning techniques can help reducing the amount of training data. In particular, Chun et al. [37] use internal program features such as values of variables, loop and branch counts as metrics x. Authors select relevant features and use them to buid performance models of CPU-bound programs using the non-linear regression. During the prediction phase features can be calculated from the program inputs using program splices. Authors claim

that even when being trained using 10% of the dataset, their model is significantly more accurate ($\varepsilon = 0.06$) than the traditional black-box model trained using 20% of the dataset ($\varepsilon = 0.35$).

Similarly, Westermann et al developed a methodology for iterative selection of points into the training set [99]. 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 and the type of machine learning method employed.

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. In particular, statistical models were widely used to predict performance of disk I/O subsystem. Wang et al [97] developed a model of the hard drive using the CART tree. The model predicts the duration of the I/O request based on the workload intensity, request type and size, and sequentiality of requests. The accuracy of the model $\varepsilon \in (0.19 \dots 1.90)$ greatly depends on the representativeness of the training set. E. Anderson developed the disk I/O performance model based on the k-NN algorithm. Author uses the size of the request, its sequentiality, and length of the I/O sheduler queue to predict the running time of the request [21]. The accuracy of the model ranged in $\varepsilon \in (0.02 \dots 0.2)$. Huang et al. relied on a similar approach to build the performance model for the SSD disk using a regression tree [56] . Auhors report error $\varepsilon \in (0.17 \dots 0.25)$.

2.1.3 Simulation models

Simulation is a technique where the behavior and the structure of the model follows the behavior and the structure of the system. Although some of these models can be solved analytically, simulation remains the main tool for predicting performance using simulation models.

Simulation is the most flexible approach to performance modeling. 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.

Applications of performance simulation range from the full-system models that simulate the program at the granularity of a processor instruction [85] [30] to models that simulate large computer networks at the granularity of individual hosts [98] [60]. However, in the context of our work we concentrate on simulations of parallel and multithreaded programs.

A variety of formal methods for building simulation models have been developed. The first such methodology was queuing networks [63]. In particular, van der Mei et. al. used queuing networks to model impact of networking parameters at the performance of the web server [95]. 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 [101] [80]. 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. For example, nodes representing hardware components are placed at the lowest level of the network. Nodes themselves can create new requests and send them to other nodes in the network. This feature allows LQN simulating both synchronous and asynchronous calls in the system as well as message routing.

LQN is a popular technique for simulating message-passing and distributed systems. In particular, van Hoecke et al relied on the LQN to build models of simple CORBA applications and web services [54]. Their models have relative error within $\varepsilon \in (0.02 \dots 0.05)$. 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 [82].

Wu and Woodside [102] extend LQN by introducing Component-Based Modeling Language (CBML) – an XML-based language that describes components of the system in the terms of LQN models. CBML allows for a better modularity and reusability of LQN models. Similarly, Xu et al. [103] introduced the concept of LQN templates – a reusable fragments of the LQN model. Authors create LQN templates for different types of Java beans, and then manually compose them into a model of the distributed EJB application. The resulting model predicts throughput of the system with $\varepsilon \in (0.02 \dots 0.25)$.

The Mean Value Analysis (MVA) technique allows to obtain mean values of predicted performance metrics for the queuing model and their extensions analytically. However, MVA makes various assumptions regarding the distributions of model parameters, such as service time or arrival rate. Using MVA to model even simple threading constructs such as "fork-join" patterns [43] or mutexes [66] imposes many restrictions on the workload and on the behavior of the system that can be modeled. Moreover, a separate mathematical apparatus must be developed for modeling every single construct. As a result, author of this thesis is not aware of any applications of MVA to solve models that combine multiple simple behaviors, or models that represent a more complex behavior, such as barriers, semaphores, or concurrent I/O operations.

Another well-known simulation methodology is Petri nets and their extensions. One of the most widely used extensions is the colored Petri nets [62] 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 [86]. Authors rely on Petri nets to simulate mutexes and resource contention in the program. Although they do not simulate more complex aspects of the parallel program, such as disk I/O or complex locks.

Nguen and Apon [72] used colored Petri nets to build the model of a Linux Ext3 file system. Their implementation simulates read and write operations, and allows modeling the system's page cache and the filesystem journal. The model takes into account multiple parameters such as the block size and locality of the operation and predicts the average throughput of the system with $\varepsilon \in (0.12 \dots 0.34)$. This study was extended [73] 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 [24]. Similar to the LQN, a single node in the QPN can combine both the queue and service nodes. QPNs offer a variety of servicing, queuing and routing disciplines, which allows using them to model behavior of complex systems. In particular, Kounev, Spinner, and Meier used Queuing Petri Nets to simulate distributed component-based and event-based systems [61].

However, as the number of queues and tokens in the QPN grows, the state-space of the model grows exponentially. To make QPNs practical for modeling large systems, Hierarchical Queuing Petri Nets (HQPN) have been developed [25]. In HQPN, the single node can contain another QPN network instead of a single queue.

PEPA networks is a methodology that combines features of the Petri nets and PEPA stochastic process algebra. Gilmore et al. [46] used PEPA networks to simulate a secure Web service.

Palladio Component Model (PCM) [26] is another technique for building performance models of the computer programs. In PCM the system is divided into a number of interconnected components. Components can use computation resources such as CPU or hard drive; resource demand can be expressed by random variables. The connections between the components denote information flow in the system.

Although much more flexible than queuing networks, PCM has the number of limitations, including the absence of internal state and no support for concurrency [26]. As a result, using PCM to simulate multithreaded behavior can be problematic.

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

One notable example is the IRONModel developed by Thereska and Ganger [94]. IRONModel relies on a queuing network to simulate the flow of the request, and uses analytical models to simulate performance of certain components, such as the network and hard drive. IRONModel is used to detect bugs and performance deviations in the the Ursa Minor cluster-based storage system.

PACE framework employs hierarchical approach towards building models of MPI appilcations. 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 hardware, such as CPU, cache and network I/O subsystem. PACE uses a specialized language to describe performance aspects of the program.

The main application of PACE is the Titan predictive scheduler for the high-performance grid systems. Titan relies on PACE models to predict performance of incoming tasks; predictions are used by the optimization engine to schedule tasks in a way that improves their overall performance while meeting individual deadlines for each task. Using Titan allowed to increase overall utilization of the grid by 30-40% and improve performance of individual tasks by up to 83% [59]. Furthermore, PACE is also used to predict the execution time of the `nreg` medical image processing application with the $\varepsilon <= 0.1$ [58].

Simulation models are more flexible then analytical or statistical models. However, their construction is significantly more difficult, mostly because they require extensive information on the system's internals and functionality. This information can be retrieved manually, as in [101], [95], [54], [103]. 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.

2.2 Automatic analysis and performance modeling of computer programs

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

2.2.1 Automatic analysis of computer programs

A significant amount of work has been dedicated to automated analysis and comprehension of parallel and distributed programs. Techniques for program analysis, used in these works, can be also used for the purpose of generating performance models automatically.

Program analysis have been extensively used to understand structure of software systems, and semantics of interaction between its components. In particular, Reiss proposed CHET - a tool for extracting specifications from the parallel program itself [77]. CHET uses symbolic execution to discover interactions between the program's components and then extracts behavioral specifications of the program in a form of automata.

Similarly, Burnim and Sen discovers deterministic specification in the multithreaded programs [32]. In particular, 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.

Barham et al develop Magpie - a tool for understanding the characteristics of the system's workload [22]. Magpie tracks key events in the system's functioning by tracking a selected set of API functions. Then Magpie relies on the user-supplied schema to infer the flow of request the from the sequence of API calls. Magpie employs a sophisticated algorithm to accurately retrieve resource demands for the request.

Similar approaches have been extensively used to study behavior and performance of distributed systems. Chen et al developed a framework to track the flow of request between the components of such system [34]. Authors use this information to identify request flow paths that are associated with the increased risk of failure. Hellerstein et al propose an ETE framework, which measures and

visualizes performance of transactions in distributed systems [51]. Similarly to Magpie, ETE relies on the user-provided schema to reconstruct the sequence of events in a transaction. Aguilera with co-authors propose a low-invasive approach to identify performance bottlenecks in the distributed legacy system [18]. Authors rely on signal processing techniques to identify causality of calls between the nodes in the system.

Program analysis techniques have 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 [91]. 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 [38]. Their profiler automatically measures how the input size of the program's function affects the running time of that function. Authors measure input size as the number of distinct memory cells read by the prologue of the 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 [105]. 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 [47]. Authors instrument the program and run it on different workloads. Then authors approximate the execution time of different basic blocks of the program using linear or power law. The result is the approximate formula that describes the computational complexity of the program.

Some approaches towards analysis of parallel programs rely on stack sampling in order to reduce the analysis overhead. Tallent and Mellor-Crummey develop a technique for identifying parallel idleness and parallel overhead in the multithreaded program [90]. 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 [68].

Software visualization is another direction where program analysis is used extensively. DeRose and Reed proposed a SvPablo - a system for analyzing performance of parallel and distributed programs written using a variety of languages [84]. SvPablo instruments the program first. As the program executes, the tool collects and summarizes data from multiple processors. Once execution is over, SvPablo integrates different data sources and presents them to the user.

Reiss developed an approach to understand the structure of the program for the purpose of visualization [78]. He discovers threads, transactions, and tasks in the multithreaded system and their semantics of integration using dynamic program analysis. The resulting information is visualized to facilitate program understanding.

By working along these lines, Reiss produced a system to identify locks (synchronization mechanisms) in a multithreaded program and to determine their types [76]. 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.

2.2.2 Automated generation of simulation models

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 [55]. The study assumes the request-processing system, where each request has a unique identifier. As the request flows through the system, individual components record the request ID, its arrival and departure times. 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 [57] 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 [100] 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 $\varepsilon = 0.3$.

Zheng et al propose a methodology to predict the parameters of the LQN model using the autoregressive statistical model [106]. First, authors use Kalman filter to estimate values of the model parameters. Then authors use an autoregressive model to predict future values of these parameters based on their past values. This work is important in a sense that it allows using existing performance models even if workload parameters have changed.

Brosig, Huber, and Kounev [28] 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. In most cases their predictions of CPU utilization and the response time of the system are accurate within $\varepsilon \in (0.1 \ldots 0.3)$. However, in configurations with intense workload the relative error $\varepsilon \geq 0.5$.

In [65] authors show that the PCM can be automatically transformed into the queuing Petri network (QPN). The resulting QPN can be solved analytically, which allows increasing the performance of the model significantly. However, the accuracy of prediction for certain metrics (e.g. the response time) can decrease as a result of a conversion.

Hauk, Happe and Reussner use a set of heuristics to automatically deduce the type of the load balancing algorithm used by the OS thread scheduler [50]. They distinguish between lazy, active and immediate load-balancing strategies. The resulting type of the scheduler is integrated into the PCM model, which increases the accuracy of the prediction.

Xu and Subhlok [104] automatically extract the topology and resource demands of distributed MPI applications from their traces. Their models successfully predicted the performance of the application running on a different cluster ($\varepsilon \leq 0.15$). However, the accuracy drops to $\varepsilon \in (0.3 \ldots 0.5)$ in configurations where the system experiences resource contention due to sharing of CPUs between multiple processes.

PACE framework [74] described earlier can 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.

Resource demands for the program can be discovered by instrumenting the program and measuring the resource demands of its individual components [22]. 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 [81][83]. Similarly, Brosig et al rely on a Service Demand Law [29] to infer resource demand for PCM-based models.

Another interesting approach for building models automatically can be the "design-for-simulation" paradigm, where the software system is capable of generating a performance model for itself. A notable example of such system is the Ursa Minor cluster-based storage [94]. Ursa Minor uses this model to detect performance anomalies in the program.

Despite a great variety in techniques for automated modeling of computer programs, they share one common feature: virtually all of them are aimed at modeling 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. To overcome this limitation we propose a modeling technique that can correctly simulate behavior of a multithreaded application and a framework that can automatically generate these models.

Chapter 3

Model definition

In this section we define the model for predicting performance of multithreaded programs. We first describe our representation of computations in the parallel program for the purposes of performance prediction. Next we present the architecture of our performance model. Finally, we discuss approaches for simulating resource contention by the model.

3.1 Representing computations in a multithreaded program

For the purpose of simulation we represent computations performed by the program as request processing. We denote a *request* as something the program has to react to. The program processes the request by performing certain operations.

This approach naturally allows simulating reactive systems which constitute a majority of modern computer programs. For example, in a web server a request corresponds to an incoming HTTP connection. The web server reacts to the request by reading a web page from the disk, generating a response, and sending it to the user. In a 3D renderer a request can denote a pixel, or a subset (a row, a column, or a tile) of pixels. Processing of the request involves rendering the scene and generating the final image. In a scientific application the request can correspond to an object in the physical system, whose behavior is simulated by the program. The program responds to the request by performing computations to determine the next state of the object based on its current state and the states of objects in the system.

The request itself is an abstract entity. In the multithreaded program the request is represented by a *task*. The task is a data structure that is created as a response to the request. It represents a discrete unit of work that can be performed by the thread in the program [75]. Normally tasks are implemented using a certain data structure, which may be a class in an object-oriented program, or a data structure.

The performance of the request processing system can be described by various metrics, such as the response time R (an overall delay between request arrival and its completion), throughput T (the number of requests served in the unit of time), or the number of requests dropped. In our work we

do not predict performance of individual requests, but rather the aggregated values of performance metrics over a period of time.

We build discrete-event models of multithreaded systems. The discrete-event simulation is a model, where the simulation time t is advanced by discrete steps. It is assumed that the state of the system does not change between time advances.

Our models are built according to the hierarchical principle [42] and consist of three tiers. The high-level model explicitly simulates the flow of requests as they are being processed by the program, from their arrival to completion. The middle-level models simulate delays that occur inside the program's threads as they process requests. 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.

3.2 High-level model

The high-level model is based on a queuing network model [63] where queues $\{q_1, ..., q_n\}$ correspond to the program's queues and buffers, and service nodes $\{tr_1, ..., tr_m\}$ correspond to the program's threads (the full notation used to describe the model is provided in the Table 3.1). The set of queues in the high-level model represent 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 3.1 (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. The thread places the 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, retrieves the necessary request information, 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 request, it calls the model of the corresponding thread to simulate the amount of time necessary to process that request. Finally, the high-level model does not explicitly define service demand S for requests; 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 R, T, or the number of requests in the system.

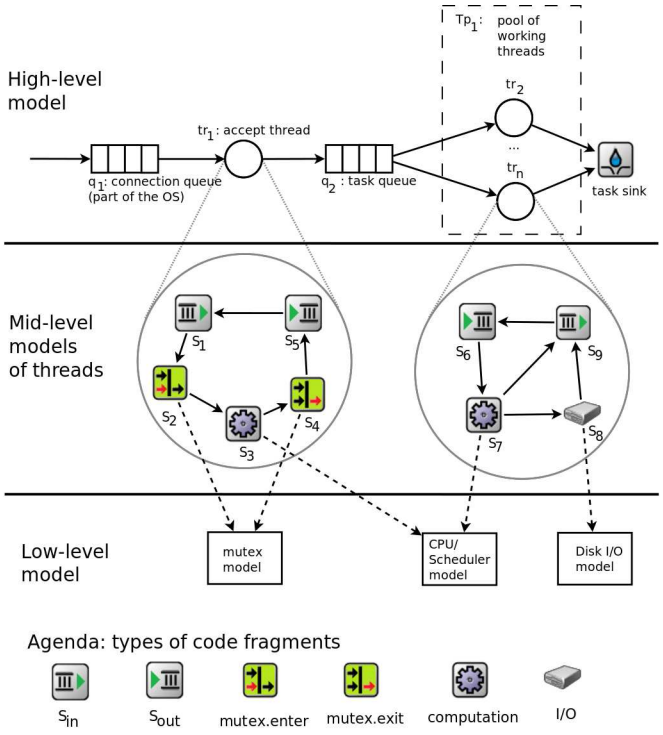

Figure 3.1: A model for a web server

3.3 Middle-level model

Our middle-level models simulate the *delays* that occur in the program's threads as they process tasks. The thread model represents a probabilistic call graph (PCGs) for the corresponding thread. Each vertex $s_i \in S$ of such PCG 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 of the graph represent a possible transition of control flow between the CFs, which is modeled probabilistically. Namely, 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

$$\sum_{j=k}^{m} p(s_i, s_j) = 1 \tag{3.1}$$

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 $S_{next} = 0$. 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 computations performed by the program: delays

caused by computation, delays caused by input-output (I/O), and delays caused by synchronization activities. These correspond to basic *classes* $\{c_{CPU}, c_{IO}, c_{sync}\} \subseteq C$ of code fragments: computation code fragments (denoted as c_{CPU}), I/O code fragments (c_{IO}), and synchronization code fragments (c_{sync}). There are two additional CF classes $\{c_{in}, c_{out}\} \subseteq C$ used to communicate with the high-level queuing model. With regards to the performance modeling, the class c_i of the code fragment s_i is one of the most important characteristics of that CF.

I/O code fragments $s_i : c_i = c_{IO}$ represent executions of the program constructs that perform I/O, either directly or indirectly by reading filesystem metadata. Examples of such CFs in a Java program are the calls to `File.exists(), File.isDirectory(),FileInputStream.readBytes(), and FileOutputStream.write()` methods.

Synchronization code fragments $s_i : c_i = c_{sync}$ represent synchronization operations. In a Java program synchronization CFs correspond to following constructs:

- calls to synchronization primitives such as `Object.wait(), Semaphore.acquire(), CyclicBarrier.await()`;
- entering or exiting from the Java synchronized regions and synchronized methods;
- executions of program constructs that explicitly alter the threading behavior of the program, such as calls to `Thread.sleep()` or `Thread.join()`.

The type of the synchronization operation determines the *type* of the synchronization code fragment.

Computation code fragments $s_i : c_i = c_{CPU}$ represent computations performed by the thread.

The thread model communicates with the higher-level queuing model using input and output vertices. Input vertices $s_i : c_i = c_{in}$ can fetch requests from one or more of the queues $\{q_i, ..., q_j\}$ of the high-level queuing model. As a part of this the thread model can suspend its execution until the request become available. Output vertices $s_i : c_i = c_{out}$ generate requests and send them to one of the queues $\{q_i, ..., q_j\}$ in the high-level model.

In the context of the multithreaded program, c_{in} and c_{out} CFs correspond to operations on the program's shared queues. Namely, the c_{in}/c_{out} CF may represent the producer-consumer pattern in a multithreaded program. In particular, the c_{out} CF that represents the "producer" may correspond to calling `BlockingQueue.add() or BlockingQueue.offer()` methods in a Java program. An appropriate c_{in} CF that represents the "consumer" will correspond to calling the `BlockingQueue.take() or BlockingQueue.poll()` method. c_{in}/c_{out} CFs can also represent operations with OS queues. One example of such c_{in} CF is the call to the `ServerSocket.accept()` that fetches connection requests from the queue for incoming network connections.

For an example, consider a (simplified) model of a web server depicted at the Figure 3.1. The accept thread listens for incoming connections (represented by the CF s_1 in its thread model). Once the connection has been accepted, the accept thread creates a task object (s_2 - s_4) and puts (s_5) it into the task queue. Once one of the working threads becomes available, it fetches (s_6) the task from the queue and processes it (s_7 - s_8). The working thread verifies that the requested page exists,

reads the corresponding file from the disk, prepares the response, and sends it to the client. Finally, the thread closes the connection and fetches the next task from the queue.

Formally, our thread model can be described as a *probabilistic time automaton (PTA)* [89]. According to the notation presented in [89], the PTA can be defined as a tuple $\langle S, s^0, \delta, T, \Theta \rangle$, where:

- $S = (s_1, ..., s_n)$ are the PTA states, which correspond to the vertices of the probabilistic call graph;
- $s^0 \in S$ is a start state;
- $\delta : S \to P(S)$ is a distribution that represents probabilities of transition from one state to another;
- $T = (\tau_1, ..., \tau_n); \tau_i \in \mathbb{R} \geq 0$ are delays occurring when the automaton transitions to states $s_1, ..., s_n$ respectively.
- $\Theta = (\theta_1, ..., \theta_n)$ are action flags that describe actions performed when the PTA transitions to the corresponding states $s_1, ..., s_n$. The action $\theta_i \in \{c_{CPU}, c_{IO}, c_{sync}, c_{in}, c_{out}\}$ corresponds to the class of the CF and defines how the thread model interacts with other models in hierarchy. External actions $\{c_{in}, c_{out}\}$ allow the PTA to directly interact with the high-level queuing model, and internal actions $\{c_{CPU}, c_{IO}, c_{sync}\}$ interact with the low-level model.

  For the purpose of modeling we consider actions θ to be mutually exclusive. Namely, each state s_i is described with only one single action $\theta_i \in \{c_{CPU}, c_{IO}, c_{sync}, c_{in}, c_{out}\}$.

The PTA works as following. At the start the PTA transitions to the state s^0. When the PTA enters some state s_i, it performs the action θ_i and introduces the discrete delay τ_i. After delay is expired, the PTA transitions to another state s_j according to the transition distribution δ.

In [89] transitions are also influenced by external events $a_i \in \Sigma$, so the transition distribution takes the form $\delta : S \times \Sigma \to P(S)$. Currently we do not specifically define any external events in the model. We assume $\Sigma \equiv \varnothing$, so transitions between the PTA states are completely probabilistic. However, we may incorporate the notion of events in future. In particular, these events may represent certain cases of determinism in the behavior of the multithreaded program we'll describe in the next sections of this thesis.

3.4 Low-level model. Simulating time delays in threads

Execution of each code fragment (CF) results in the delay τ. While the call graph structure $\langle S, s_0, \delta \rangle$ 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 state 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-tier models. In particular, the lower-tier models simulate following elements of the parallel system:

- the OS thread scheduler and the CPU;
- the OS I/O scheduler and the hard drive;
- the synchronization constructs (*locks*) $\{l_1, ..., l_m\} \in L$ present in a multithreaded 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.

In this work we explicitly simulate delays caused by CPU-intense computations, disk I/O operations, and synchronization operations. We do not explicitly model memory and cache operations. Instead, operations such as memory allocation or memory access are simulated as computations. We anticipate that for certain subset of programs and workloads saturation of the memory bandwidth may inflict a noticeable impact on performance. Thus a lightweight and accurate model of a memory subsystem remains a future work.

We do not simulate disk I/O operations caused by page faults (swapping). However, page faults are improbable in a well-configured program, so currently we ignore them. We also do not simulate the asynchronous disk I/O operations, which includes the model for disk writes.

We also do not have a dedicated model of network I/O operations. Delays caused by network I/O are currently simulated implicitly as CPU computations. We have built models of a number of networking applications, and the absence of the network model did not cause noticeable problems with models' accuracy. However, we anticipate that for some workloads network operations may have a noticeable impact on performance of the system. Thus the model of the network I/O is left as a subject for a future work.

3.4.1 Modeling CPU Computations

CPU-intense computations are simulated by the computation CFs whose type is c_{CPU} (denoted as $s_i : c_i = c_{CPU}$). Parameters of these CFs are $\Pi_{CPU} = \{\tau_{CPU}\}$, where τ_{CPU} is the *CPU time* for

the CF s_i. 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 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 back of the "ready" queue. Once the service node becomes available, it fetches the job from the queue and introduces a delay equal to $min(\tau_{CPU}, OS\,time\,quantum)$. After the delay is expired, the CPU/Scheduler checks if computations are complete for the job. In this case the CPU/Scheduler deletes the job and notifies the thread model. Otherwise it places the job back into the "ready" queue, where it awaits another time quantum.

3.4.2 Modeling disk I/O operations

I/O operations are simulated using I/O code fragments $s_i : c_i = c_{IO}$, whose parameters form a distribution $\mathbb{P}^{\Pi}_{IO}$. Members of this distribution are tuples $\Pi_{disk} = \langle dio_1, ..., dio_k \rangle, k \geq 0$ of low-level disk I/O operations initiated by s_i. Properties of each I/O operation dio_j include the amount of data transferred and the type of the operation. Currently our model supports following types of disk I/O operations: regular file read, readahead, and the metadata read. These cover different types of disk read operations which can be encountered in practice.

The size k of the tuple Π_{disk} denotes the number of I/O operations issued by s_i and allows implicit modeling of the system's page cache. It has been shown [41] that after performing a sufficient number of I/O operations the cache reaches a steady state where the probability of cache hit converges to the constant value. In the terms of our model, the distribution $\mathbb{P}^{\Pi}_{IO}$ for the state s_i becomes stationary after serving a sufficiently large number of requests (10^4 to 10^5 in our experiments). This allows simulating effects of a page cache on disk I/O operations.

When the mid-level thread model must simulate the I/O CF, it fetches a sample of disk I/O operations $\Pi_{disk} = \langle dio_1, ..., dio_k \rangle$ 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 (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.

DiskIO is a queuing model whose queue represents the request queue in the actual I/O scheduler, and the service node represents the hard drive. Upon receiving a call from the mid-level model, the DiskIO inserts a new job into the queue. The actual I/O scheduler arranges requests according to the index of the disk block they are accessing. But since this information is not known to the model, jobs are inserted at the beginning of the queue and the service node fetches them from the random

positions.

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} depends on on many factors: the locality of the operation (how close are the disk sectors accessed by different requests), the number of requests in the queue, and others. Since many of these factors are beyond the control of the model, 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. Typically, modern OSes handle disk writes asynchronously using a strategy called write-back. Namely, the data to be written is transferred into the OS write buffers first. After buffers accumulate a certain amount of data, one of the OS working threads writes the contents of these buffers to the hard drive.

Modeling write I/O operations would require either explicitly simulating the behavior of the OS write cache and corresponding caching policies, or modeling write operations using a Markov chain. 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.

3.4.3 Modeling synchronization operations

Synchronization operations are simulated using synchronization code fragments $s_i : c_i = c_{sync}$. 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 this parameter depend on the type of the lock itself. For example, the possible values of *optype* for a semaphore are $\{wait, signal\}$, for a mutex *optype* may be $\{acquire, release\}$, and for a barrier *optype* is $\{await\}$;
- $\tau_{sync} \in (0, ..., \infty)$ is the timeout for synchronization operation. The default value of the 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.

The lower-level model simulates each lock $\{l_1, ..., l_m\} \in L$ in the program. Locks are simulated at the semantic level. Namely, if a program implements a complex lock such as a barrier using a set of low-level constructs such as Java synchronized sections and monitors, we model it as a single high-level lock (in our case - a barrier). We developed separate models for barriers, semaphores, mutexes, etc.

Each lock $l_j \in L$ is described using a tuple of parameters $\langle ltype, lparam \rangle$, where

- *ltype* is the type of the lock, such as a semaphore, a barrier, or a mutex;
- *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.

Once the low-level model receives a call from the synchronization CF s_i, it locates the lock l_j specified by the Π_i. The lower-level model explicitly simulates behavior of each lock. For example, when a model of a particular thread calls the model of the barrier, that barrier model adds a reference to the caller thread to the list of waiting threads. If the size of the list is equal to the capacity of the barrier, the barrier notifies all the waiting thread models that the delay is complete. Otherwise it waits for the call from another thread.

If the wait time becomes higher than the timeout τ_{sync} specified by the calling synchronization CF, the lock model notifies the calling CF that the operation has timed out.

Table 3.1: Notation used for description of the model and its parameters

Notation used in a high-level thread model	
$tr_1, ..., tr_n$	A set of all threads present in the program
$Tp_1, ..., Tp_m, m \leq n$	A set of all threads pools present 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 PCG
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
$\delta : S \rightarrow P(S)$	Transition probabilities between nodes of the probabilistic call graph
$\Pi_{disk} = \langle dio_1, ..., dio_k \rangle$	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
$\Pi_{sync} = \langle l_i, optype, \tau_{sync} \rangle$	Parameters of a synchronization CF: the lock which is called, the type of synchronization operation, the timeout of synchronization operation
$\Pi_{sin} = \langle \{q_i, ..., q_j\}, \tau_{out} \rangle$	Parameters of c_{in} CF: a set of queues from which the task can be fetched, the timeout of operation
$\Pi_{sout} = \langle q_i, ..., q_j \rangle$	Parameters of c_{out} CF: queues to which the task can be sent to
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

Chapter 4

Model Implementation

In this section we describe **PERSIK (PERformance SImulation Kit)** – a framework we developed for building performance prediction models. PERSIK is used to implement performance models according to the methodology described in the Section 3.

We first provide a necessary background by describing the OMNET++ modeling framework, which serves as a base for implementing PERSIK. Next we describe the architecture of PERSIK model and its individual components. Then we evaluate PERSIK by building models of two multithreaded programs.

4.1 OMNET++ modeling framework as a basis for PERSIK

PERSIK itself is implemented on the base of the OMNET++ discrete event simulation framework [2]. To facilitate understanding of PERSIK below we provide a brief description of OMNET++ models and their architecture.

OMNET++ was initially designed to simulate computer networks and distributed systems. Unlike other tools for discrete event simulation, such as SimEvents [3], OMNET++ does not offer a ready set of components for implementing models of networks. Instead, OMNET++ provides a generic platform for implementing discrete event models. It is a responsibility of the developer to implement the set of components necessary for building a particular type of a model [96]. These components can form a library which can be reused for building similar models.

Building OMNET++ models may require investing a significant amount of time into implementing individual components of the model. Nevertheless, such approach allows building more flexible and diverse models. This flexibility allowed using OMNET++ for building a wide range of simulations. In particular, OMNET++ applications include the Mobility Framework [39] for simulating wireless and mobile networks and the INET Framework [4] for simulating IP-based wired communication networks. Similarly, PERSIK is implemented as a set of components that allow simulation of multithreaded programs in OMNET++.

An OMNET++ model consists of interconnected *blocks* communicating using messages [96]. In

the terms of OMNET++ blocks are also called *modules*, thus we will use both terms interchangeably throughout this thesis. Modules represent components of the system being simulated. For example, in the model of a distributed system blocks (modules) may represent hosts, routers, and other elements of the system.

OMNET++ models are hierarchical. At the bottom level, the basic units of OMNeT++ models are the *simple modules*. Simple modules define the *behavior* of the model. OMNET++ does not provide a pre-defined set of simple modules. Instead, this is a developer's responsibility to implement simple modules using C++ language.

A set of interconnected simple modules can be grouped into the *compound module*. In turn, compound modules can be grouped into higher-levels OMNET++ modules. The number of layers in a hierarchy is not limited. Such hierarchy allows an efficient componentization of the model and reuse of individual blocks. At the highest level OMNET++ blocks are grouped into the *network model* of the system being simulated.

Both simple and compound modules are instances of corresponding *module types* [96]. Modules of the same type have same behavior, but may have different parameters. Correspondingly, the model may contain multiple instances of the module of the same type. For example, a certain module type may represent the host in a distributed system, which may receive and generate the requests. Another module type may represent a router, whose functionality is re-routing requests between the hosts. A model of a distributed system may contain a number of hosts and routers. This relation between module types and the actual modules resembles the relation between classes and class instances (objects) in an object-oriented programming.

Modules communicate using *messages*. Modules receive and send messages using input and output gates respectively. An output gate of one module can be linked to the input gate of another module using a *connection*. In this case the connection can serve as a route for modules to exchange messages. Altogether, modules and connections constitute the graph. The structure of this graph represents the *topology* of the model.

It is possible to have multiple incoming connections for a gate, so the single module may receive messages from multiple transmitting modules. Correspondingly, the single output gate of a module may be connected with input gates of multiple receiving modules. However, in this case the problem of routing arises. Namely, the transmitting module must decide to which receiving module it will send the message.

The message can be also sent to the receiving module directly, without passing it through the output gate. We use this approach to send message to the models of locks and hardware in PERSIK.

The OMNET++ allows defining parameters for blocks. Parameters are identified by their names and may take string, numeric, or boolean values. Parameters can be defined for simple modules as well as for compound modules (in the latter case the compound module passes the values of its parameters to the inner modules). Normally parameters are used to pass configuration values to the model blocks and to control behavior of these blocks. In fact, parameters can be seen as an interface between the model internals and the outside world. Applications of block parameters

include distributions for generating or servicing requests, defining routing in the network, or altering behavior of individual modules.

Similarly to blocks, messages may have parameters too. Unlike the block parameters that are used to configure the model, the message parameters are used solely for the internal purposes within the model. For example, PERSIK relies on parameters to assign timestamps to the messages and to pass information between the model blocks, such as the resource demand or the type of operation to be carried out by the block. Values of message parameters can be set only by the model blocks and cannot be defined from the outside.

A combination of parameters for all the modules define the configuration of the model (it is important to emphasize that message parameters are not a part of the model configuration). Usually the goal of an analyst is to study the behavior of the model across a range of different configurations. This necessitates running the model with different values of modules' parameters. To increase the flexibility, OMNET++ physically separates the values of model parameters from the definition of the model's behavior and its structure. As a result, the PERSIK model consists of the following parts:

- the **behavior** of the model is implemented in modules.

  As a part of our work on PERSIK we developed a library of modules, which can be later combined into the model of the parallel program. Most of these modules are simple modules written using using C++. Remaining modules are the compound modules, which consist of a number of interconnected simple PERSIK modules.

- the **topology** of the model defined using the set of .ned files.

  .ned file is a textual file that contains the list of blocks in the model, their types, names of the blocks (the name of the block is used as a unique identifier of the block within the model), and connections between these blocks.

  The PERSIK model includes one .ned file to describe the high-level model and a separate .ned file to describe each working thread. These files are created separately for every model of the program.

- the **parameters** of individual modules (blocks) in the model defined in the .ini file.

  .ini file is the text file that contains the list of parameters in the ``name = value'' format, where `name` is the full name of the model block concatenated with the name of the parameter.

  Every PERSIK model has one .ini file that defines parameters of the model.

Such separation of the model's functionality simplifies reuse of the blocks across different models. More importantly, it also simplifies multiple runs of the same model in different configurations. In this case the only part of the model that must be updated is the .ini file.

4.2 The architecture of PERSIK models

We implemented PERSIK framework as a library of OMNET++ modules (blocks) and messages. The majority of these blocks correspond to entities in formal model. For example, the queue is represented as a queue block, while the c_{in} CF is represented as a computation block in PERSIK. Similarly, parameters of the formal model, such as the distribution of execution times $\mathbb{P}^{\tau}_{CPU}$ for the CF, correspond to parameters of corresponding PERSIK blocks.

PERSIK models generally follow the hierarchical modeling architecture outlined in the section 3. The largest architectural difference between the formal models and PERSIK models is that PERSIK models are not three-tiered but two-tiered. The upper-level PERSIK model is a queuing model of the system. It also contains models of locks, I/O subsystem, and the CPU/Scheduler model, which occupy the lower-level tier of our formal model. The lower-level PERSIK model implements the models of working threads, which correspond to the middle tier of the formal model. Upper-level and lower-level PERSIK models are composed of different classes of blocks.

The architectural differences between the formal and PERSIK models intend to facilitate the actual implementation of the models. These differences are of a cosmetic nature and do not violate the semantics of our formal model. Namely, there is no direct interaction between the PERSIK blocks in the queuing model and blocks that represent locks, OS and hardware. Instead, the upper-level PERSIK model calls the models of working threads, and these thread models call blocks which simulate locks, I/O subsystem and CPU/Scheduler.

Such strict correspondence between elements of the formal model and blocks and parameters of the PERSIK model greatly simplifies translation of the formal model into the PERSIK model.

4.2.1 Upper-level PERSIK models

The upper-level PERSIK model creates requests, queues them, and sends them to low-level thread models for processing. The upper-level model contains blocks that represent request sources, sinks, queues, threads, and program-wide locks (barriers, critical sections etc). The upper-level model also contains blocks simulating the I/O subsystem and the thread scheduler. The complete list of upper-level PERSIK blocks and their properties is provided in the Table 4.2, while Table 4.1 lists properties common for all PERSIK blocks.

As in the formal model, each thread in the upper-level PERSIK model is represented as a separate block. For example, if the program has 8 working threads, these correspond to 8 thread blocks in a PERSIK model. In the upper-level model threads appear as "black boxes" without any notion of their internal structure. Instead, thread blocks are implemented as the compound OMNET++ modules.

The request flow is simulated by a *request message* flowing from one block to another. The request messages normally correspond to tasks in the real-life program.

The Figure 4.1 depicts the high-level model of the Tornado web server [5] with 1 working thread. The internal structure and the behavior of the Tornado follows the behavior of the example web

server described in the Section 3 (see Figure 3.1). Requests in this model correspond to the HTTP requests, and arrows depict flow of the requests between the model's blocks. Three blocks on the upper left of the figure denote OS and hardware models: `diskIOModule` block is the model of the disk I/O subsystem, `cpuScheduler` is the model of the thread scheduler, and `osLimits` simulates OS limitations (see the section 4.2.3 below).

This high-level model contains models of three threads. The `main_4` block is the model of the main thread that launches the remaining threads, the `ListenThread_5` block is the model of the accept thread, and the `ServerThread0_0` is the model of the working thread. `connectionSrc` generates incoming network connections and `connectionQueue` routes them to the accept thread. These two blocks represent to the networking components of the operating system. The `taskQueue` simulates the request queue in the web server.

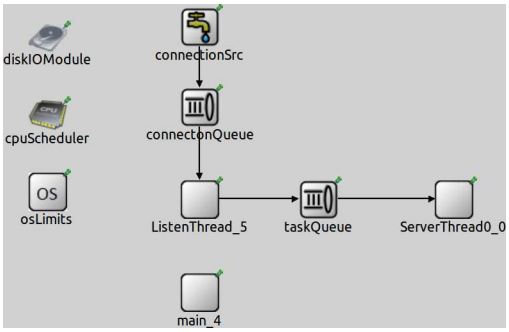

Figure 4.1: The high-level PERSIK model of the Tornado

4.2.2 Thread models

Thread internals are simulated with the low-level thread model. The thread model implements a probabilistic call graph (PCG) for a given thread. Each code fragment $s_i \in S$ of the PCG is represented by a corresponding block in the thread model. Computation, I/O, and synchronization CFs are represented by the blocks of corresponding types. The complete list of blocks allowed in PERSIK thread models is provided in the Table 4.3.

Execution flow in the thread model is simulated by a *computation flow message (CFM)*. Upon start of the simulation the thread model creates the CFM using the `sourceOnce` block and sends it to the initial block that represents the s_0 CF. Then the computation flow message starts traveling through the call graph of the thread, which simulates the execution of the actual thread. The flow of the CFM is normally controlled by the *dispatch blocks* that implement probabilities δ. In rare occasions (see Section 5.4.2) the control flow is controlled by loop blocks that explicitly simulate loops in the program's code.

In addition to blocks that represent code fragments, thread models also contain service blocks. One class of service blocks is the `stopper` block, which stops the simulation after receiving the predefined number of CFMs. Another class of service blocks are `setTimer` and `readTimer` blocks,

which are responsible for collecting results. The collecting of prediction results is described in the Section 4.2.4 in more detail.

To pass the request message in and out of the thread we rely on a `reader` and `writer` blocks that implement c_{in}, c_{out} actions of our formal model.

Once the CFM reaches the `reader` block, that block attempts to fetch the request message from the queue in the upper-level model. If no request is available, the `reader` block delays the CFM until the request becomes available, or until a timeout is passed. This behavior of the `reader` block correspond to a locking behavior in the producer-consumer pattern.

The timeout is specified using a configuration parameter of the model. If the queue is still empty after the timeout, the reader block reroutes the the CFM. This behavior allows modeling the cases of the determinism in the multithreaded program.

Similarly, when the CFM reaches the `writer` block, the `writer` outputs a request message to the queue of a high-level model. If the queue cannot accept the request (e.g. it is already full), the writer will wait until the queue becomes available again. If the queue is still not available after a specified timeout, the block will reroute the CFM.

The `reader` can attach the retrieved request message to the CFM. Correspondingly, when such CFM reaches the `writer` block, `writer` outputs the request message to the upper-level model. This allows collecting statistics on requests flowing between the thread boundaries.

`reader` and `writer` blocks can access multiple queues. If more that one queue can be accessed at the moment, the actual queue will be chosen according to one of the predefined policies. Namely, the queue may be choosen randomly, by the round-robin principle, or according to the pre-defined priority for the queues. The list of the queues that can be accessed by the block and a specification of a policy for accessing these queues are defined by configuration parameters.

Depending on the values of configuration parameters, `reader` and `writer` blocks may exhibit a variety of different behaviors. Such flexibility allows PERSIK to simulate a variety of different queue types.

The Figure 4.2 shows thread models for the Tornado. Here arrows depict the flow of the CFM between the threads' code blocks.

The right part of the figure depicts the model of the accept thread (`ListenThread_5` in the upper-level model). Upon the start of the model the `start` block generates the CFM and sends it to the `accept` initial block. `accept` is the `socket_accept` block that corresponds to the call to the `ServerSocket.accept()` method. `accept` retrieves the incoming request from the `connectionQueue` queue. `prepareRequest` represents the code that performs initial processing of the request; the `putRequest` is the c_{out} block that emits the request to the `taskQueue` queue in the upper-level model. The `startTimer_requestStart` block is the `setTimer` block used to collect performance measurements.

The left part depicts the model of the working thread (`ServerThread0_0` in the upper-level model). The `fetchRequest` is the initial block; it is an s_{in} block that fetches requests from the `taskQueue` queue. Once the request is fetched, the CFM is routed to the `prepareStat` block, which

simulates computations related to initial processing of request. Then the the CFM reaches the the `doStat` block. `doStat` is a disk I/O block; it simulates the call to the the filesystem metadata that verifies if the requested page exists. If the page does not exist, the `dispatchStat` dispatch block reroutes CFM back to the `fetchRequest` block. Otherwise, the loop consisting of `doFileRead`, `processFileBlock`, and `dispatchRead` blocks simulates reading the file in chunks and sending its contents back to the client. Blocks `stopTimer_Success` and `stopTimer_NoFile` are the `readTimer` blocks that record the response time for the request.

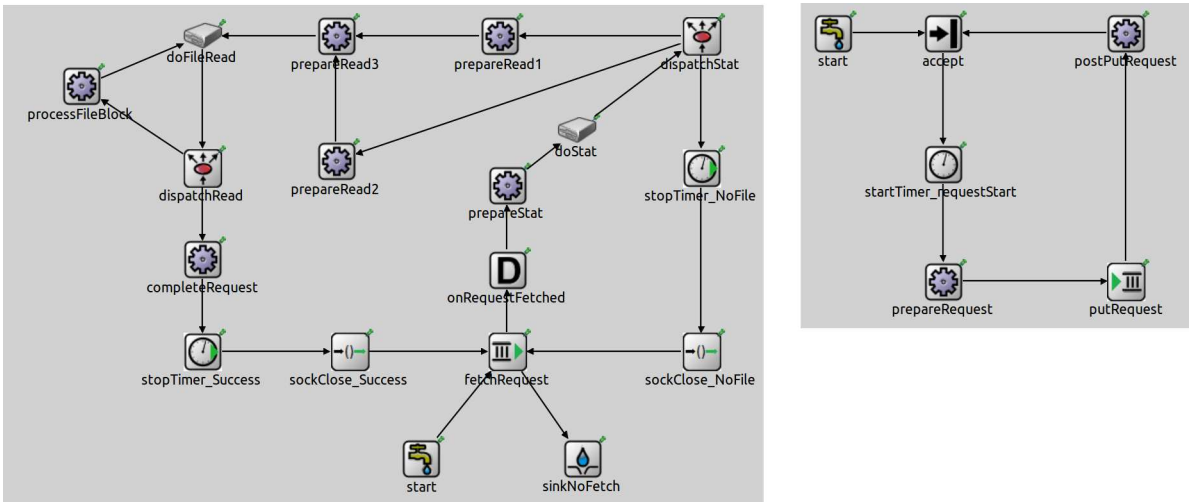

Figure 4.2: PERSIK models of the Tornado threads: working (left) and accept thread (right).

4.2.3 Simulating Delays in Thread Execution

As the CFM travels through the thread model, various blocks can delay its passage, thus simulating delays τ that occur during the thread execution. These delays occur because of CPU-intense computations, I/O activities, or when execution of the thread is blocked by synchronization mechanisms such as critical sections or semaphores. As it was mentioned in the Section 3, the duration of these delays depend on resource contention and must be carefully modeled.

To simulate delays caused by resource contention two groups of blocks are employed: *caller blocks* and *central blocks*. Different types of caller and central blocks are used to simulate different causes of delays. However, all of them interact according to the same principle.

Caller blocks are parts of the thread model. Examples of these blocks are computation blocks, I/O blocks, and synchronization blocks which simulate execution of c_{CPU}, c_{IO}, and c_{sync} CFs correspondingly. Each caller block contains the reference to the corresponding central block. When the caller block receives a CFM, it delays that CFM message. Next, it forms a separate *inter-layer message*. The exact class of that inter-layer message depends on the type of the caller/central block pair. The parameters of the inter-layer message are sampled from the $\mathbb{P}^{\Pi}$ – distribution of parameters for the caller block. That inter-layer message is directly sent to the corresponding central block.

Central blocks are parts of a upper-level PERSIK model. However, they correspond to the entities of the lower-level formal model. Central blocks implement the model of CPU and thread scheduler (represented by the CPUScheduler block), the model of Disk I/O subsystem (the DiskIO block), and the models of locks $l_1, ..., l_m$, which are implemented by different block classes.

Once the central block receives the inter-layer message from the caller block, it updates the internal state of the model $Q(t)$. Then, it uses message parameters and the $Q(t)$ to simulate the delay τ. Once the delay has passed, the central block sends the inter-layer message back the caller block. The caller block in turn sends the original CFM to the next block in the thread model.

Simulating synchronization delays

To simulate high-level synchronization primitives in the program the modeling framework employs a range of different block classes. Every lock in the program is represented by a corresponding central block. Parameters of the central block correspond to properties of the lock. For example, the barrier block has one parameter – the capacity of the barrier. Correspondingly, caller blocks represent calls to these locks.

Every type of synchronization primitive is represented by different central/caller block pair. For example, the barrier is represented by a `SyncBarrier` central block; calls to the barrier are represented by `SyncBarrier_await` caller blocks.

Central blocks explicitly simulate the functioning of the lock. When the caller block sends the *synchronization message* to the central block, the central block updates its internal state accordingly and makes a decision if the calling thread should block or not. If the thread should not block, the central block sends the synchronization message back to the caller immediately. However, if the central block decides that the calling thread must wait, it delays sending the synchronization message back until the caller can be unblocked.

Simulating computations

To simulate delays that occur due to CPU-intense computations the model uses a combination of computation blocks (caller blocks) and a CPU/Scheduler block (a central block). In addition to simulating delays, the CPU/Scheduler also measures CPU utilization by the program.

When the computation block sends the *computation message* to the CPU/Scheduler block, it passes a τ_{CPU} as a parameter of that message. τ_{CPU} is sampled from the $\mathbb{P}^{\Pi}_{CPU}$ for the corresponding code fragment. The distribution $\mathbb{P}^{\Pi}_{CPU}$ is normally represented as a sample of execution times for a code fragment, obtained from running the instrumented version of the actual program. The sample is stored as an array, and the particular value of the τ_{CPU} is sampled from the random position of that array. Alternatively, the values of the τ_{CPU} can be retrieved from the array sequentially, which allows to "replay" the execution trace of the actual program.

The CPU/Scheduler block simulates the CPU with the given number of cores and the round-robin OS thread scheduler with equal priority of all the threads. The detailed description of the algorithm used by the CPU/Scheduler block is described in the section 3.4.1.

At the high-level model shown at the figure 6.20 the CPU/Scheduler central block is represented by the `cpuScheduler` block. Thread models depicted at the Figure 6.20 contain few computation

blocks that call `cpuScheduler`, e.g. `prepareRequest`, `prepareStat`, and `processFileBlock`.

Simulating Disk I/O

To simulate delays caused by disk I/O operations the model uses a combination of the DiskIO-Operation caller block and the DiskIOModule central block. DiskIOModule simulates the disk I/O subsystem of a computer and also measures disk utilization.

DiskIOOperation represents a disk I/O fragment. When the DiskIOOperation block receives the CFM, it retrieves the number k and parameters of the low-level I/O messages $\{dio_1, ..., dio_k\}$ from the distribution $\mathbb{P}^{\Pi}_{disk}$. In the case of the cache hit the $k = 0$, and the DiskIOOperation immediately sends the CFM to the next block. Otherwise the DiskIOOperation sends k *disk I/O messages* to the DiskIOModule block. Each message represents a low-level I/O request $dio_i, i \in (1, ..., k)$. These messages are sent to the DiskIOModule sequentially. If the message represents a synchronous operation, the DiskIOOperation waits for its completion before sending the next disk I/O message. Otherwise it sends the next disk I/O message to the DiskIOModule immediately.

The DiskIOModule combines models of the I/O scheduler and the hard drive. The detailed description of the algorithm used by the CPU/Scheduler block was described in the section 3.4.2.

The high-level model shown at the figure 6.20 contains the `diskIOModule` DiskIOModule central block. The model of the working thread depicted at the Figure 6.20 contains two DiskIOOperation blocks: the `doStat` that simulates the metadata read operation, and the `doFileRead` that simulates the regular read operation.

Simulating limits imposed by the OS

OS can impose a variety of limits on resources available to the program. This includes the maximum number of open file descriptors available for a thread, the maximum amount of memory available, or the CPU time. In certain scenarios these limits can severely affect the program's behavior and can be viewed as additional parameters of the system.

For example, an overloaded web server may exhaust the limit of the available file descriptors. In this case the server will cancel the processing the request immediately and return the error message to the user. In comparison to the requests that have been processed successfully, canceled requests will have significantly lower response time. However, failed requests may become a severe violation of the service agreement. This example demonstrates that detection of configurations that lead to violating of the OS limits can be an important problem.

We model the resource limits imposed by the OS using a combination of a central block and caller blocks. A code fragment that attempts to acquire a resource is represented by the OSCallMayFail caller block. When the CFM arrives to the caller block, the block calls an OSLimits central block and requests to allocate an instance of corresponding resource.

Upon receiving the request from OSCallMayFail the OSLimits updates usage information for a particular resource and notifies the caller if the resource was granted or not. In case if the resource cannot be allocated the caller may perform following actions:

- stop the simulation after a certain number of resource request fails;
- reroute the CFM message to a different destination block;

- log the fact of the resource exhaustion for the future analysis.

It is important to notify the OSLimits when the program no longer uses the particular resource. We use the OSCallRegular block for this purpose. This block calls the OSLimits central block when the program frees the resource.

4.2.4 Collection of performance metrics

In order to be useful the simulation framework must collect and output predicted values of performance metrics. Although OMNET++ offers some capabilities for recording results of simulation, exporting these results for further analysis (e.g. with MATLAB) proved to be troublesome. This motivated us to develop our own tools for collecting and recording performance metrics from the model of the program. In particular, PERSIK records following performance metrics:

- performance metrics for individual model blocks;
- performance metrics for groups of model blocks;
- performance metrics for requests;
- utilization of computational resources;
- the number of requests in the model's queues.

Values of performance metrics are recorded in text files in the .csv format. Each metric is recorded in the separate .csv file, which facilitates further analysis of simulation results.

During its execution the model can generate the vast amounts of information, which can consume a significant amount of storage space and can noticeably slow down the execution of the model. Thus it is essential to collect only those performance metrics that are necessary for performance analysis. Selecting the performance metrics that must be collected can be done through the configuration file of the model.

Collecting performance for individual model blocks

The only performance metric we collect for individual model blocks is its execution time (the amount of time required for the computer system being simulated to execute that block). This value of the metric is calculated as a difference between the receival of the CFM and its transmission to the next block. Execution time is collected for every execution of the block, and the block writes this data into the .csv file. The name of the .csv file and path to that file are specified with parameters of the block. Predicting the block execution time is an important tool for detecting bottlenecks in the system.

Collecting performance for groups of model blocks

PERSIK allows collecting execution time for groups of the blocks that are executed sequentially by a certain thread. The time is calculated as a difference between the start of the execution of the first block in the sequence and the end of execution of the last block. This, for example, allows to measure the amount of time require for the thread to process the request.

Measuring the execution time is done by setting and reading the timestamp to the CFM. Multiple timestamps can be set for the same CFM, where each timestamp is identified by a separate numeric identifier (ID). This allows measuring execution times for different groups of blocks in the same thread.

Setting and reading timestamps requires inserting additional blocks into the model. In particular, timestamps are set by the `setTimer` model block and read by the `readTimer` block. Parameters of the `setTimer` are the ID of the timestamp that must be set. Parameters of the `readTimer` are the ID of the timestamp that should be read and the name of the .csv file where measurements should be written.

The groups of blocks for which the execution time should be collected may not necessary form the straight sequence of nodes in the program's probabilistic call graph (PCG). This sequence of blocks may contain loops and/or branches. For example, the sequence may have the same starting block but multiple ending blocks, thus representing a tree-like structure in the program's PCG. An example of such sequence, which simulates processing of the request by the Tornado web server, is depicted at the Figure 6.20 (left). After reading the task from the queue (simulated by the block `fetchRequest`) the control flow can follow two distinct routes: the first one one leads to the block `sockClose_NoFile` (simulates the request to the non-existing document); and the second one leading to the block `sockClose_Success` (the correct HTTP request).

To record performance for the group of blocks with different exit nodes, separate `readTimer` blocks may be inserted before each exit node. Similarly, if the group has multiple starting nodes, separate `setTimer` blocks may be inserted before each starting node. It is essential that all these blocks use the same timestamp ID.

Collecting performance for requests

Performance metrics for requests include the response time R, which is the amount of time required for the system to process the request, and the throughput T, which is the number of requests processed in a time unit.

As the request is being processed, it travels through the boundaries of the working threads. Namely, one thread may receive the request, create a task object that represents the request in the terms of a program, and place it in a queue. Another thread reads the task from the queue and processes it. As a result, collecting performance metrics for requests is more challenging then collecting performance for an individual block or for a group of blocks.

To collect performance information for individual requests PERSIK relies on a combination of reader/writer blocks and timestamps. As the reader block reads the request from one queue, it attaches the request message to the CFM. Subsequently, the writer blocks detaches the previously attached request message from the CFM and sends it to another queue. The `setTimer` block can timestamp the request message, attached to the CFM. Correspondingly, the `readTimer` can read the timestamp from the attached request message, and write it into the .csv file.

For an example, consider the Tornado model at the Figure 6.20. The block `accept` in the accept thread simulates listening for the network connection. It fetches the request message from

the `connectionQueue` in the upper-level model (see Figure 6.19) and attaches it to the CFM that belongs to the model of the accept thread. The `startTimer_requestStart` block timestamps the attached request message. The `putRequest` outputs the timestamped request message to the task queue of the web server (represented by the `taskQueue` in the upper model). The `fetchRequest` block of the Tornado working thread fetches that message from the queue, and attaches it to the CFM that belongs to the model of the working thread. As the working thread finishes processing the request, one of the blocks `stopTimer_Success` or `stopTimer_NoFile` reads the timestamp of the request message and outputs it to the file.

The timestamp of the request message explicitly represents the response time R. By dividing the number of processed requests by the time difference between the processing of the first and last requests one may calculate the throughput T of the program.

Utilization of computational resources

Resource utilization is one of the most important performance metrics of a system. For certain applications, like scientific computing programs, high resource utilization is desirable in order to use the hardware efficiently. At the same time, a full utilization of a certain resource by the server may be the sign of reaching the state of saturation, where further increase in the workload intensity will result in dramatic increases in the response time.

PERSIK collects utilization of the CPU and the hard drive. This data is collected by CPUScheduler and DiskIOModule central blocks, which simulate the CPU and hard drive correspondingly. Hard drive utilization is reported after each disk I/O operation. CPU operations are normally much shorter and frequent, thus CPU utilization is reported every second.

We do not currently collect the level of contention for the program's locks (the ratio between the time when the lock is contended and the total running time). This functionality may be easily implemented in future versions of the PERSIK framework.

The number of requests in the program's queues

Length of queues is another important metrics of the queuing system's performance. In PERSIK each queue block reports the length of the associated queue. The length of the queue is reported after each read or write operation.

Table 4.1: Properties common for all PERSIK blocks

Property name	**Description**
`string statisticsDir`	the path to the directory, where statistics for this block will be collected
`string statisticsEnabled`	a comma-separated list of statistics that will be collected for this block. Each block defines its own set of statistics that can be collected

Table 4.2: PERSIK high-level model blocks and their properties

 CPUScheduler: the model of the OS thread scheduler and the CPU

Property name	Description
`int CPUCount`	the number of the CPU cores
`double quantumLength`	the distribution of time quantas provided by the scheduler for each thread. Not used if quantumLength_FromSample parameter is specified
`string quantumLength_FromSample`	the name of the file, which contains the sample of the time quantas
`string quantumLength_SampleType`	the type of the time quantas sample. "random": the time quantas will be sampled from the sample randomly. "trace": the quantas will be retrieved sequentially

 DiskIOModule: the model of the I/O subsystem. Includes models of the OS I/O scheduler and the hard drive

`string pathToSample`	specifies the path to the file that contains the distribution $P(\tau_{disk}\|dio\_type, dio\_rate, dio\_parallel)$

 OSLimits: simulates limitations imposed by the OS, such as the maximum number of open files or the maximum number of network connections

`int ulimitDescriptors`	the maximum number of open file descriptors
`int ulimitSockets`	the maximum number of open network connections
`int stopAfterFailureCt`	the maximum number of failed OS calls after which the simulation will be stopped
`double stopAfterFailurePerc`	the percentage of the failed OS calls after which the simulation will be stopped

 SyncBarrier: the model of a barrier

`int capacity`	the capacity of the barrier. The barrier will be lifted once the number of waiting threads reaches the capacity

 CriticalSection: the model of a mutex

No parameters

 PassiveQueue: represents a queue or a buffer in the program

`int capacity`	the maximum capacity of the queue. -1 denotes an infinite capacity
`bool fifo`	if true, the queue will be a FIFO (first-in-first-out) queue. Otherwise the queue will be a LIFO queue
`bool directAccess`	if true, the reader can access the queue by the means of a direct call. Otherwise the requests will be fetched through the in and out gates

 Source: generates requests

`int numJobs`	the total number of requests to be generated during the course of simulation
`volatile double interArrivalTime`	the distribution of interarrival times for requests. Not used if interArrivalTime_FromSample is defined
`string interArrivalTime_FromSample`	the name of the file, which contains the sample of interarrival times
`string interArrivalTime_SampleType`	the type of the interarrival times sample. Allowed values: "random" or "trace"
`string interArrivalTime_ColumnNo`	the column number in the interArrivalTime_SampleType file that specifies the interarrival time
`double stopTime`	the time when the module will send out its last request. -1 denotes no limit on time

 Sink: used to absorb the requests that depart the model

No parameters	

Table 4.3: PERSIK thread model blocks and their properties

 SourceOnce: generates a single CFM for a thread upon the start of the model

Property name	**Description**

`int TID`	the ID of the thread for which the CFM is generated

 DispatchBlock: probabilistically reroutes the CFM in the thread model from a single input gate to one of the output gates

`string probFileName`	the full path to the file that denotes probabilities of transition. Each line in the file contains a probablity of routing the CFM to the corresponding gate

 Loop: executes the given blocks (a loop body) in a loop

`volatile int iterationCount`	specifies the distribution of iterations counts for the loop
`string iterationCount_FromSample`	the full path to the file that provides a sample of the iterations count
`int iterationCount_ColumnNo`	the column number in the iterationCount_FromSample file that specifies the iterations count
`string iterationCount_SampleType`	the type of the sample specified in the iterationCount_FromSample. Allowed values: "random" or "trace"

 Delay: simulates a time delay in the thread

`volatile double delayTime`	explicitly specifies the distribution of delay times. Not used if delayTime_FromSample parameter is specified
`string delayTime_FromSample`	the full path to the file that provides the sample of delay times for this block
`int delayTime_ColumnNo`	the column number in the delayTime_FromSample file that specifies the delay times sample
`string delayTime_SampleType`	the type of the sample specified in the delayTime_FromSample. Allowed values: "random" or "trace"

 ComputBlock: simulates CPU-bound computations. Corresponds to a computation CF in a formal thread model

`string CPUBlockName`	the full OMNET++ name of the CPUScheduler block that will be called by this block
`volatile double computationTime`	explicitly specifies the distribution of CPU times for this block. Not used if computationTime_FromSample parameter is specified

`string computationTime_FromSample`	the full path to the file that provides the sample of CPU times for this block
`int computationTime_ColumnNo`	the column number in the computationTime_FromSample file that specifies the CPU time sample
`string computationTime_SampleType`	the type of the CPU time sample specified in the computationTime_FromSample. Allowed values: "random" or "trace"

 DiskIOBlock: simulates a single elementary I/O operation. I/O operation can be either read, metadata read, or readahead

`string diskIOModulePath`	the full OMNET++ name of the DiskIOModule block, which implements the model of disk I/O subsystem and will be called by this block

 DiskIOOperation: simulates an I/O CF in the thread. A single execution of DiskIOOperation can initiate an arbitrary number of read, metadata read, or readahead operations. This is a complex block that contains three DiskIOBlock basic blocks and three Loop blocks. Each DiskIOBlock simulates read, metadata read, or readahead operation; while the corresponding Loop block allows repeating this operation a given number of times

`string diskIOModuleBlockPath`	the full OMNET++ name of the DiskIOModule block, which implements the model of disk I/O subsystem and will be called by this block
`volatile int iterationCountRead`	the distribution of the number of read I/O operations to be performed by this block. Not used if iterationCountRead_FromSample parameter is defined
`string iterationCountRead_FromSample`	the full path to the file that provides the sample for the number of read I/O operations
`int iterationCountRead_ColumnNo`	the column number in the iterationCountRead_FromSample file that specifies the number of read I/O operations
`volatile int dataAmountReadOp`	the distribution of data amounts to be transferred by the read I/O operation. Not used if dataAmountReadOp_FromSample parameter is defined
`string dataAmountReadOp_FromSample`	the full path to the file that provides the sample for the amount of data transferred by the read I/O operations
`int dataAmountReadOp_ColumnNo`	the column number in the dataAmountReadOp_FromSample file that specifies the sample of data amounts

`volatile int iterationCountRA`
`string dataAmountRAOp_FromSample`
`int iterationCountRA_ColumnNo`
`volatile int dataAmountRAOp`
`string dataAmountReadOp_FromSample`
`int dataAmountReadOp_ColumnNo`
	parameters that denote the number and properties of readahead operations. The semantics of these parameters is same as for parameters of read operations

volatile int iterationCountMetadata string iterationCountMetadata_FromSample int iterationCountMetadata_ColumnNo volatile int dataAmountMetadataOp string dataAmountMetadataOp_FromSample int dataAmountMetadataOp_ColumnNo	parameters that denote the number and properties of readahead operations. The semantics of these parameters is same as for parameters of read operations

 OSCallMayFail: simulates an OS call that may fail, such as opening the file

bool rerouteOnFailure	if true, than the CFM will be rerouted to the "outFailure" output gate in the case of a call failure. Otherwise the CFM will be rerouted to the "out" gate
string OSBlockName	the full OMNET++ name of the OSLimits block, which simulates limitations imposed by the OS
string OSCallName	the full OMNET++ name of the system call simulated by this block

 OSCallRegular: simulates an OS call that do not fail, such as closing the file

string OSBlockName	the full OMNET++ name of the OSLimits block, which simulates limitations imposed by the OS
string OSCallName	the full OMNET++ name of the system call simulated by this block

 CritSection_enter: simulates a "mutex.enter" operation

string syncServerName	the full OMNET++ name of the server block, which implements the model of the mutex itself and will be called by this block

 CritSection_leave: simulates a "mutex.exit" operation

string syncServerName	the full OMNET++ name of the server block, which implements the model of the mutex itself and will be called by this block

 SyncBarrier_await: simulates a "barrier.await" operation

`string syncServerName`	the full OMNET++ name of the server block, which implements the model of the mutex itself and will be called by this block

 QueueReaderSimple: simulates fetching the request from the queue (a simple model). The block simulates only fetching the request from the queue; computations pertaining to this operation are not simulated. QueueReaderSimple incurs less simulation overhead, thus we recommend using it instead of QueueReaderDetailed unless a significant drop in accuracy is observed

`string queueNames`	full OMNET++ names of the input queues from which the request can be fetched
`string fetchingAlgorithmIn`	denotes the algorithm used to select a queue from which the request will be fetched. This parameter is used only if there are multiple queues that contain requests. Allowed values are "priority", "random", "roundRobin", and "longestQueue"
`bool doBlockIfQueueEmpty`	if this property is set to true, then the reader will delay the propagation of the CFM if all the input queues contains no requests. The reader will wait until some requests will appear in the queue or until the timeout will pass (whatever happens first). If the block will be able to fetch the request before the timeout expires, then the CFM will be rerouted to the outFetched gate. Otherwise the CFM will be rerouted to the outNotFetched gate
`string waitTime_FromSample`	the full path to the file that provides the sample of timeout
`string waitTime_SampleType`	the type of the sample specified by the waitTime_FromSample. Allowed values: "random" or "trace"
`bool doKeepJob`	if true, then the fetched request will be attached to the CFM. Otherwise the request will be discarded

 QueueReaderDetailed: simulates fetching the request from the queue (a detailed model). Fetching the request involves locking the queue mutex, performing CPU computations associated with the retrieval operation, releasing the mutex, and fetching a request from the queue. QueueReaderDetailed block allows for more accurate modeling of queue operations, but incurs a significant overhead during the simulation. Thus it is recommended to use QueueReaderSimple block to simulate queuing operations whenever possible. QueueReaderDetailed should be used only if QueueReaderSimple results in an unacceptable decrease in the prediction accuracy. Parameters of this block are a the superset of parameters of the QueueReaderDetailed block

`string queueNames`	see QueueReaderSimple
`string fetchingAlgorithmIn`	see QueueReaderSimple
`bool doBlockIfQueueEmpty`	see QueueReaderSimple

`string waitTime_FromSample`	see QueueReaderSimple
`string waitTime_SampleType`	see QueueReaderSimple
`bool doKeepJob`	see QueueReaderSimple
`string lockName`	the full OMNET++ name of the CritSection block associated with the queue
`string _CPUBlockName`	the full OMNET++ name of the CPUScheduler block that will be used to simulate computations
`volatile double _computationTime`	specifies the distribution of CPU times required to fetch the request from the queue. Not used if _computationTime_FromSample parameter is specified
`string _computationTime_FromSample`	the full path to the file that provides a sample of CPU time
`int _computationTime_ColumnNo`	the column number in the _computationTime_FromSample file that specifies the CPU time
`string _computationTime_SampleType`	the type of the sample specified by the _computationTime_FromSample. Allowed values: "random" or "trace"

 QueueWriterSimple: simulates sending a request to the queue (a simple model). The block simulates only sending the request to the queue; computations pertaining to this operation are not simulated. QueueWriterSimple incurs less simulation overhead, thus we recommend using it instead of QueueWriterDetailed unless a significant drop in accuracy is observed

`string queueNames`	full OMNET++ names of the queues to which the request can be sent
`string sendingAlgorithm`	denotes the algorithm used to decide to which queue the request should be sent to. Allowed values are "priority", "random", "roundRobin", "shortestQueue", and "longestQueue"
`bool doGenerateJob`	if this property is set to true, then the writer will generate a new job and send it to the queue. Otherwise it will send the job that was attached to the CFM

 QueueWriterDetailed: simulates sending a request to the queue (a detailed model). Sending the request involves locking the queue mutex, performing CPU comutations that represent a sending operation, releasing the mutex, and sending the request to the queue. Similarly to QueueReaderDetailed, this block should be used only if QueueWriterSimple block results in a significant decrease in the model accuracy. Parameters of this block are a the superset of parameters of the QueueWriterSimple block

`string queueNames`	see QueueWriterSimple
`string sendingAlgorithm`	see QueueWriterSimple

bool doGenerateJob	see QueueWriterSimple
string lockName	the full OMNET++ name of the CritSection block associated with the queue
string jobName	the name of the newly created request. This parameter is used only if doGenerateJob is set to true
string _CPUBlockName	the full OMNET++ name of the CPUScheduler block that will be used to simulate computations
volatile double _computationTime	specifies the distribution of CPU times required to fetch the request from the queue. Not used if _computationTime_FromSample parameter is specified
string _computationTime_FromSample	the full path to the file that provides a sample of CPU time
int _computationTime_ColumnNo	the column number in the _computationTime_FromSample file that specifies the CPU time
string _computationTime_SampleType	the type of the sample specified by the _computationTime_FromSample. Allowed values: "random" or "trace"

 Socket_accept: simulates accepting a network connection. The block fetches a request that represents an incoming connection from the network connection queue, and then attempts to open the file descriptor for it. If the open operation was successfull, the CFM will be rerouted to the outSuccess gate. Otherwise the request may be sent back to a network connetion queue, and the CFM will be rerouted to the outFailure gate. Socket_accept is a complex block that includes the QueueReaderSimple, OSCallMayFail, and QueueWriterSimple blocks

string connectionQueueName	the full OMNET++ name of the network connection queue
string OSBlockName	the full OMNET++ name of the OSLimits block
bool rerouteOnFailure	if this parameter is set to true, then upon a failed OS call the connection request will be sent back to the network connection queue, and the CFM will be rerouted to the outFailure gate

 SetTimer: the block used to timestamp a CFM or a request message

int timerIdx	the index of the timer used to timestamp the message
bool stampInternalJob	if set to true, then the request message attached to the CFM will be timestamped. Otherwise the CFM itself will be timestamped

 ReadTimer: reads the timestamp from the CFM or from the request message and writes the value of the timestamp to the log

`int timerIdx`	index of the timer used to timestamp the message
`bool readInternalJob`	if set to true, then the timestamp will be read from the request message attached to the CFM. Otherwise the timestamp of the CFM itself will be read

 Stopper: stops the simulation after receiving a given number of CFMs

`int stepsBeforeStop`	the number of CFMs after which the simulation will be stopped

4.3 Verification of performance models

To test our approach for performance prediction we manually built models of two multithreaded programs. These programs are different in their purpose, architecture, behavior, and programming languages and thus can be representative for a larger class of applications. The first program is Galaxy, a CPU-bound scientific computing application implemented in a Java programming language. The second program is tinyhttpd, a disk I/O-bound web server implemented in C language. These programs used `java.util.concurrent` and `pthreads` frameworks for implementing parallel programming correspondingly. If necessary, PERSIK can be extended to support different frameworks for parallel programming, such as MPI.

Manually building models of even small and relatively simple multithreaded programs proved to be time-consuming and error-prone activity. However, at this point we were more interesting at verifying the overall validity of our approach to performance prediction and the accuracy of the resulting models.

We utilized a mixed approach towards building models of these programs. We manually analyzed the program at the high level to establish its structure and semantics. The manual analysis of the program involved following steps:

- studying the code of the program in order to understand its structure and behavior;
- identification of threads $tr_1, ..., tr_m$, thread pools $Tp_1, ..., Tp_k$, and queues $q_1, ..., q_n$ in the program and semantics of their interaction. This allows to determine the general sequence of operations that happen during the request processing and provides information necessary for constructing the high-level queuing model of the program;
- identifying locks L in the program and determining their parameters Π_{lock};
- discovering code fragments $s_1 ... s_n$ in the program's source and determining their classes;

- instrumenting the program by inserting probes at the borders of individual code fragments. Instrumentation is done at the source level. The instrumentation library is statically linked to the program and then probes, which are implemented as calls to the library functions, are inserted into the program's source code.
- creating the program's schema. The schema includes the list of all the CFs of the program and corresponding probe IDs. The creation of schema is necessary for the upcoming automated analysis of the program.

The instrumented program is executed in a single configuration, where its behavior and demand for computational resources are representative for a larger number of configurations. The instrumentation generates an execution trace for the program, which is analyzed automatically. The automatic analysis of the program's trace yields following information:

- the probabilistic call graphs for the program's threads;
- the CPU times $\mathbb{P}^{\Pi}_{CPU}$ for each computation CF;
- the total amount of time required to execute each code fragment (wallclock time) and the request processing time R. These metrics are used for model debugging and analysis of simulation results.

Data obtained from the instrumented program do not include information on I/O operations initiated by the program. Unfortunately, this information cannot be completed in the user mode. To collect data on I/O operations we instrumented the Linux I/O scheduler using the SystemTap framework [6]. The resulting log of low-level kernel I/O operations allowed us to obtain parameters $\mathbb{P}^{\Pi}_{IO}$ of I/O code fragments.

Once all the necessary information was collected, we manually created PERSIK models of the test programs.

In order to be useful, the model must predict performance accurately, and our primary concern was the accuracy of prediction. To estimate accuracy of the model we ran the program in different configurations and recorded actual performance of the program for each configuration. Afterwards we simulated the program in the same configurations and recorded predicted performance. To obtain reliable performance measurements we performed multiple runs of both the actual program and its model in each configuration and used the mean values of measured and predicted performance metrics.

Finally, we calculated relative error ε between mean values of measured and predicted performance metrics as

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

The higher is the relative error the worse is the accuracy of prediction. For the ideal model that predicts the program's performance without any errors $\varepsilon \to 0$.

All our experiments were conducted on a PC equipped with an Intel Q6600 quad-core 2.4 GHz CPU, 4 GB RAM and 160 GB hard drive running under Ubuntu Linux 10.04 OS.

4.3.1 Galaxy: the n-body simulator

Galaxy is a simple Java scientific computing application that simulates the gravitational interaction of multiple celestial bodies. Although mostly used as an educational example, this program employs a variety of synchronization techniques and is a good representative of a multithreaded scientific application.

Galaxy uses a conventional approach to the problem of n-bodies simulation. It discretizes time into small steps and calculates movement of objects during the each such step. To achieve good performance, the Galaxy implements the Barnes-Hut [23] algorithm, which involves building an octree. A single iteration of the Galaxy algorithm involves three major actions in a strict order: calculating forces acting on bodies and updating bodies' positions; rebuilding the octree; and checking bodies for collisions. The length of each iteration can be viewed as a response time R and thus represents the most important performance metric of the Galaxy.

Galaxy uses multiple thread pools to speed up computations. The first thread pool ("force threads") calculates forces and updates positions of the bodies, while the second thread pool ("collision threads") detects body collisions. Thread pools communicate through two synchronized queues. The ordering of operations is enforced by the main thread of the program, which uses barriers to synchronize threads in thread pools. The main thread is also responsible for rebuilding the octree.

Both high-level and thread models of the Galaxy were built manually. Below we describe the structure and semantics of these models in detail.

The high-level model of Galaxy in a configuration with one force thread and one collision thread is shown at the Figure 4.3. Upon model initialization the task are created by the `fillBodies` block, which sends them to the `positionsQueue`. `positionsQueue` and `forcesQueue` are queue blocks that represent synchronized queues in the program. `galaxy_forcesthread1` and `galaxy_collisionthread1` blocks represent the force thread and the collision thread, while the `galaxy_mainthread` block represents the main thread of the program.

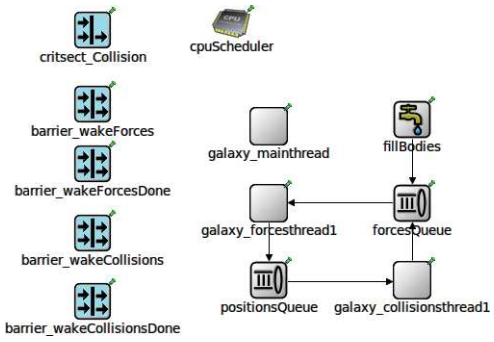

Figure 4.3: A manually constructed model of the Galaxy (high-level)

Low-level thread models for the Galaxy threads are shown at the Figure 4.4. The model of the main thread contains four blocks that call corresponding barrier blocks of the high-level model. `wakeForces_await` and `wakeForcesDone_await` blocks are used to wake up/suspend force threads,

while `wakeCollisions_await` and `wakeCollisionsDone_await` do same for the collision threads. The `octreeBuild` computation block simulates rebuilding the octree.

The model of the force thread uses the `queueForcesReader` reader block to fetch tasks from the `forcesQueue`. If the queue is empty, the control is immediately transferred to the `readTimer` block, and force thread finishes its work for the current iteration of Galaxy. Otherwise, the control is transferred to the the `dispatchIfDeleted` dispatch block, which simulates a check if the body has been marked as collided with another body. If the collision has occured, the corresponding task is deleted and the CFM is sent back to the `queueForcesReader`. Otherwise the CFM is sent to the `calcForcesPositions` computation block that simulates calcuation of the net force acting on the body. Next, the model outputs the task to the `positionsQueue` using the `queueCollisionsWriter` writer block and attempts to fetch a new task from the `forcesQueue`. Once all the requests in the `forcesQueue` have been processed, the thread uses the `wakeForcesDone_await` caller block to notify the main thread. Finally, `wakeForces_await` block suspends the thread until it is waken up by the the central thread during the next iteration.

The model of the collision thread appears to be the most complex of all the thread models since it performs a lot of "housekeeping" operations in Galaxy. The overall execution flow of the collision thread is similar to one of the forces thread. The `queueCollisionsReader` reader block fetches tasks from the `positionsQueue`. If the queue is non-empty, the task representing the body is fetched. The Galaxy verifies if the body has escaped out of the area of space that is being modeled (simulated by the `OctreeIntersects` block), and the `dispatchIfIntersects` block simulates checking on the result of previous computation. In if the body has escaped the corresponding task is deleted, and the CFM is transferred back to the `queueCollisionsReader` block. The `collide` block simulates the collision check itself. The sequence of `critSectionCollision_enter`, `processCollision`, and `critSectionCollision_exit` blocks simulates the processing of the collision by the thread, which involves merging the first of the collided objects with the second one, and marking the second object for deletion. Finally, the `queueForcesWriter` block outputs the task representing the object into the `forcesQueue`.

To define parameters for the low-level thread models we instrumented the Galaxy code with 29 probes and ran it in the configuration with 2 force threads and 2 collision threads. In these experiments the probability of collision was $1.01 * 10^{-5}$, the mean value of $\overline{\tau}_{CPU}$ (`octreeBuild`) = $5.40 * 10^{-3}$ sec, and $\overline{\tau}_{CPU}$ (`calcForcesPositions`) = $4.33 * 10^{-5}$ sec.

We used the model to predict the iteration length of the Galaxy in each configuration. The comparison of actual and predicted iteration lengths is shown at the Figure 4.5.

The relative error for the iteration length varies in $\varepsilon \in (0.002, 0.179)$ depending on the program configuration. The average error measured across all the configurations is $\overline{\varepsilon} = 0.073$, which is comparable to statistical prediction models [64], [37]. Relative errors for all the configurations are listed in the Table 4.4.

These results convince us that the model predicts iteration length of Galaxy with reasonable accuracy. Furthermore, the model locates those configurations that result in the high performance

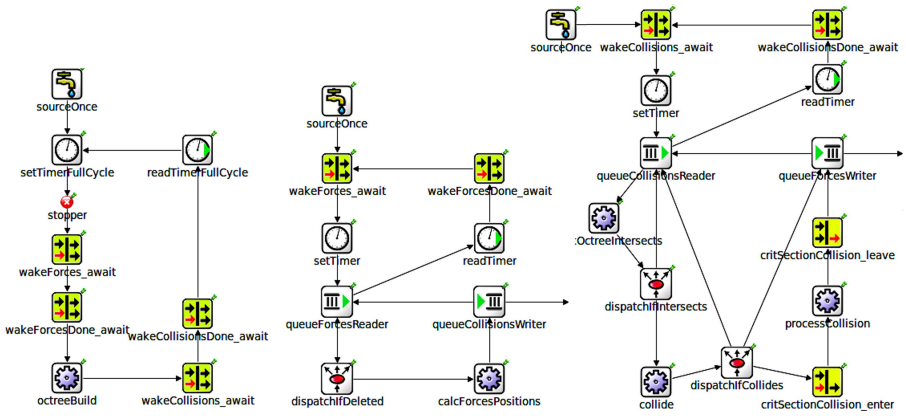

Figure 4.4: Manually constructed models of Galaxy threads. Left: the model of the main thread. Center: the model of the force thread. Right: the model of the collision thread.

Table 4.4: Relative errors for predicting the Galaxy iteration length

Num. collision threads	The number of force threads						
	1	2	4	6	8	12	16
1	0.054	0.161	0.084	0.067	0.110	0.074	0.111
2	0.155	0.102	0.018	0.007	0.038	0.028	0.005
4	0.179	0.132	0.086	0.072	0.056	0.048	0.076
6	0.151	0.115	0.090	0.067	0.042	0.059	0.054
8	0.143	0.126	0.075	0.051	0.050	0.036	0.062
12	0.174	0.122	0.069	0.057	0.036	0.058	0.054
16	0.033	0.105	0.052	0.014	0.053	0.069	0.070

of the program. In particular, it correctly points that the number of force processing threads must be $>= 4$ (which equals to the number of available CPU cores), while the number of collision threads has no significant impact on Galaxy performance.

Table 4.5 provides the predicted CPU utilization values for the Galaxy on the test system *averaged over the whole run* (value of 100% denotes a full utilization of a single CPU core). Note that *on average* Galaxy never fully utilizes all four CPU cores. Although force calculations and collision detections are prefectly parallelizable and can utilize all the CPU cores, rebuilding the octree is not a parallelizable operation. It is executed only by a main thread, which can use only a single CPU core at a time. Information on CPU utilization can be used to improve the Galaxy algorithm and further tune configuration options of the program.

4.3.2 tinyhttpd: the web server

Predicting performance of the web server is a more complex task since it involves simulating not only computations, but also I/O operations. To verify the applicability of our approach to modeling the performance of disk-heavy server applications we built the model of a tinyhttpd multithreaded

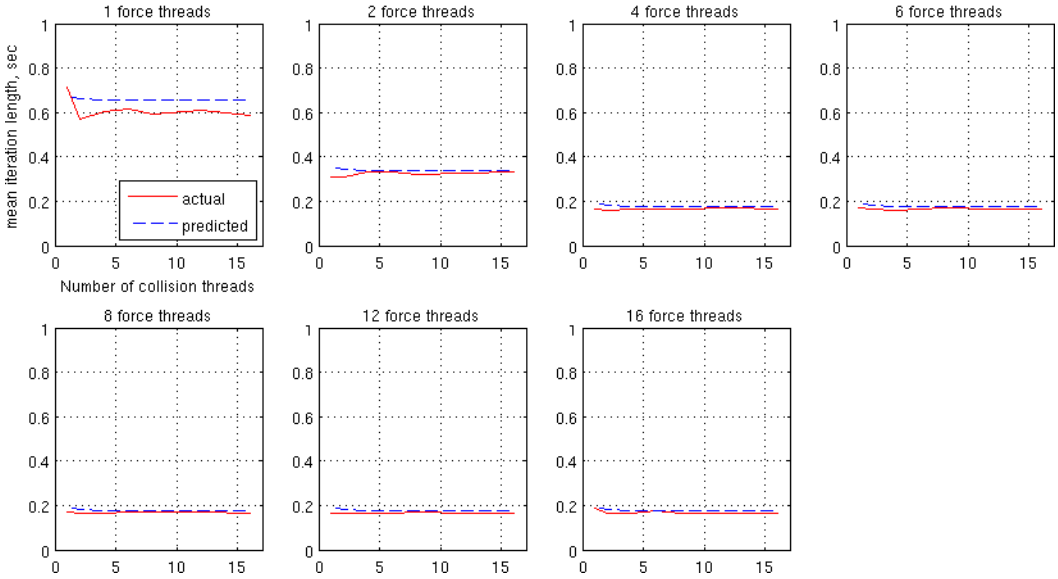

Figure 4.5: Experimental results for the Galaxy

Table 4.5: Predicted average CPU utilization for the Galaxy, %

Num. collision threads	The number of force threads						
	1	2	4	6	8	12	16
1	100.0	102.1	103.1	103.1	103.1	103.1	103.1
2	189.5	197.0	201.0	201.0	201.0	201.0	201.0
4	342.9	368.3	382.4	382.4	382.6	382.4	382.5
6	343.0	368.2	382.5	382.4	382.5	382.4	382.5
8	342.9	368.3	382.4	382.4	382.4	382.4	382.4
12	342.7	368.2	382.3	382.2	382.3	382.1	382.2
16	342.6	367.9	382.1	382.1	382.0	382.1	382.1

web server [7]. tinyhttpd is written on C programming language. It is simple and compact, which facilitates its analysis, but at the same time it is representative for a larger class of server applications.

tinyhttpd relies on a pool of working threads to process HTTP requests. When the tinyhttpd receives an incoming request, it creates the corresponding task and puts that task into the queue until one of its working threads becomes available. The working thread then picks the task from the queue, retrieves the local path to the requested file, and verifies its existence using a `stat()` function. If the file exists, the thread opens it for reading. If the file was opened successfully, the thread reads the file in 1024-bytes chunks and sends them to the client. Otherwise the server deletes the task and sends the "Internal Server Error" response to the client (in this case it is said that the request is "dropped"). Once data transfer is complete, the working thread closes the connection and picks up the next incoming task from the queue.

In our experiments we used the tinyhttpd to host 200000 static web pages from the Wikipedia

archive. According to the common practice, the `atime` filesystem functionality that logs time of every access of the particular file was disabled to improve the performance of the server.

We relied on the modified version of the http_load software [8] to simulate client connections to our web server. Our http_load reads a list of URLs from the file and then retrieves these pages from the web server with a given rate. httpd_load is running on a client computer (Intel 2.4 GHz dual-core CPU, 4 GB RAM, 250 GB HDD) connected to the server with a 100 MBit Ethernet LAN.

The main metric used to measure the performance of a web server was the response time R. The R is defined as a time difference between accepting the incoming connection and sending the response (more accurately – closing the communication socket). Additional performance metrics are the total throughput T and the number of error responses.

The configuration space of the web server includes two parameters: the incoming request rate (IRR) and the number of working threads of the web server. The IRR quantifies the workload of the server. In the industrial setting the IRR is adjusted using a load balancer – a separate computer that distributes requests between the web servers in the datacenter. Varying the IRR allows simulating the behavior of the web server under the different load. In our experiments we vary IRR from 10 requests per second (rps) to 130 rps with the step of 10 rps. The number of working threads is the only configuration parameter of the web server itself that affects its performance. We run the web server with 2, 4, 6 and 8 working threads.

As a result, the total number of different experimental configurations is 13*4=52, which includes all the possible combinations of the number of threads and incoming request rates. For each configuration we ran both the actual program and its model and record average values of performance metrics. During each run 10,000 requests were issued.

We manually instrumented the code of the tinyhttpd server with 21 probes and built the model of the server using the configuration with 4 threads and IRR=70 rps.

The comparison of predicted and actual results for tinyhttpd are presented at the Figure 4.6 and in the Table 4.6.

Depending on the values of IRR the web server has two distinct states of operation (see Figure 4.6). For IRR $\leq$ 50 rps the I/O subsystem is not fully utilized and the R is minimal ($R \in$ (10-20 ms)). IRR $\geq$ 60-70 rps result in the overload of the I/O subsystem. Processing the request takes longer time, and incoming connections start accumulating in the web server queue. As a result, the web server is brought to the point of the saturation, where it exceeds the OS-imposed limit of 1024 open file descriptors (remember, each connection requires an open file descriptor). The server is unable to open files on the disk for reading, and the number of error responses increases. At this point the R reaches 14-17 sec. and remains steady. The total throughput T, however, continues to grow as it does not distinguish between requests that fail or return successfully.

One interesting observation is that the number of working threads has a relatively small influence on R. This is explained by the fact that the performance of the web server is largely determined by the performance of the I/O system, and the I/O system (hard drive) can effectively carry out only a single I/O operation at a time. As a result, the increase in the number of parallel operations is

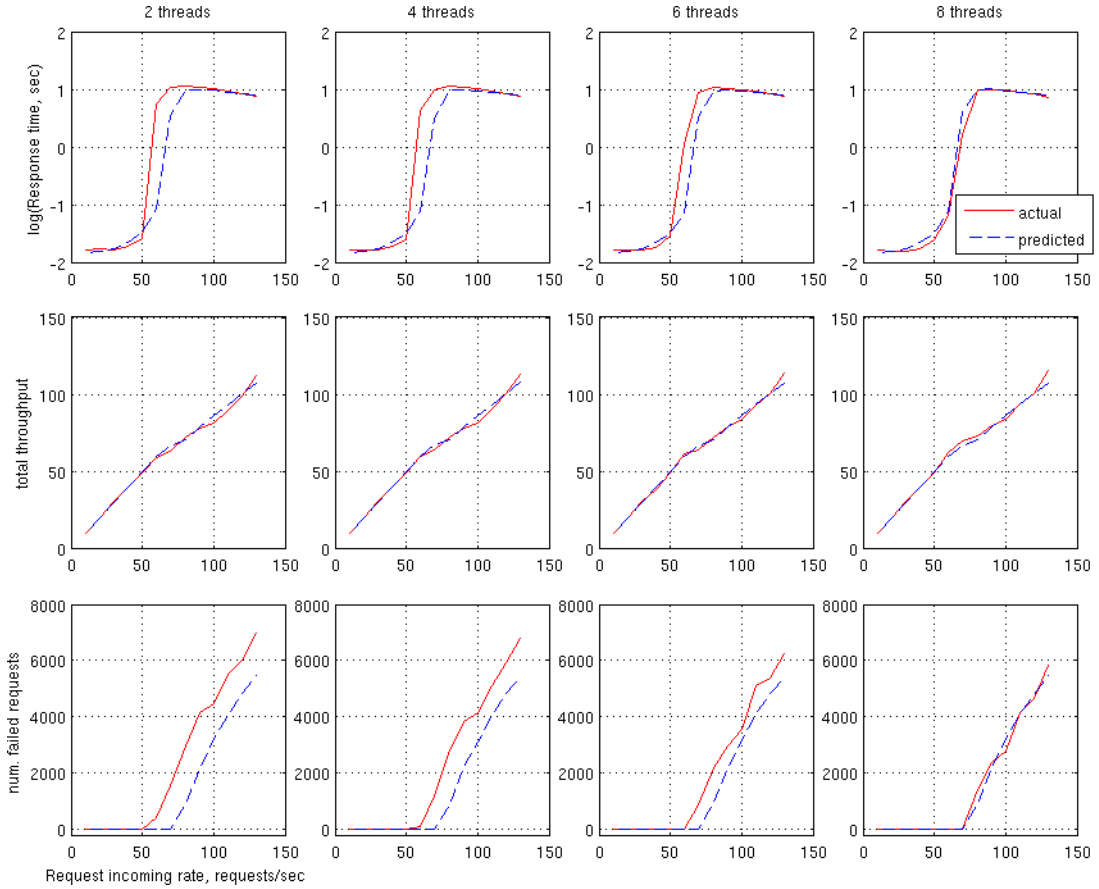

Figure 4.6: Experimental results for the tinyhttpd. top row: the response time R (logarithmic scale); middle row: the throughput T; bottom row: the number of error responses

negated by a proportional increase in the average execution time for each individual I/O operation.

Our model predicts the R for stationary states reasonably well, with $\varepsilon \leq 0.30$. However at transition point between the saturated and non-saturated behaviors the accuracy of the model decreases to $\varepsilon \in (0.093...1.555)$, see Table 4.6. But since the size of the transitional region is small, the average error across all the configurations $\overline{\varepsilon} = 0.203$. The total throughput T is predicted highly accurately ($\varepsilon \leq 0.021$), but in order to correctly interpret T, one has to take into account the number of error responses. The model predicts this metric with somewhat lower accuracy, which is comparable to the prediction error for R. The average error for predicting the number of error response $\overline{\varepsilon} = 0.214$, and at the transition point $\varepsilon = 1$. However, the number of failures in the transition region is small ($\leq$ 10% of all the requests), so even the slight variation in the actual number of failed requests significantly affects prediction accuracy.

The mean prediction errors reported by the the model of the tinyhttpd web server are higher than errors reported by the model of the Galaxy scientific computing application (0.21 vs. 0.073 correspondingly). Nevertheless, our model is applicable for solving many practical problems. In

Table 4.6: Relative errors for predicting the tinyhttpd performance metrics

Response time R

Num. threads	Incoming request rate												
	10	20	30	40	50	60	70	80	90	100	110	120	130
2	0.098	0.073	0.083	0.184	0.448	0.981	0.570	0.115	0.059	0.070	0.050	0.018	0.037
4	0.081	0.050	0.092	0.182	0.366	0.981	0.543	0.118	0.046	0.068	0.042	0.013	0.038
6	0.080	0.033	0.093	0.233	0.145	0.927	0.448	0.083	0.025	0.041	0.011	0.007	0.063
8	0.068	0.010	0.126	0.275	0.415	0.550	1.555	0.005	0.000	0.011	0.011	0.005	0.078

Total throughput (including error responses) T

Num. threads	Incoming request rate												
	10	20	30	40	50	60	70	80	90	100	110	120	130
2	0.003	0.003	0.015	0.004	0.022	0.013	0.051	0.016	0.009	0.061	0.043	0.018	0.039
4	0.004	0.003	0.016	0.004	0.022	0.003	0.051	0.020	0.002	0.056	0.034	0.005	0.045
6	0.004	0.003	0.016	0.034	0.020	0.034	0.042	0.023	0.002	0.032	0.002	0.002	0.057
8	0.004	0.003	0.015	0.003	0.023	0.040	0.045	0.023	0.008	0.031	0.004	0.000	0.071

Number of error responses

Num. threads	Incoming request rate												
	10	20	30	40	50	60	70	80	90	100	110	120	130
2	0.000	0.000	0.000	0.000	0.000	1.000	1.000	0.684	0.482	0.281	0.263	0.199	0.214
4	0.000	0.000	0.000	0.000	0.000	1.000	1.000	0.690	0.411	0.249	0.202	0.180	0.203
6	0.000	0.000	0.000	0.000	0.000	0.000	1.000	0.566	0.265	0.104	0.199	0.102	0.136
8	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.366	0.068	0.159	0.011	0.037	0.066

particular, it accurately predicts values of configuration parameters where the transitional behavior occurs. This result is important, since usually the goal of performance models is not just to predict performance of the program across all the possible configurations, but to find those configurations that result in high performance. Possible causes of prediction errors for models of I/O-bound applications will be discussed in more detail in the Section 6.

Chapter 5

Automatic Model Generation

In this section we describe the procedure for automatic generation of performance models of multithreaded programs. Initially we list the data required to build the model of the multithreaded program. Next we provide a description of static and dynamic program analysis which collects all the necessary data. We describe each stage of the analysis in detail. Finally, we discuss how this data is used to build the resulting model.

5.1 Data required for building a performance model

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

- The set of queues and buffers used to exchange tasks between different components of the program. These correspond to the queues in the high-level model (see Section 3.2);
- The set of threads in the program. Threads correspond to the service nodes of the high-level queuing model;
- The set of thread pools (see Section 3.2). Sizes of thread pools are configuration parameters that have major impact on the program's performance;
- Information on interactions between the threads and queues in the program. This corresponds to c_{in}/c_{out} CFs in the middle-tier 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). These are required to build PCGs of the program's threads;
- The classes and parameters Π of CFs, which are required to model delays τ;
- The set L of locks, their types, and parameters Π_{lock}. These are required to build the lower-level models of locks.

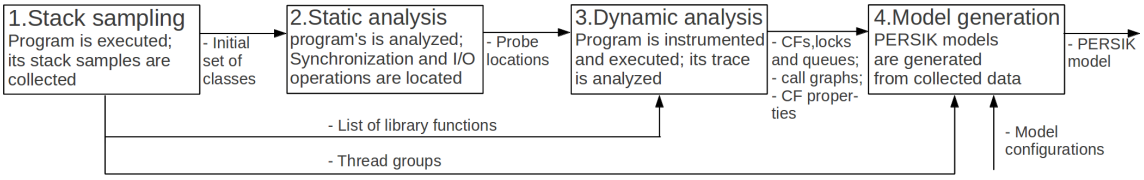

Figure 5.1: Model creation stages and intermediate results

We collect required data using a combination of static and dynamic analysis. With regard to the data collection, we have following assumptions about the program whose model we are building:

- The program is free of bugs. Namely, the program does not deadlock or crash during the data collection;
- There is no intensive paging that occur because of the shortage of the RAM;
- No other significant computation activity is going on the hardware where the program is being executed. Namely, no other program is performing CPU-intense or I/O intense computations in parallel with the program being studied.

During the data collection the program is executed in a single representative configuration. By representative we mean that $< S, \delta >$ and Π would be similar to the $< S, \delta >$ and Π of a larger set of configurations for which the program's performance should be predicted. Normally this requires the usage scenario for the program (e.g. the probabilities of accessing particular web pages for a web server or an input dataset for a scientific computing application) to remain similar across the configuration space.

We build the model in four stages (see Figure 5.1). Each stage saves intermediate results into the text and .xml files, which are inputs for the subsequent stages.

First, the program is executed in the representative configuration and its call stack is sampled. The stack samples are used to detect thread groups, program libraries, and to identify significant portions of the system.

Second, a static analysis of the program is performed. During this stage we detect synchronization, c_{in}, c_{out}, and I/O 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 event model. **Unless explicitly noted, all these operations are performed automatically.**

Below we describe these stages in more details. Throughout this section we will rely on an object-oriented notation to illustrate our program analysis. Namely, we assume that the program's code is grouped into methods, while methods as well as the data items belong to the classes. However, unless otherwise noted, the same approach can be applied to non-object oriented programs. In this

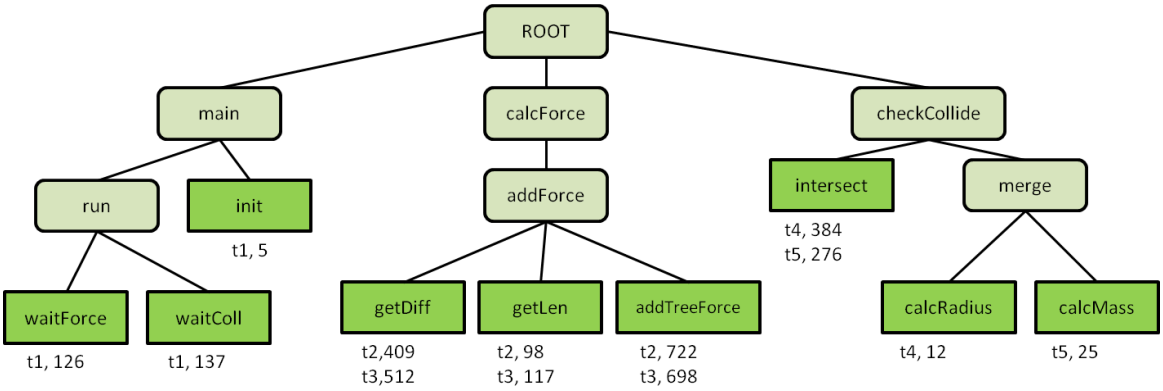

Figure 5.2: An example of a call trie

case the methods will correspond to the program's functions, and the program data is assumed to be stored in the global variables and/or structures.

5.2 Collecting stack samples

During the stack sampling stage our framework finds thread pools, frequently called functions and methods in the program, and frequently called libraries. Frequently called functions and methods serve as starting points for the static analysis of the program. The frequently called libraries need to be identified in order to generate the probabilistic call graphs correctly (see Section 5.4.2).

As the program is being executed, our framework periodically takes "snapshots" of the call stack of the running program. Different programming languages provide different means of collecting stack samples. In case of the Java application snapshots are collected by the Java agent class, which is inserted into the target program using the Java Management Extensions (JMX) mechanism. In C/C++ programs stack samples can be collected using debugging mechanisms that are specific to a particular operating system (OS).

Stack samples are merged to build a call trie of the program. A trie (also called a prefix tree) is a tree-like structure where the data entries are stored in leaf nodes, and non-leaf nodes form a prefix for these entries [87].

In a call trie each leaf node contains the code location being executed, and non-leaf nodes provide a call stack for that code location. Leafs are identified using the combination of a class name, a method name, and a line number being executed. For each leaf the framework maintains the list of pairs $\langle tr_1, ct_1 \rangle, \ldots \langle tr_n, ct_n \rangle$, where the ct_i is the number of executions of that code location by the thread tr_i (the notation used to describe stack sampling is provided in the Table 5.1).

An example of the call trie for a multithreaded program is depicted at the Figure 5.3. 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 $ct_1 = 126$. Similarly, the method

`getLen()` was executed by threads tr_2 and tr_3 98 and 117 times respectively.

The call trie is used to detect thread pools in the program. We detect thread groups in two steps. During the first step a map $\mathbb{T}$ is created. Its keys are thread tuples discovered by sampling, and values are execution counts for these tuples. For each leaf in the trie the framework retrieves a tuple $Tp_i = \langle tr_1, \dots tr_n \rangle$ of threads that executed the node along with the total number of executions $Ct_i = \sum(ct_1, \dots, ct_n)$. If $\mathbb{T}$ does not contains the tuple Tp_i, the pair $\langle Tp_i, Ct_i \rangle$ is inserted into $\mathbb{T}$. Otherwise the number of executions for the existing tuple is increased by Ct_i.

In our example the following tuples are created:

- $Tp_1 = \langle tr_1 \rangle$, $Ct_1 = 5 + 126 + 137 = 268$
- $Tp_2 = \langle tr_2, tr_3 \rangle$, $Ct_2 = 409 + 98 + 722 + 512 + 117 + 698 = 2556$
- $Tp_3 = \langle tr_4, tr_5 \rangle$, $Ct_3 = 384 + 276 = 660$
- $Tp_4 = \langle tr_4 \rangle$, $Ct_4 = 12$
- $Tp_5 = \langle tr_5 \rangle$, $Ct_5 = 25$

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 in order to be detected by the stack sampling. Consequentially, tuples Tp_4 and Tp_5, which correspond to those "spurious" thread pools were formed. As a result, an additional data cleanup is necessary in order to obtain an accurate set of thread pools.

During the second step the data in $\mathbb{T}$ is cleaned up. In particular, we detect spurious thread tuples in $\mathbb{T}$ and merge them with the correct ones. The tuple $< Tp_1, Ct_1 >$ is considered a spurious one and can be merged with $< Tp_2, Ct_2 >$ if and only if all threads in Tp_2 also present in Tp_1 and $Ct_1 \gg Ct_2$. The resulting tuple is formed as $< Tp_1, Ct_1 + Ct_2 >$. Once all the merges are complete, the tuples $Tp_1 \dots Tm \in \mathbb{T}$ represent the thread pools detected in the program.

In the example depicted at the Figure 5.3, the tuple Tp_4 and Tp_5 is merged into Tp_3 because $Ct_3 \gg Ct_4$ and $Ct_3 \gg Ct_5$. 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. In order to locate libraries the framework transforms the call trie into the call graph. For every function f the framework generates a set of callee functions $\langle f_1, ..., f_n \rangle$ that ever called f. If the number of callee functions $n > 1$, the function f is added to the set of *library functions*.

The results of the stack sampling are the .xml files that contain the following information:

h

Table 5.1: Notation used for thread pool detection

$tr_1, ..., tr_n$	The set of threads that ever executed a leaf node of the call trie
$ct_1, ..., ct_n$	The number of times each thread $tr_1, ..., tr_n$ executed a leaf node of the call trie
$Tp_i = \langle tr_1, ..., tr_n \rangle$	A tuple of threads executed by a leaf node of a call trie.
$Ct_i = \sum ct_1, ..., ct_k$	The total number of times the thread tuple $\langle tr_1, ..., tr_n \rangle$ executed leaf nodes in the call trie. The sum is calculated across all the leaf nodes in a trie.
$\mathbb{T}$	The map of thread tuples. Keys are thread tuples $Tp_1, ..., Tp_m$ detected during the stack sampling. Values are tuples of execution counts $Ct_1, ..., Ct_m$ for these tuples. After the cleanup step the keys $Tp_1, ..., Tp_m$ correspond to thread pools in the program.

- the set of thread pools in the program. For each thread pool the file contains names and IDs of thread which constitute that thread pool;
- the set of functions that belong to the program's libraries. Although the stack sampling may not detect some rarely executed library functions, this does not affect correctness of the PCGs in our experiments.

5.3 Static analysis

During the static analysis our framework scans the code of the program and detects synchronization CFs, I/O CFs, c_{in} and c_{out} CFs. It also detects creation of locks and queues in the program, which provides information for constructing the high-level model.

Generally, the set of classes obtained during the stack sampling does not include all the functions and methods that were executed during the program's run. To identify the missing code the framework builds the dependency graph of the program. It takes into account both code dependencies (e.g. a method from the class A calls the class B) and data dependencies (e.g. class A uses data from the class B or creates an instance of the class B). This allows to identify code that can be executed either by directly calling a function or a method by another method, as well as code that can be loaded and executed using reflection or dynamic loading. The desired superset of classes is the transitive closure of all the nodes in the dependency graph.

The static analyzer traverses the dependency graph, starting from the methods discovered during the stack sampling. It scans code of the methods (or functions for non-object oriented programs), searching for references to other classes and modules. Referenced classes and modules are loaded

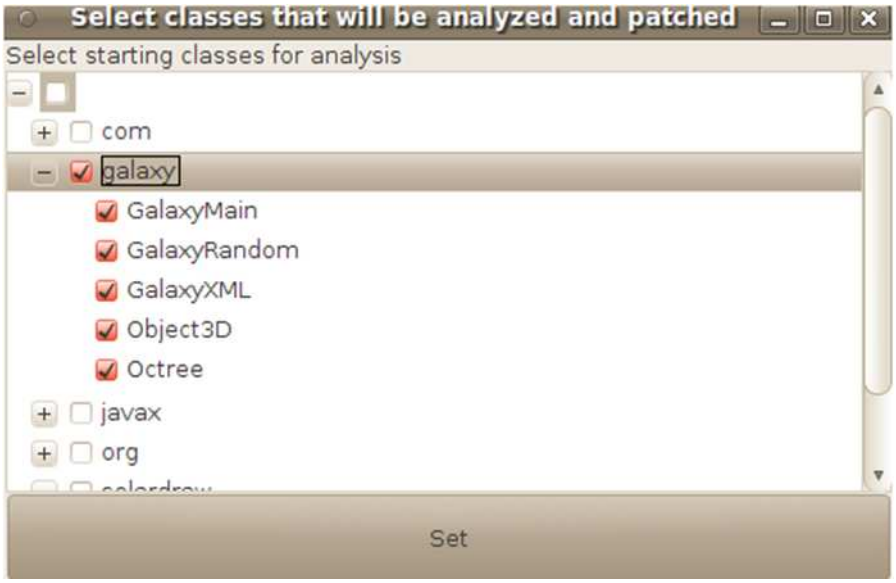

Figure 5.3: Selecting classes in Galaxy

and their methods are analyzed as well. The result of the search is the complete list of the methods that can be executed by the program.

However, it is often the case that some of the code discovered by the static analysis is widely referenced by the program, but, at the same time it does not perform perform synchronization, c_{in}, c_{out}, or I/O operations. In Java examples of such code are the fundamental data classes from `java.lang.*` package or classes from the or `java.math.*` package. Analyzing these classes will not add any useful information for the model. Thus in order to simplify both further analysis and the generated model, and to reduce the time required to analyze the program, the static analyzer allows the programmer to select the relevant program classes from a hierarchical dialog and then restrict the analysis to these classes. Figure 5.3 depicts the dialog box which allows filtering relevant classes in the Galaxy program.

Discovery of Code Fragments

Our static analyzer scans the code of the program in a search of certain constructs. These code constructs are treated as code fragments (CFs). For each CF the analyzer records the type of the CF (synchronization, c_{in}, c_{out}, or I/O CF), and its location within the program's code.

The detection of synchronization CFs

There are numerous ways to implement synchronization and thread interaction in the program. Practically all the modern programming languages provide low-level primitives to implement threading and synchronization. Normally, these primitives are built around the concept of the mutexes and condition variables [53]. In older versions of programming languages these low-level primitives

are implemented as external libraries. For example, in C mutexes and condition variables are implemented by the POSIX threading library. New languages provide these constructs as a part of the language specification. In particular, C11 and C++11 standards of C and C++ programming languages provide mutexes and condition variables as a part of the specification. In Java programming language those are implemented as synchronized regions and wait sets as a part of the language runtime.

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.

Developing threading semantics using high-level locks necessitates developers to implement these locks using low-level synchronization primitives, such as mutexes and condition variables. 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 in an arbitrary program, and determining their operation types *optype* may require complex analysis that is very hard to automate.

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, read-write locks, synchronization queues, concurrent collections, thread pools and other means for thread interaction. These constructs are designed to be used as building blocks for building multithreaded applications [9]. 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. This allows to reduce programming effort, increase performance and reliability of the application, and improve its maintainability [9].

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 for development of program analysis techniques that could accurately identify queues $Q = \{l_1...l_n\}$ and locks $L = \{l_1...l_m\}$ 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. Below we describe detection of locks, queues, and synchronization, c_{in}, and c_{out} CFs in such multithreaded program. We illustrate our approach using standard implementation of locks and queues from the `java.util.concurrent` package. In this package high-level locks and queues are implemented as classes, while synchronization operations are invocations of methods of these classes. This lets us accurately identify program-wide queues and locks and determine their semantics.

To detect synchronization CFs the analyzer searches for the specific functions and methods which constitute the API of the corresponding locking library. Every call to such method is considered as an instance of the synchronization CF. The type *optype* of the corresponding synchronization operation is deduced from the signature of the call.

For example, in a Java program the call to the `Semaphore.acquire(int permits)` is considered as a synchronization CF whose type is *optype*="Semaphore_acquire". Similarly, the call to the `Semaphore.release()` method is a synchronization CF whose type is *optype*="Semaphore_release". The call to the `CyclicBarrier.await(long timeout, TimeUnit unit)` method is a synchronization CF with *optype*="CyclicBarrier_await".

The framework also tracks low-level synchronization constructs, such as calls to mutexes and condition variables. These constructs are often used to implement simple synchronizations. For example, a mutex may be used to guard data in the program against concurrent modification by multiple threads. Low-level synchronization constructs are considered as synchronization CFs as well and are explicitly simulated in our models. For example, the entry to the synchronized region in a Java program is considered as a synchronization CF with *optype*="Mutex_entry". Exit from that region is a synchronization CF with *optype*="Mutex_exit".

However, if a combination of old-style synchronization constructs is used to implement a high-level lock such as a producer-consumer queue or a semaphore, the behavior of the resulting model might be incorrect. The reason is that the PCG may not capture deterministic behavior of such lock correctly. For example, the custom implementation of the cyclic barrier maintains the counter of threads waiting on the barrier. If the value of the counter is less than the barrier capacity, the calling thread is suspended. Otherwise, the program wakes up all the waiting threads. In the PCG this behavior will be reflected as a fork with the probability of waking up the thread 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.

The detection of c_{in}/c_{out} CFs

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 type of the CF, be it c_{in} or c_{out}, is deduced from the signature of the call.

For example, in a Java application a call to the `take()` or `poll()` method of `ArrayBlockingQueue` class is considered as a c_{in} CF. Similarly, a call to the `ArrayBlockingQueue.put()` or to the `LinkedBlockingDeque.offer(E e, long timeout, TimeUnit unit)` is considered as a c_{out}. It is important to track not only invocations of methods of particular classes, but also methods of particular interfaces. Thus in Java we also track invocations of the methods of `BlockingQueue` interface, whose underlying implementation may be one of the many Java blocking queue classes.

The detection of queues and locks

Instances of locks and queues are not directly detected during the static analysis. Instead, the analyzer detects calls to the constructors and initializers of such locks and queues. These calls are

not considered as CFs of any kind. However, they are used to detect queues and locks in the program and retrieve their parameters during the dynamic analysis. The types *ltype* of locks are obtained by analyzing signatures of these constructors and initializers.

The detection of I/O CFs

To discover I/O code fragments the static analyzer tracks API functions or methods that can perform disk I/O. The analyzer has two different strategies for detecting different types of disk I/O operations in a user program.

The first strategy is to track I/O operations that access filesystem metadata. Here the analyzer searches the code of the program for calls to the methods that can only initiate metadata I/O operations and do not perform other type of I/O activities. Calls to these methods are considered as I/O CFs. An example of such I/O CF in a C++ program is a call to the `stat()` `libc` function. In a Java program examples of such CF are calls to methods of the `File` class, such as `File.exists()`, `File.isFile()`, or `File.canRead()`.

The second strategy is for detecting file I/O operations. File I/O operations cannot be reliably detected in the same manner as metadata accesses. High-level programming languages such as Java or C# provide a variety of classes used to perform file I/O. However, same classes can be also used for other types of I/O, including network I/O or reading/writing to strings. Examples of such classes are `BufferedReader` and `TextReader` Java classes. Thus instead of tracking calls to these high-level classes the static analyzer treats bodies of low-level functions or methods that perform file I/O as I/O CFs. Some examples of such low-level methods in Java are native methods of the `FileInputStream`, `FileOutputStream`, and `RandomAccessFile` classes. These methods are executed for every file I/O operation in a computer program, and tracking those allows reliably detection of all the file I/O operations.

The results of the static analysis are the set of text and .xml files that contain following information:

- the set of synchronization, c_{in}, c_{out}, and I/O CFs; their locations in the program; the types of operations *optype* for synchronization, c_{in}, c_{out} CFs.
- the location of constructors of all locks and queues in the program; the types of these locks and queues.

Fragments of the program code that do not involve I/O or synchronization are considered as computation CFs and are detected during the dynamic analysis.

5.4 Dynamic analysis

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

Static analysis does not provide all the information necessary for building the model. It cannot explicitly detect locks and queues, it cannot determine which lock is called by the synchronization

Table 5.2: Information reported by instrumentation for different types of code fragments

The class of the instrumented CF	**Information reported by the start probe**	**Information reported by the end probe**
all CFs	- the timestamp - the probe ID - the thread ID	
lock/queue constructor	- parameters of the lock/queue	- the ObjectID of the lock/queue
synchronization	- the lock ID - the operation timeout	
c_{in}	- the queue ID - the operation timeout	- the ObjectID of the retrieved task
c_{out}	- the queue ID - the ObjectID of the task to be sent	
I/O	- the ID of the file accessed	

CF, it does not provide the parameters of the various CFs, and it does not detect computation CFs directly. To collect all this missing information we perform a second round of dynamic analysis of the program.

The dynamic analyzer instruments the program and runs it again in the same configuration as the initial stack-sampling run. The implementation details of the instrumentation are specific to the programming language used to write the program being analyzed and to the framework used to implement threading in that program. Below we describe the process of dynamic analysis in general terms, agnostic to the instrumentation details. A detailed description of how Java programs are instrumented is presented in the Section 5.4.1 of this thesis.

Each CF detected during the static analysis is instrumented with two probes. The *start probe* is inserted immediately before the start of the CF, and the *end probe* is inserted right after the end of the CF. Each probe is identified by the unique numeric identifier (probeID). The pair ⟨*start probe ID, end probe ID*⟩ identifies the CF.

During its execution the instrumented program generates a log file. The log file provides the sequence of probe hits on a per-thread basis, which constitute a *trace* of each thread's activity (see Figure 5.4 for an example of such trace).

The trace itself is divided into sections, where each section contains probe hits from a particular thread. Within the section each probe report the timestamp, the probe ID, and the thread ID. In addition, probes report extra information whose format depends on the CF being instrumented. Please refer to the Table 5.2 for a complete list of information returned by probes for the different CFs classes.

Two coincident probe hits in the trace represent execution of a single code fragment. Namely, the ending probe ID of the CF becomes the starting probe ID of the next CF, so code fragments

```
  ProbeID  Timestamp  ObjectID    Arguments/      Comments
                                  return value

THREAD 1 main                                    #Start of the section.
                                                 #Contains thread name and ID
  10       11345231   7683745     0 5             #reading from a queue (start probe)
  11       11387461   7683745     4387459         #reading from a queue (end probe)
  27       11391365   87235467                    #entering the sync. region (start)
  28       11392132                               #entering the sync. region (end)
  29       19873872   87235467                    #exiting the sync. region (start)
  30       19873991                               #exiting the sync. region (end)
  27       19874384   87235467                    #entering the sync. region (start)
  28       19875297                               #entering the sync. region (end)
  29       22155291   87235467                    #exiting the sync. region (start)
  30       22156112                               #exiting the sync. region (end)
  33       22157534   7683745     4387459         #writing to the queue (start probe)
  34       22172675   7683745                     #writing to the queue (end probe)
DONE                                             #end of the section
THREAD 10 CalcForceThread_1                      #Start of the next section
      ...
```

Figure 5.4: A sample fragment of the log file.

appear "overlapping" in the trace. In particular, the trace depicted at the Figure 5.4 represents execution of the following sequence of code fragments: $\langle 10, 11\rangle$, $\langle 11, 27\rangle$, $\langle 27, 28\rangle$, $\langle 28, 29\rangle$, $\langle 29, 30\rangle$, $\langle 30, 27\rangle$, $\langle 27, 28\rangle$, $\langle 28, 29\rangle$, $\langle 29, 30\rangle$, $\langle 30, 33\rangle$, $\langle 33, 34\rangle$.

Pairs of probe IDs that represent executions of I/O, synchronization, or c_{in}/c_{out} CFs in the trace are interleaved with pairs of probe IDs that represent computations. These pairs of probe IDs represent computation CFs, which were not detected explicitly during the static analysis.

The figure 5.4 depicts an example of such trace. Here the CF $\langle 10, 11\rangle$ is a c_{in} CF. It corresponds to the call to the `BlockingQueue.poll(int, TineUnit)` method, which reads the request from the queue. The object ID=7683745 recorded by the probe 10 identifies the queue. Argument values 0, 5 correspond to the timeout of 0 milliseconds, where "5" denotes the measurement unit (milliseconds), and "5" corresponds to the actual measurement. The probe 11 reports the return value 4387459, which is an ID of the retrieved object.

CFs $\langle 27, 28\rangle$ and $\langle 29, 30\rangle$ are synchronization CFs that correspond to the entry and exit from the synchronized region. The object ID=87235467 reported by the start probe identifies the monitor associated with that region. The end probe does not report the object ID in order to reduce the instrumentation overhead (the object ID is identical to one reported by the start probe).

CF $\langle 33, 34\rangle$ is an c_{out} CF. It corresponds to the call to the `BlockingQueue.put(E e)` method, which writes the request back to the queue. The object ID=7683745 identifies the queue. The sole argument of the call is the request object, denoted by the object ID=4387459.

Remaining tuples $\langle 11, 27\rangle$, $\langle 28, 29\rangle$, $\langle 30, 27\rangle$, and $\langle 30, 33\rangle$ are the computation CFs.

5.4.1 Instrumentation of Java programs

Our instrumentation must return a large amount of data about the program, including the probe ID, the timestamp, and information about method being instrumented (see Table 5.2). The method-specific information includes the ID of the object which contains the method (an ObjectID), the values of the method arguments, and the return value of the call. Instrumentation for collecting information on method calls can be implemented in two ways:

- By inserting probes into entries and exits of methods of interest;
- By inserting probes around instructions that call these methods.

Whenever possible we rely on the second approach. Although it is more complex, it preserves information about the origin of the call, which is necessary for generating a semantically correct model.

The details of the instrumentation depend on the nature of the CF being instrumented. If the CF is a call to the method that has no arguments (e.g. `CyclicBarrier.await()`), the first probe is inserted immediately before the JVM INVOKExxx instruction that performs the call (which can be one of INVOKEVIRTUAL, INVOKESPECIAL, or INVOKEINTERFACE instruction). In this case the objectID of the callee object can be retrieved from the Java execution stack without ruining it in the process. The second probe is inserted right after the INVOKExxx instruction. It reports the value returned by the call.

If the CF is a call to the method with at least one argument (e.g. `CyclicBarrier.await(int, TimeUnit)`), then callee ID and values of the method's arguments are stored in the Java execution stack. The execution stack does not allow accessing its values in the random order, so the instrumentation cannot retrieve callee ID and argument values directly from it. To obtain this information we instruments such methods separately using a wrapper.

Instrumenting the program using a wrapper is done in three steps. First, for every distinct call to the given method we create a static method inside the caller class – a *wrapper*. The first argument in the wrapper's signature is the reference to the callee object; the remaining arguments are same as the signature of the callee. The only functionality of the wrapper is to reroute the call to a callee method. Once called, the wrapper loads the reference to the callee object and values of its parameters from the operand stack, calls the callee method, and returns. Second, we instrument the wrapper itself. When the wrapper is being executed, values of its arguments are stored in the operand stack, which allows retrieving them in the random order. Our instrumentation retrieves values of all the arguments and reports them Finally, we modify the program so it will call not the original method, but the wrapper.

To detect file I/O operations we instrument entries and exits from the native JRE methods, which are called for every file read, write, open, and close operation in the program. This includes `FileInputStream.read()`, `FileInputStream.readBytes()`, `FileInputStream.open()`, `FileInputStream.close0()` methods.

The body of the native methods cannot be directly modified, thus we also employ wrappers in this case. First, we rename the actual native method by adding a prefix to it. Then within the same JRE class we create a wrapper method whose name and signature are identical to the name and the signature of the native method. As a result, all the calls to the native method will be rerouted through the wrapper. Finally, we instrument the wrapper so it will report the timestamp and values of the method arguments.

Instrumenting native methods allows detecting every file read, write, open, and close operation in the program. Corresponding probes have fixed probeIDs do distinguish them from the rest of instrumentation.

To instrument synchronized regions we just add probes before and after the corresponding MONITORENTER and MONITOREXIT instructions. In this case the probe reports object ID of the associated monitor object. To instrument the synchronized methods we convert these methods to non-synchronized. We insert MONITORENTER and MONITOREXIT instructions at the beginning and at the end of the method and clear the "synchronized" modifier for the method. After that we instrument them as usual synchronized regions.

Instrumentation of the Java program is performed dynamically upon the program startup. Our instrumentation involves adding and renaming methods in the program's classes. These operations are possible only before the class is loaded by the JVM, thus the whole process of instrumentation is performed in two steps. First, a JVMTI agent is invoked for each class before that class is loaded by the JVM. If necessary, the JVMTI agent adds wrappers to that class and transforms its synchronized methods.

Once all the classes are loaded, a second step begins. During the second step a JMX agent adds probes to the loaded program's classes using the ASM library for bytecode analysis and transformation. The agent implements the `ClassFileTransformer` interface and is injected into the program using the "-javaagent" option.

5.4.2 Construction of probabilistic call graphs

A naïve approach to generating the probabilistic call graph (PCG) 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 PCG. 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

$$p(s_i, s_j) = \frac{count(s_i, s_j)}{\sum_{l=k}^{m} count(s_i, s_l)} \tag{5.1}$$

Probabilities of transition for every pair of nodes constitute the mapping $\delta : S \to P(S)$ in our formal model.

The Figure 5.5 depicts the resulting graph for the trace example we discussed before (shown at the Figure 5.4).

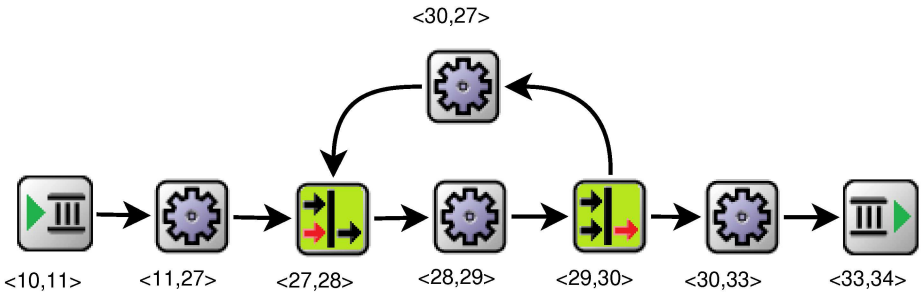

Figure 5.5: A resulting call graph for the trace sample

However, the naïve approach often results in problems while generating call graphs for real-world applications. First, it may not represent calls to the program's libraries correctly. Second, it generates an overly complex probabilistic call graph. Finally, it does not account for possible determinism in the behavior of the thread.

Below we discuss the issues arising with the the naïve approach to graph construction. We also present our solutions to these problems.

Correct representation of library calls.

Distinct non-intersecting execution paths in the program must be represented as non-intersecting paths in the PCG. This ensures that the control flow in the model will not be transferred from one such path to another. This condition, which is a prerequisite for a semantically correct functioning of the model, may not be preserved if some code fragments are located in one of the program's libraries.

Suppose that the program performs two unrelated calls to a library that contains a code fragment (CF). Executing that CF would emit the pair of same probe IDs for both library calls. But since probe IDs are used to uniquely identify CFs, both of these unrelated executions will be represented by the same node in the call graph. Correspondingly, two distinct execution paths in the PCG will be connected by the common node. As a consequence, during the simulation the thread model may "switch" from one execution path to another unrelated execution path, which is semantically incorrect.

For an example, consider the fragment of the Java program in the Figure 5.6. This program performs a simple sequence of actions: it enters the synchronized region (line 5), reads data from the file (6), and exits the region (7). Then it retrieves an object from the synchronized queue (8), reads data from another file (9), performs some lengthly computation (10), and writes the object to another queue (11). The "ground truth" call graph generated by hand for this program is depicted at the figure 5.7. It is a straight sequence of CFs without any loops, where two distinct file reads at lines 6 and 9 are represented with two distinct I/O CFs (CF2, CF5).

In this program two unrelated file read operations (lines 4 and 123) will call the same function `FileInputStream.readBytes()`, which contains an I/O CF $\langle 1013, 1014 \rangle$. Every such call will emit

```
Line no.
 1        byte[] signal1 = new byte[1024];
 2        byte[] signal_target = new byte[1024];
 3        FileInputStream file_in = new FileInputStream(inputFileName);
 4        FileInputStream file_target = new FileInputStream("file_target.wav");
 5        synchronized(lock) {
 6           file_in.read(signal1);
 7        }
 8        ComparisonResult comparison = queueIn.poll();
 9        file_target.read(signal_target);
 10       comparison.isMatched = doSignalMatch(signal1, signal_target);
 11       queueOut.push(comparison);
```

Figure 5.6: A code fragment that performs file I/O

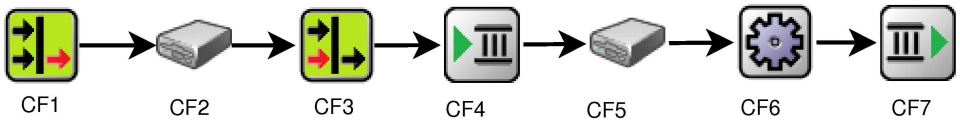

Figure 5.7: A call graph that should be generated for the code fragment

the same pair of probe IDs (1013 and 1014) into the log trace (see Figure 5.8), which correspond to the execution of the same CF $\langle 1013, 1014 \rangle$ twice. Thus instead of two distinct nodes corresponding to two different file reads, all the I/O operations will be represented by the single node in the call graph (see Figure 5.9). As a result, the request flow within the graph will contain a loop, which can be executed a number of times, or not executed at all. Any of these scenarios will lead to semantically incorrect behavior of the model.

If the loop will not be executed, the model will not release the mutex (the CF $\langle 29, 30 \rangle$ will not be executed). The model will also attempt to send a request to the destination queue (by executing the CF $\langle 33, 34 \rangle$) without reading it from the source queue. Alternatively, if the loop will be executed multiple times, the model will attempt to release the mutex during every iteration. Moreover, the model will attempt to read multiple requests from the queue (by executing CF $\langle 10, 11 \rangle$ in a loop), while writing out only a single request.

To simulate library calls correctly the dynamic analyzer uses the node splitting technique described in [79]. 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. As a result, calls to the library CFs that originate from different locations in the program are represented as separate nodes in the PCG. In terms of our model, the origin of the library call is represented as a sequence of one or more previous probe IDs, which form the *prefix* of a code fragment. In particular, to correctly represent the I/O CFs in the program a last non-library probe is added as a prefix to every I/O CF that performs file I/O operations (see Figure 5.10).

Large and complex programs use libraries extensively. As a result, similar problems will occur for different types of CFs, not just I/O CFs. Suppose the $\langle 1013, 1014 \rangle$ CF is located inside a library

```
ProbeID  Timestamp  ObjectID    Arguments/      Comments
                               return value
    ...
27       12415876   87235467                    #entering the sync. region (start)
28       12416127                               #entering the sync. region (end)
1013     12435875   1872565     8238435         #file read (start)
1014     18357221   1872565     1024            #file read (end)
29       18375678   87235467                    #exiting the sync. region (start)
30       18411752                               #exiting the sync. region (end)
10       18413873   7683745                     #reading from a queue (start)
11       18418937   7683745     4387459         #reading from a queue (end)
1013     18423144   5763292     5731786         #file read (start)
1014     29278981   5763292     1024            #file read (end)
33       34827982   7683745     4387459         #writing to the queue (start)
34       34828216   7683745                     #writing to the queue (end)
    ...
```

Figure 5.8: A corresponding fragment of the trace

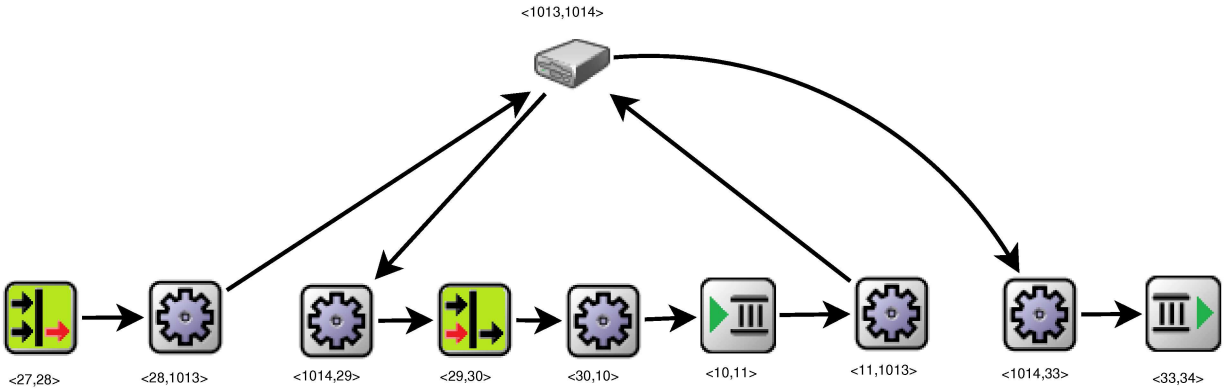

Figure 5.9: An incorrectly generated PCG for the code fragment that performs I/O

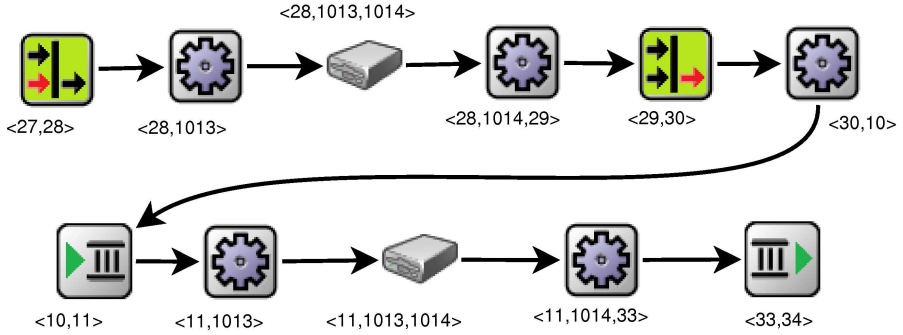

Figure 5.10: A correctly generated PCG with context information for I/O CFs

function and it performs a complex synchronization operation, such as acquiring a semaphore or awaiting on the barrier. Normally these operations control the order of the executions for the program as a whole. Violating any semantics of complex locks may result to a completely dysfunctional model. Thus developing a more general solution to a library CFs is necessary in order to allow modeling large and complex programs.

To obtain the context information for CFs the dynamic analyzer instruments calls to all the library methods discovered during the stack sampling (see Section 5.2). An entry *library probe* is inserted before every call to a library method; an exit library probe is inserted after such call. To minimize overhead library probes output only the probe ID.

As the analyzer scans the trace, it maintains a call stack of library probes. When the start 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.

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 500, 27, 28, 1013, 1014, 29, 30, 502, 10, 11, 1013, 1014, 503, 33, 34, 501. The corresponding sequence of CF is ⟨500, 27, 28⟩, ⟨500, 28, 1013⟩, ⟨500, 28, 1013, 1014⟩, ⟨500, 28, 1014, 29⟩, ⟨500, 29, 30⟩, ⟨500, 30, 10⟩, ⟨500, 502, 10, 11⟩, ⟨500, 30, 10⟩, ⟨500, 502, 11, 1013⟩, ⟨500, 502, 11, 1013, 1014⟩, ⟨500, 502, 11, 1014, 33⟩, ⟨500, 33, 34⟩. This sequence is which is consistent with the ground truth PCG.

Addressing probabilistic call graph bloat.

According to the naïve approach, all the computations between I/O, synchronization, and c_{in}/c_{out} CFs are represented as computation CFs. However, the majority of these computation CFs have a little (if any) impact on the performance of the program. For example, the call graph (see Figure 5.10) generated for the program depicted at the Figure 5.6 has 5 computational CFs. Out of these only the $\langle 11, 1014, 33\rangle$ CF represents a significant computation in the program (a call to the `doSignalMatch()` routine); its duration is 5549.0 microseconds. Remaining computation CFs $\langle 28, 1013\rangle$, $\langle 28, 1014, 29\rangle$, $\langle 30, 10\rangle$, and $\langle 11, 1013\rangle$ do not perform any computations and are artifacts of the instrumentation. Their summary duration is 44.53 microseconds, which constitute less than 1% of performed by the all computation CFs in this trace. Similarly, every synchronization region is represented as a pair of CFs, even if it is very short and never becomes contended in practice.

Presence of CFs that do not have any noticeable impact on performance leads to a highly bloated PCG. In particular, the PCG of a large and complex application may consist of thousands of CFs. Such unnecessary 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 simplification is performed in two steps (see Figure 5.11). The first step is finding stages in the program's execution that do not affect performance measurements and excluding these stages from modeling. The second step is analysis of the remaining CFs and eliminating those which do not have a noticeable impact on performance.

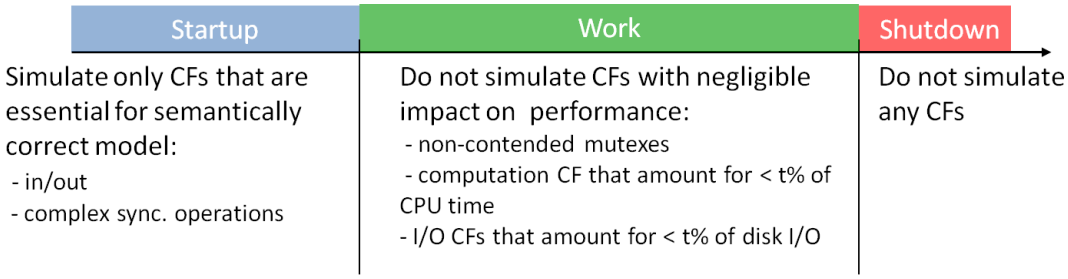

Figure 5.11: Model optimizations during different execution stages of the program

During the first step the whole timeline of the program's execution is split into three phases depending on their task processing behavior:

- The startup phase, when the program doesn't process tasks yet;
- The work phase, when the program processes tasks;
- The shutdown phase, when the program doesn't process tasks any more.

Finding stages 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 stages cannot be easily defined for a program, which can be the case for certain types of multithreaded programs (e.g. scientific computing applications or 3D renderers), we assume these stages are absent in the trace.

When the dynamic analyzer scans the trace of the startup stage of the program, it skips all the computation CFs, I/O CFs and *data-guarding regions*. Here data-guarding regions are the mutexes (or synchronized regions in a Java program) that contains only computations CFs, disk I/O CFs, and, possibly, another data-guarding regions. Usually, the sole purpose of data-guarding regions is to protect program's data from a modification by other threads. Skipped CFs are considered as insignificant and are not included into the model. Only CFs that affect semantics of the program's thread interactions are retained in the model. This includes c_{in}, c_{out} CFs as well as complex synchronization constructs, such as acquiring/releasing semaphores or awaiting on barriers.

The dynamic analyzer consider all the CFs executed during the shutdown stage as insignificant. In fact, when the program enters the shutdown stage, all the performance information has been already collected and there is no need to further simulate the program. Thus all the CFs executed during the shutdown stage are skipped from the model.

During the second step we detect insignificant CFs that were executed during the work stage and removing those from the model. Following categories of CFs encountered during the work stage are considered as insignificant:

- Non-contended data-guarding regions. A data-guarding 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.

The program may express the deterministic behavior, which may have a strong impact on performance. This deterministic behavior must be addressed in the model in order to obtain accurate prediction.

Currently we track and model two aspects of the program's determinism. First, the execution flow in the 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 we insert two "virtual" nodes after each s_{in} node in the PCG: $s_{in}^{fetched}$ and $s_{in}^{notfetched}$. The $s_{in}^{fetched}$ node is executed when the s_{in} CF was able to fetch the request from the queue. $s_{in}^{notfetched}$ is executed if s_{in} did not fetch the request and exited by timeout.

Second, representing program's loops as cycles in a PCG may affect the model's accuracy. If a loop that performs exactly n iterations is represented as a cycle in a PCG, then the number of iterations X for that cycle will not be a constant. It will rather be a random variable that follows a geometric distribution with mean n and a probability mass function:

$$Pr(X = k) = \frac{1}{n} \cdot (1 - \frac{1}{n})^{k-1} \tag{5.2}$$

In most cases this representation does not have a major 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 in programs with batch processing workloads, where the loop performs an initial population of the program's queues with tasks.

For an example, consider a scientific application that computes gravitational interaction between bodies using an $O(N^2)$ algorithm, where N denotes the total number of bodies. Suppose that the workload involves calculating interactions for 5 bodies ($N = 5$). Then assuming each iteration taking 1 millisecond, the average running time of the algorithm will be 25 milliseconds. However, if the loop that initializes the bodies is modeled as a cycle in the PCG then the total number of bodies in 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 [69]. If the loop contains the c_{out} node, the model explicitly simulates it as a loop with a given number of iterations. Otherwise the loop is represented as a cycle in the PCG.

We anticipate there may be other manifestations of the program determinism that affect the execution flow in the program. Thus we plan developing a more generic approach to modeling such determinism.

Retrieving parameters of code fragments

We retrieve parameters of the model's blocks from the instrumentation log of the program. Below we discuss parameter retrieval for different types of model blocks in detail.

Locks and request queues. Parameters of a lock include the type *ltype* of that lock and the type-specific lock parameters *lparam*. Also, each lock within the program must be identified, so it is implied that each lock $l_i \in L$ has a lock ID associated with it. Similarly, each queue $q_j \in Q$ must be identified using a unique queue ID.

Parameters of locks and queues are obtained from the arguments passed to constructors and initializers of these locks and queues, and from their return values. These constructors were detected during the static analysis and instrumented. Values of their arguments as well as the return values were reported in the instrumentation log.

In particular, the type *ltype* of the lock is inferred from the signature of the constructor/initializer of that lock. The type-specific parameters *lparam* are retrieved from the arguments passed to that constructor. The lock ID is obtained from the reference to the lock returned by the constructor. Parameters of the program's queues as well as identifiers of these queues are obtained in the same manner from the arguments of queue initializers and their return values.

For an example, consider the retrieval of parameters of locks and queues in a Java program written using the `java.util.concurrent` library. In `java.util.concurrent` the parameters of locks and queues are specified in the arguments to the constructors of the corresponding lock and classes. For example, the capacity of the cyclic barrier is specified by the value `parties` argument of the `CyclicBarrier(int parties)` constructor. The capacity of the queue is specified as the `capacity` argument of the `ArrayBlockingQueue(int capacity)` or `LinkedBlockingQueue(int capacity)` constructors. The ID of the object returned by the constructor uniquely identifies the corresponding lock or queue.

Synchronization and c_{in}/c_{out} CFs. Parameters of synchronization CFs include the reference to the lock l_i, the type of the synchronization operation *optype*, and the timeout τ_{sync}. Similarly, parameters of c_{in} and c_{out} CFs include the reference to the queue $q_j \in Q$ and the timeout τ_{in}, τ_{out}. The type of the operation is implied by the type of the CF itself: c_{in} CFs fetch requests from the queue, while c_{out} send requests to the queue.

Parameters of synchronization, c_{in}, and c_{out} CFs are 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 l_i is obtained from the reference to the lock; it is matched to the identifier of the l_i returned by the lock constructor/initializer. The type of synchronization operation *optype* is inferred from the signature of the called function or a method. The operation timeout τ_{sync} and τ_{out} is retrieved from the arguments passed to the function.

The concrete type of c_{in}/c_{out} CF is inferred from the signature of the function or method that carries out operation on the queue. The remainder of parameters for the c_{in}/c_{out} CFs are obtained in the same manner as parameters of the synchronization CFs.

For an example, consider again the discovery of parameters of synchronization, c_{in}, and c_{out} CFs in the Java program that relies on the `java.util.concurrent` library for locking. These CFs correspond to the calls to the methods of appropriate classes. In particular, the call to the `CyclicBarrier.await(long timeout, TimeUnit unit)` method will be considered as a synchronization CF. The actual type *optype* = *barrier.await* of synchronization operation is inferred from the method signature. The lock ID is the reference to the instance of the `CyclicBarrier` Java class being called. The timeout τ_{sync} is retrieved by analyzing the values of `unit` and `timeout` parameters. `unit` denotes units in which the timeout is measured (may range from days to nanoseconds), while the `timeout` parameter denotes the amount of timeout. Similarly, the call to the `ArrayBlockingQueue<E>.add(E e)` method is considered as an c_{out} CF, while the call to the `LinkedBlockingQueue.poll(long timeout, TimeUnit unit)` method is considered as a c_{in} CF.

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

Computation CFs. The parameter of the computation CF is the distribution $\mathbb{P}^{\tau}_{CPU}$ of CPU times for that CF.

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 a particular thread.

However, different programming languages and operating systems provide different mechanisms for measuring thread CPU time. As a result, developing a general-purpose method for obtaining precise and low-overhead measurements of actual CPU time across a wide range of existing programming languages, platforms, and operating systems remains a challenging problem.

In particular, modern operating systems usually offer API functions to get accurate low-overhead measurements of thread CPU times, such as `clock_gettime` in Linux, and `GetThreadTimes` in Windows. These APIs can be directly called by the programs compiled into the native code, such as programs developed using C or C++ programming languages.

However, these functions cannot be directly called from programs written using managed languages, such as Java. Although Java provides a `ThreadMXBean` class for measuring thread CPU

times, this class has a precision of one jiffie (10 ms in Linux), which is not sufficient for our purpose.

Thus in general we measure the τ_{cpu} as the difference between the timestamps of start and end probes of that CF, substituting clock time for CPU time. This approach allows precision up to 1 ns, but puts limitations on the program configurations we can use for building the model. Namely, we need to avoid configurations where CPU congestion is likely and clock time cannot be substituted for CPU time.

I/O CFs. Each I/O CF can initiate a number of low-level requests $\{dio_1, ..., dio_k\}$ to the disk subsystem. The parameters of the corresponding I/O CFs are the number k (in the case of the cache hit $k = 0$) and properties of these requests. Properties of the request include the type of the operation (read, write, metadata read, readahead, etc.) and the amount of data transferred.

This request-specific data can be retrieved only from the OS kernel. We use the BTrace [10] tool for this purpose. BTrace is launched simultaneously with the instrumented program and stores the log of all the low-level I/O requests.

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 Btrace log to the instrumentation log the dynamic analyzer uses cross-correlation – a technique used in signal processing [88]. 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 $(f \star g)[\Delta t]$.

Our analyzer represents sequences of I/O operations obtained from the kernel-mode trace and user-mode trace as signals that can take 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 BTrace log. The analyzer discretizes those signals with the sampling interval of one millisecond.

The Figure 5.12 depicts the cross-correlation between fragments of Java and Btrace logs, represented as signals $j(t)$ and $b(t)$ correspondingly. The cross-correlation signal $(j(t)f \star b(t))(t)$ reaches its maximum value at the point $\Delta t = 324$, which means that the Java signal $j(t)$ is shifted forwards by $\Delta t = 324$ milliseconds with relation to the Btrace signal $b(t)$.

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 $\Delta t_{ij} = \arg\max_t(xcorr_{ij})$. It considers that 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 all the signal pairs.

Next the analyzer aligns user and kernel-mode traces by subtracting the Δt from the timestamps of the user-mode trace. In the presented example logs are aligned by subtracting $\Delta t = 324ms$ from timestamps for all the events in the user-mode trace (see the Figure 5.13).

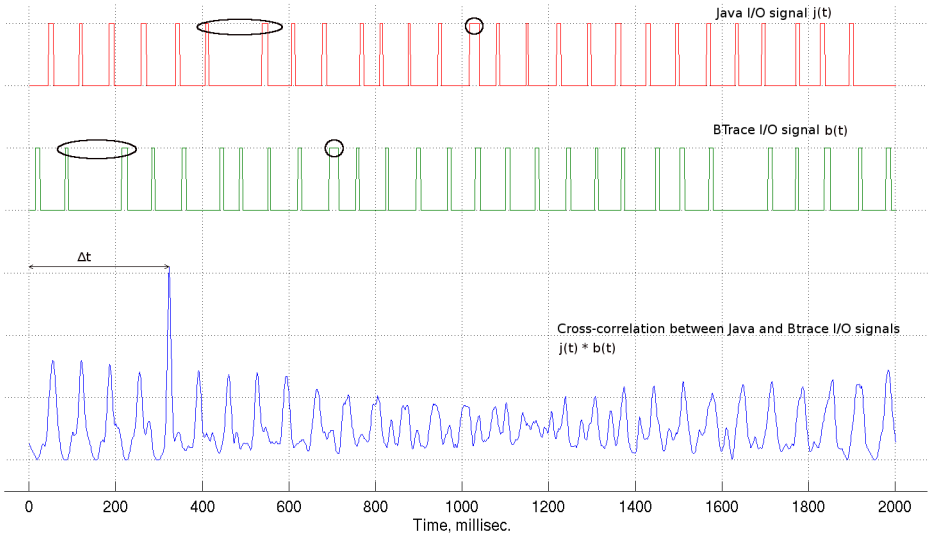

Figure 5.12: Cross-correlation between Java and Btrace I/O logs. Distinctive features of the signals are highlighted by circles.

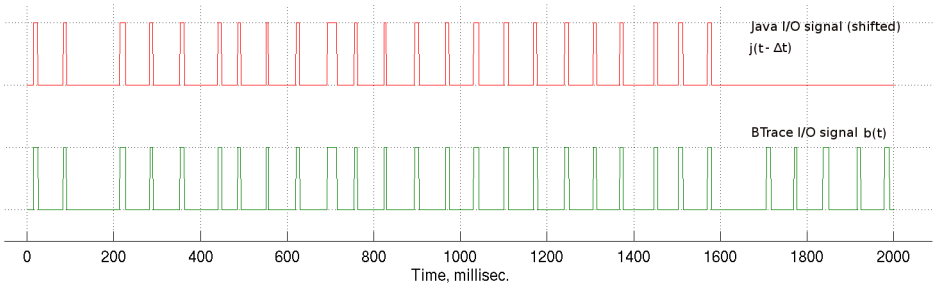

Figure 5.13: Java and Btrace I/O logs aligned

Finally, the kernel-mode I/O operations are associated with the user-mode states. Each kernel mode I/O operation dio_j is described as a time interval $[t^b_{start}, t^b_{end}]$ between its start/end timestamps. Similarly, invocations of the user mode I/O CFs are described as time intervals $[t^u_{start}, t^u_{end}]$. The kernel-mode I/O operation dio_j is considered to be caused by the user-mode I/O CF if the amount of intersection between their corresponding time intervals is maximal across all the I/O CFs in the trace. Correspondingly, a sequence $dio_j ... dio_{j+k}$ 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 ($k = 0$).

5.5 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 properties Π, and transition probabilities δ; the set of locks and their properties. This information is used to generate the three-tier PERSIK performance models described in the Section 3. The models are implemented using the OMNeT simulation toolset [2] and can reviewed in the OMNeT IDE.

Generating the resulting model is a relatively straightforward process because entities discovered during the program analysis have a direct representation in the PERSIK model.

Locks, queues, and threads discovered in the program explicitly match the corresponding blocks in the high-tier PERSIK model, while parameters of these locks and queues correspond to parameters of the appropriate PERSIK blocks. Thus constructing the high-tier PERSIK model requires generating the .ned file containing the set of PERSIK blocks of the corresponding types, and adding entries, which correspond to parameters of these blocks, into the model's .ini file.

Similarly, CFs explicitly correspond to the blocks of the appropriate mid-tier PERSIK models, and CF parameters correspond to parameters of these blocks. Constructing the mid-tier PERSIK model of a particular thread group involves generating the .ned file with the list of corresponding blocks, and putting parameters of these blocks into the .ini file. In addition, building mid-tier model requires linking the blocks according to the probabilities of the control flow δ. If CF s_i has only one successor CF s_k in the PCG, then the corresponding blocks in the PERSIK model are linked directly. If the CF s_i has multiple successors $s_k, \ldots s_m$, then the dispatch block is added to the mid-tier model. The block corresponding to the CF s_i is linked with the dispatch block, and the dispatch block is linked with blocks that represent succeeding CFs $s_k, \ldots s_m$. Probabilities of transition $p(s_i, s_k), \ldots p(s_i, s_m)$ are specified as parameters of the mid-tier PERSIK model.

To start using the model the analyst must specify ranges for 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). Ranges of these values are specified in a special .xml file. The model generator generates a separate PERSIK model for each combination of these parameters.

The analyst must also specify parts of the model, for which performance data should be collected. The model generator adds `SetTimer` and `ReadTimer` blocks in the corresponding places in the model. These blocks 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).

Specifying values of model parameters and specifying for which parts of the model performance must be collected are the only manual actions performed during the model construction.

Chapter 6

Experimental Evaluation

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

First we evaluate our methodology by building models of various small-to-medium size multithreaded programs. The purpose of this step is to ensure overall validity of our approach and to test our model building framework extensively. Finally we describe building performance models of large industrial applications. In this way we demonstrate that out approach can be used to build accurate models of large, industrial-grade multithreaded programs.

6.1 Automated performance modeling of small-to-medium size applications

We used our framework to build models of five small to moderate-size open source multithreaded Java programs: a 3D rendering program (Raytracer), a financial application (Montecarlo), two scientific computing applications (Moldyn, Galaxy), and a web server (Tornado). Source code of the framework, testing applications, and their models are available for download at [11]. Information on these programs and their models, including the size of of the program, the number of instrumentation probes, and sizes of their corresponding models (calculated as a number of distinct PTA states) is presented in the Table 6.1.

Montecarlo, Raytracer, and Moldyn are parts of the Java Grande (JG) benchmark [31] suite. JG was created before `java.util.concurrent` package was released, so it implements certain locks such as barriers and producer-consumer queues using old-style Java synchronization mechanisms. We modified these programs so they would use implementations of the same synchronization mechanisms from the `java.util.concurrent` library. We updated the source code of the Tornado [5] in a similar manner.

Our methodology for validating the accuracy of the models is similar to the one described in

Table 6.1: Information on small-to-medium programs and their models

	Raytracer	Montecarlo	Moldyn	Galaxy	Tornado
Program size (LOC)	1468	3207	1006	2480	1705
Number of probes	16	18	30	72	40
Number of CFs	43	17	72	124	88
Total number of nodes in the model	25	24	46	59	36

Section 4.3. We ran each program in a number of configurations and measured its performance in each of these configurations. Then we built the model of each program using one of the configurations, ran the model on the same set of configurations, and recorded the predicted performance. To get reliable performance measurements we performed three runs of both the actual program and its model in each configuration and used the mean values of measured and predicted performance metrics to estimate prediction accuracy. We assess model accuracy by calculating the relative error between measured and predicted performance metrics as

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

For the ideal model that predicts the program's performance without any errors $\varepsilon \to 0$.

We used two hardware configurations for our experimentation. The Setup I is a PC equipped with the Intel Q6600 quad-core CPU, 4GB RAM, and 160 GB HDD. The computer was running Ubuntu Linux OS.

The Setup 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.

To uncover potential artifacts in the performance of the test programs our experimental configurations cover a variety of program's behaviors, ranging from under-utilization of computation resources (e.g. when the number of active working threads is less than the number of CPU cores) to their over-utilization (the number of active threads is significantly higher than the number of cores).

Below we discuss each of our simulations in detail. First we discuss each program at the high level and describe its performance model. Then we present results of our experiments, in which we compare measured values of program's performance to predictions provided by our models.

6.1.1 Montecarlo: a financial application

Montecarlo simulates price of marked derivatives based on the prices of the underlying assets. Using historical data on asset prices, the program generates a number of time series using Monte Carlo simulation [12]. Each time series is represented as an instance of the corresponding object, and can be considered a "task" in the term of our formal model (see Section 3). Generation of time series is parallelized across a pool of working threads, with each thread generating its own subset of time series. 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.

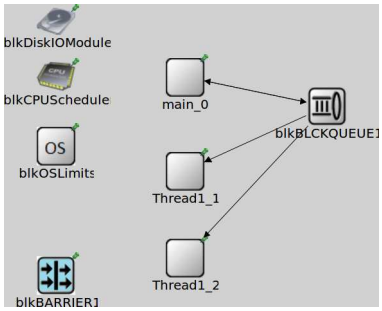

Figure 6.1: The high-level PERSIK model of the Montecarlo

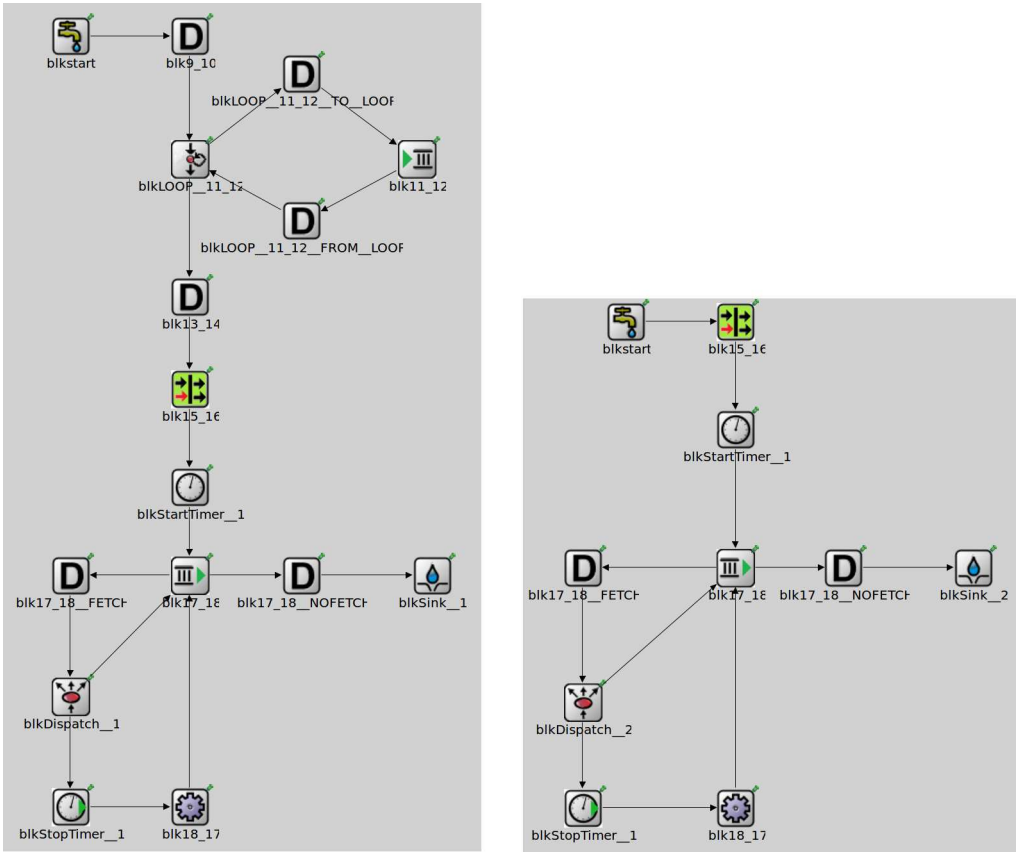

Figure 6.2: PERSIK models of the Montecarlo threads: the main thread (left) and the working thread (right).

The high-level model of the Montecarlo is depicted on the Figure 6.1. As we previously noticed, the high-level PERSIK model contains blocks that are both elements of the high-level formal thread model, such as queues, threads, request sources, and locks; as well as the blocks that represent entities in the low-level formal model, such as the disk I/O model and the CPU model (see Section 4.2 for

implementation details of PERSIK high-level models). Correspondingly, the high-level model of Montecarlo contains the `blkDiskIOModule` that implements a model of the disk I/O subsystem, the `blkCPUScheduler` that implements a model of a CPU and a thread scheduler, and the `blkOSLimits` that simulates limitations imposed by the OS. The latter two blocks are not used by the Montecarlo, they are automatically generated for every model.

Blocks that simulate underlying OS and hardware formally belong to the low-level formal model of the system, but to facilitate implementation these blocks are located in the high-level PERSIK model. As we noticed earlier (see Section 4.2 for implementation details of PERSIK high-level models), this is a purely cosmetic cosmetic feature that does not violate the semantics of the formal model.

The remainder of the blocks correspond to the components of the high-level formal model. `blkBARRIER1` barrier is used to synchronize working threads (the names of the model components are generated by the model building framework automatically). `blkBLCKQUEUE1` queue holds tasks, which correspond to time series that should be generated. The `main_0` block is the model of the main thread; `Thread1_1` and `Thread1_2` are models of the working threads.

The models of Montecarlo threads are depicted at the figure 6.2. In Montecarlo, the source code of the main thread and working threads is same, but their behavior is somewhat different. The main thread performs same computations as the working thread, but in addition to that, it also initializes program's locks and queues. This difference has been detected by our modeling framework, which generates distinct models for main and working threads despite the fact they share the same code.

In the model of the main thread, blocks `blk9_10` and `blk13_14` represent calls to the constructors of the `blkBLCKQUEUE1` and `blkBARRIER1` respectively. Tracking these calls is necessary for discovering parameters of the queue and the barrier, but they do not have any influence on the performance of the Montecarlo. Thus in the model they are represented using "virtual" blocks. These are `delay` blocks that introduce delay of 0 seconds. The loop consisting of `blkLOOP11_12` and `blk11_12` blocks performs initial population of the `blkBLCKQUEUE1` queue with tasks.

The remaining part of the model of the main thread is identical to the model of the working thread. `blk15_16` block is the call to the `blkBARRIER1`, which ensures that all the threads start computations simultaneously. Rest of the blocks simulate generation of the time series. The `blk17_18` block attempts to fetch the task from the `blkBLCKQUEUE1` queue. If the queue is not empty, it routes the CFM to the `blk17_18__FETCH` virtual block. From there the CFM is normally routed to the `blk18_17` block, which simulates computations performed while generating the time series, and then back to the `blk17_18`. Once the queue becomes empty, the `blk17_18` sends the CFM to the `blk17_18__NOFETCH` block, and the thread model stops.

Blocks `blkStartTimer__1` and `blkEndTimer__1` are respectively start and end timer blocks and are used to record the predicted running time of the program.

We predicted performance of Montecarlo on both hardware Setup I and Setup II.

Running Montecarlo in hardware Setup I with 4 CPU cores. We built the model of Montecarlo using a configuration with 2 working threads and used this model to predict the running time

of Montecarlo with 1,2,3,4,8,10,12,16 working threads. The comparison of measured and predicted performance is depicted at the Figure 6.3.

The workload of Montecarlo is perfectly parallelizable. Once the program has finished initialization, all its threads are running completely independent from each other, and are not engaged in any synchronization operations. As a result, the performance of the Montecarlo scales linearly with the number of available CPU cores. Once all the cores are utilized, the performance of the program no longer increases.

Numeric results for all the configurations are presented in the Table 6.2. The relative prediction error varies in $\varepsilon \in (0.014, 0.105)$ with the average error measured across all the configurations $\overline{\varepsilon} = 0.062$.

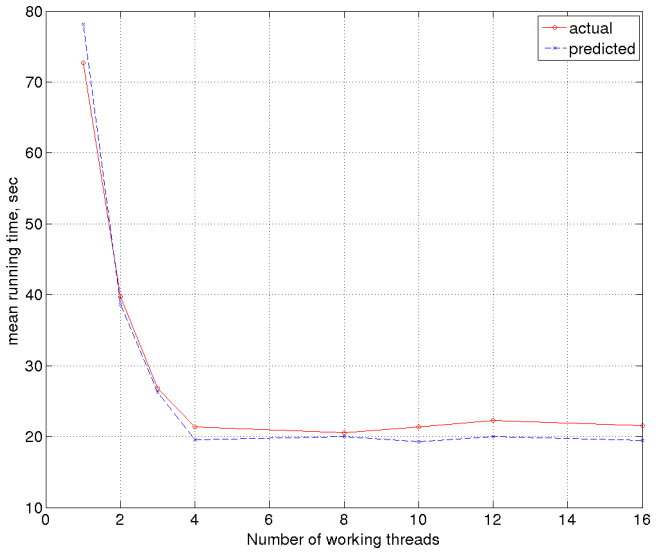

Figure 6.3: Predicted and measured running time for Montecarlo program on a 4-core machine

Table 6.2: Predicted and measured running time for Montecarlo program on a 4-core machine

The number of working threads								
	1	2	3	4	8	10	12	16
Measured running time	72.719	39.754	26.782	21.366	20.536	21.323	22.312	21.556
Predicted running time	78.275	39.202	26.111	19.677	19.811	19.530	19.962	19.838
Mean relative error ε	0.076	0.014	0.025	0.079	0.035	0.084	0.105	0.080

Running Montecarlo in hardware Setup II with 16 CPU cores.

We built the model of Montecarlo using a configuration with 4 working threads. We used this model to predict the running time of Montecarlo with 1,2,4,6,8,10,12,15 working threads. The comparison of measured and predicted performance is depicted at the Figure 6.4. Numeric results for all the configurations are presented in the Table 6.3.

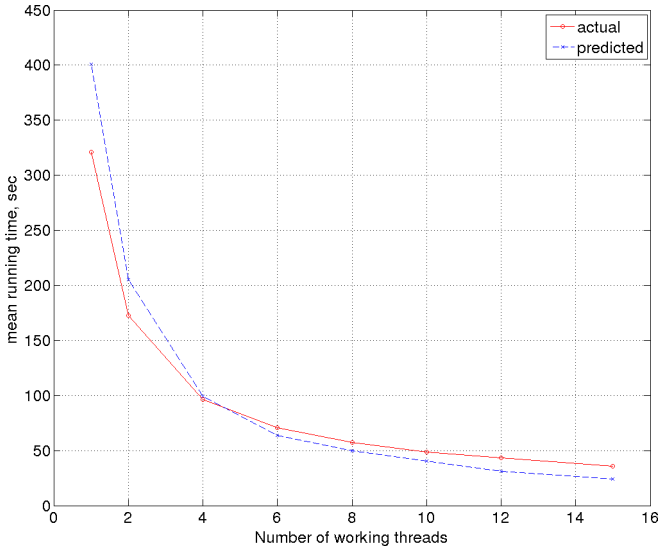

Figure 6.4: Predicted and measured running time for Montecarlo program on a 16-core machine

Table 6.3: Predicted and measured running time for Montecarlo program on a 16-core machine

The number of working threads								
	1	2	4	6	8	10	12	15
Measured running time	320.768	172.720	96.225	70.958	57.172	48.728	43.639	35.790
Predicted running time	400.629	205.197	99.048	64.005	49.648	40.394	31.156	24.367
Mean relative error ε	0.249	0.188	0.029	0.098	0.132	0.171	0.286	0.319

The relative prediction error for Montecarlo on a 16-core machine varies in $\varepsilon \in (0.029, 0.319)$ with the average error measured across all the configurations $\overline{\varepsilon} = 0.184$.

Although the prediction error remains within the acceptable limits, the performance of the Montecarlo becomes less linear in relation to the number of CPU cores. 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. We plan to address these errors by collecting the information on memory accesses by the program and by developing robust models of memory subsystem

6.1.2 Raytracer: a 3D renderer

Raytracer uses a ray tracing algorithm to render the scene containing 64 spheres at a resolution of N x N pixels. The rendering is parallelized across a pool of working threads; each thread renders a specific row of pixels in the image. Arrays containing pixel rows are considered as "tasks" in the terms of the formal model (see Section 3). These tasks are stored in a synchronized queue that is

initialized upon the start of the program.

The overall time required to render the frame is the most important performance metric of Raytracer. Given the constant size of the image, the number of working threads is a determining factor for the performance of the Raytracer.

The high-level model of Raytracer is depicted at the Figure 6.5. Apart from models of the thread scheduler, the disk I/O subsystem and the OS limits it contains models of a synchronized section `blkSYNCSECTION1` and the barrier `blkBARRIER1`. The synchronized section is used to calculate the control sum of the image (used by the JG framework to validate the rendering), and the barrier is used to sycnhronize working threads. The queue `blkBLCKQUEUE1` contains pixel rows (tasks). Initial population of the queue with task is performed by the main thread (`main_2` block)). Blocks `Thread2_0` and `Thread2_1` represent the models of working threads.

The models of Raytracer threads are shown at the Figure 6.6. Similarly to Montecarlo, the main thread (shown on the left) and working threads share same code, also their functionality is different. In the model of the main thread virtual code fragments `blk11_12` and `blk7_8` construct the queue and barrier correspondingly. The loop consisting of `blkLOOP__9_10` and `blk9_10` populate the queue with tasks (pixel rows). Next, `blk1_2` waits until all the threads complete initialization, so computations would start simultaneously. The cycle consisting of blocks `blk5_6` c_{in} and `blk6_5` c_{cpu} CFs perform actual image rendering. Once the image rendering is complete, the main thread calculates the checksum of all the pixels in the image. To do this, it uses the synchronized region consisting of `blk13_14` and `blk15_16` c_sync CFs. Calculating the checksum is relatively robust operation, when compared to rendering. Thus corresponding computation CF was considered as insignificant and omitted from the model. Again, `blkStartTimer__1` and `blkStopTimer__1` are used to measure the perfromance of the renderer.

The model of the working thread (shown on the right) has a simple semantics. Its nodes are a subset of the nodes of the main thread model, which include waiting on the barrier (`blk1_2` CF) and performing rendering itself (`blk5_6` and `blk6_5` CFs).

Running Raytracer in hardware Setup I with 4 CPU cores.

We built the model of Raytracer using a configuration with 3 working threads and used this model to predict performance of the Raytracer with 1,2,3,4,8,10,12,16 working threads. The comparison of measured and predicted performance is depicted at the Figure 6.7. Similarly to Montecarlo, the workload can be parallelized fairly well across all the CPU cores.

Numeric results are presented in the Table 6.4. The relative error varies in $\varepsilon \in (0.029, 0.156)$ with the average error measured across all the configurations $\overline{\varepsilon} = 0.117$.

Running Raytracer in hardware Setup II with 16 CPU cores.

We built the model using a configuration with 3 working threads. We used this model to predict performance of the Raytracer with 1,2,4,6,8,10,12,15 working threads. and used it to predict the running time of the program in the remaining configurations. The comparison of measured and predicted performance is depicted at the Figure 6.8. Numeric results are presented in the Table 6.5. The relative error varies in $\varepsilon \in (0.041, 0.173)$ with the average error measured across all the

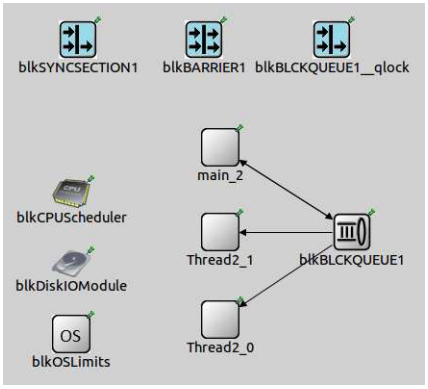

Figure 6.5: The high-level PERSIK model of the Raytracer

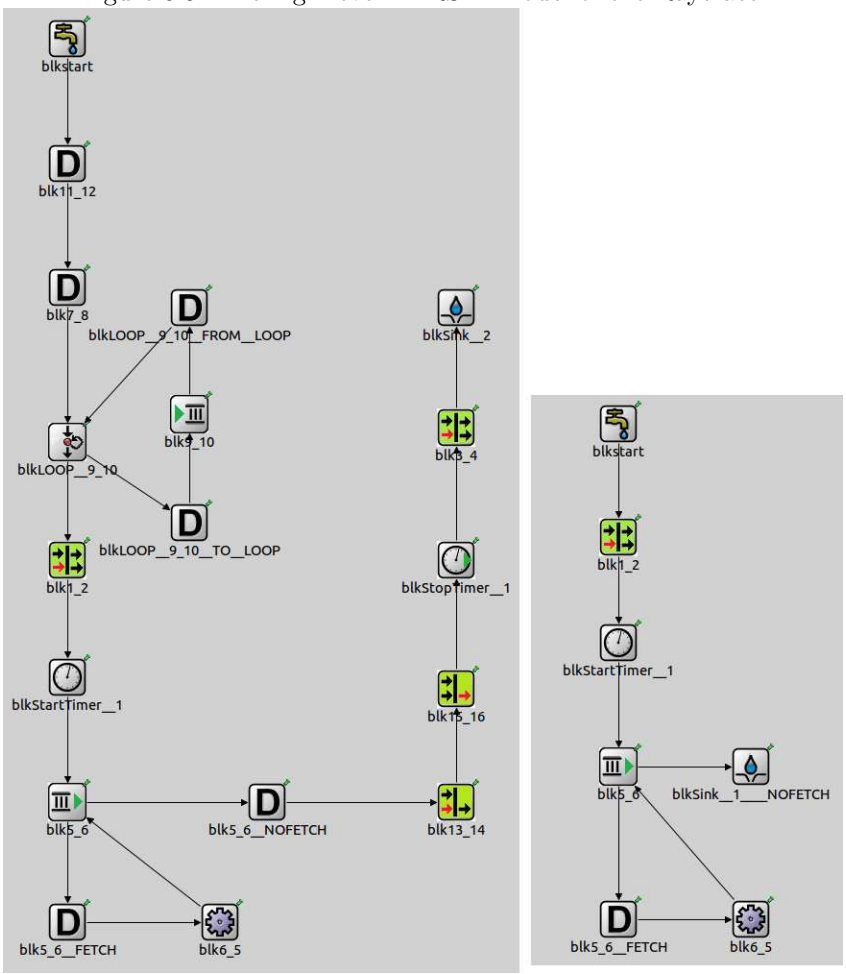

Figure 6.6: PERSIK models of the Raytracer threads: the main thread (left) and the working thread (right).

configurations $\overline{\varepsilon} = 0.086$. These results are comparable to those obtained in the Setup I.

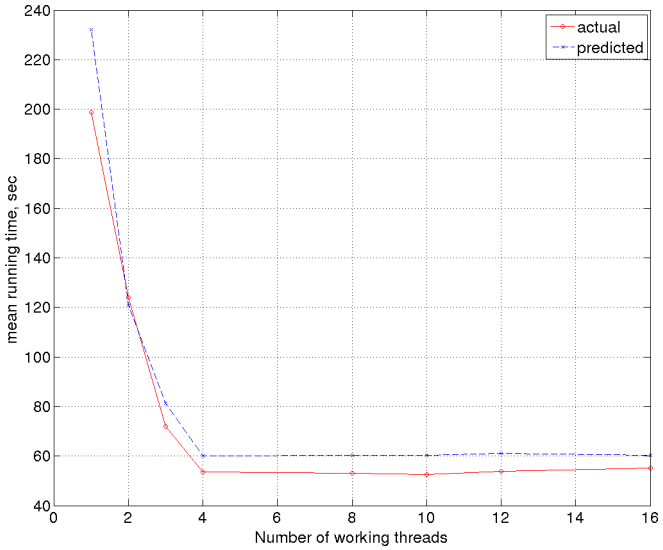

Figure 6.7: Predicted and measured running time for Raytracer program on a 4-core machine

Table 6.4: Predicted and measured running time for Raytracer program on a 4-core machine

	The number of working threads							
	1	2	3	4	8	10	12	16
Measured running time	198.707	123.871	72.073	53.545	52.973	52.547	53.938	55.056
Predicted running time	229.753	120.311	81.357	60.110	60.047	60.213	60.433	60.685
Mean relative error ε	0.156	0.029	0.129	0.123	0.134	0.146	0.120	0.102

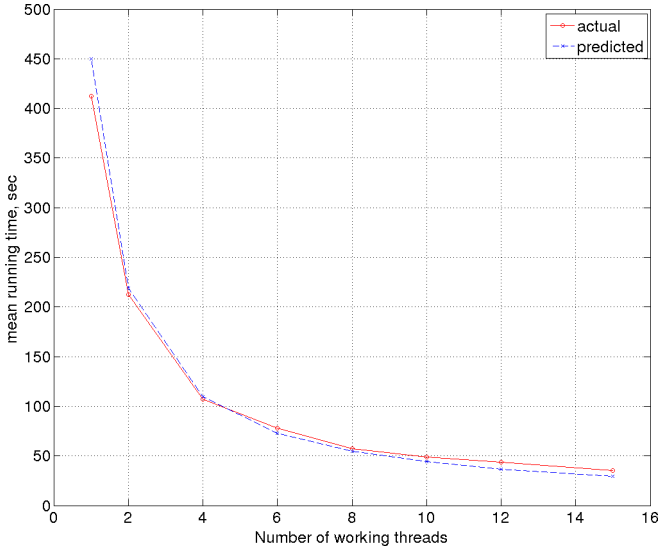

Figure 6.8: Predicted and measured running time for Raytracer program on a 16-core machine

Table 6.5: Predicted and measured running time for Raytracer program on a 16-core machine

	The number of working threads							
	1	2	4	6	8	10	12	15
Measured running time	412.199	212.287	106.853	77.623	57.174	48.827	43.944	35.369
Predicted running time	450.152	219.163	109.629	72.892	54.848	44.021	36.719	29.248
Mean relative error ε	0.092	0.032	0.026	0.061	0.041	0.098	0.164	0.173

6.1.3 Moldyn: a molecular dynamics simulator

Moldyn is the scientific computing program that simulates motion of the argon atoms interacting under the Lennard-Jones potential 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. One of these threads (the main thread) coordinates actions of other threads using barriers. Objects that represent atoms are stored in the global synchronized queue. At the beginning of each iteration the main thread copies atom's coordinates into separate data structures allocated for each thread. Working threads compute forces acting on atoms, and then the main thread merges forces computed by different threads and calculates updated positions of the atoms.

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

The high-level model of Moldyn is depicted at the Figure 6.9. Moldyn uses two queues. The `blkBLKCQUEUE1` is a queue that is initialized with coordinates of atoms upon the start of the program by the main thread. During each iteration, the threads are fetching tasks (object representing atoms) from that queue, perform computations, and store the atoms in the second queue `blkBLKCQUEUE2`. Once computations are complete, the main thread copies tasks back into the `blkBLKCQUEUE1`, preparing for the next iteration.

The Figure 6.10 depicts details of the Moldyn threads. Again, the main thread (left) and working threads (right) share the same code, but their functionality is different. In the model of the main thread blocks `blk3_4` and `blk5_6` initialize queues. The loop consisting of `blkLOOP__7_8` and `blk7_8` populate the queue `blkBLKCQUEUE1` with tasks (atoms and their coordinates).

The next couple of blocks are common for models of the main thread and working threads and are executed during each iteration. The block `blk9_10` is a barrier.await operation used to ensure that all the threads will start the iteration simultaneously. At the beginning of the iteration, Moldyn performs some short computations, but the corresponding computation CF was considered insignificant and is not represented in the model. Then Moldyn waits on the barrier again (block `blk9_10`). The cycle consisting of blocks `blk23_24`, `blk24_27`, and `blk27_28` represents computing forces acting on atoms. The `blk23_24` fetches the task (representing an atom) from the `blkBLKCQUEUE1` queue.

If the task can be fetched, the `blk24_27` computes the net force acting on an atom, and the block `blk27_28` writes the information about the atom and the computed force into the second queue `blkBLKCQUEUE2`. Blocks `blk13_14`, `blk15_16`, and `blk17_18` are also barrier.await blocks used to synchronize computations in the program. These computations, however, are too short and are omitted from the model. Tthe block `blk17_18` is the last block that is common to the model of the main thread and models of the working threads.

Once computations are complete, the main thread moves all the tasks from the queue `blkBLKCQUEUE1` to the `blkBLKCQUEUE2`. In the model this is represented by the cycle consisting of `blk25_26`, `blk26_29`, and `blk29_30`.

Finally, all the program's threads await on the barrier (block `blk19_20`). before starting another iteration. Blocks `blkStartTimer_1` and `blkStopTimer_1` are used to collect the main performance metric of Moldyn – length of the iteration.

Running Moldyn in hardware Setup I with 4 CPU cores.

We built the model of Moldyn using the configuration with 2 working threads. We used this model to predict performance of Moldyn in configurations with 1,2,3,4,8,10,12,16 working threads. The comparison of the measured versus predicted performance is depicted at the Figure 6.11. Numeric results are presented in the Table 6.6. The relative error varies in $\varepsilon \in (0.013, 0.155)$ with the average error measured across all the configurations $\overline{\varepsilon} = 0.083$.

Table 6.6: Experimental results for Moldyn

The number of working threads								
	1	2	3	4	8	10	12	16
Measured iteration length	0.503	0.264	0.214	0.170	0.147	0.147	0.162	0.163
Predicted iteration length	0.581	0.288	0.195	0.149	0.147	0.147	0.147	0.147
Mean relative error ε	0.155	0.093	0.092	0.124	0.006	0.001	0.093	0.096

Running Moldyn in hardware Setup II with 4 CPU cores.

We built the model of Moldyn using the configuration with 2 working threads. We used this model to predict performance of Moldyn in configurations with 1,2,4,6,8,10,12,15 working threads. The comparison of the measured versus predicted performance is depicted at the Figure 6.12. Numeric results are presented in the Table 6.7. The relative error varies in $\varepsilon \in (0.006, 0.485)$ with the average error measured across all the configurations $\overline{\varepsilon} = 0.255$.

The model predicts performance of Moldyn on a 16-core machine with a significantly 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 resut, as the number of threads increases, the CPU time for the CFs increases as well, which leads to the reduction in the accuracy.

Table 6.7: Experimental results for Moldyn

The number of working threads								
	1	2	4	6	8	10	12	15
Measured iteration length	0.627	0.404	0.220	0.177	0.144	0.131	0.115	0.119
Predicted iteration length	0.878	0.436	0.219	0.147	0.110	0.091	0.075	0.061
Mean relative error ε	0.400	0.081	0.006	0.173	0.238	0.305	0.349	0.485

One approach to improve the accuracy of simulation is to develop a model of a CPU cache. However, developing an accurate and compact model of a cache is a challenging task itself. Moreover, collecting data required to simulate the CPU cache may lead to increase in instrumentation overhead. An alternative way to improve the prediction accuracy could be developing a hybrid of a simulation and statistical model.

6.1.4 Galaxy: an n-body simulator

Previously (see Section 4.3) we built the model of the Galaxy manually in order to test our approach for performance prediction. In this section we build the model of the Galaxy automatically in order to test our approach for building performance models.

Galaxy simulates the gravitational interaction of celestial bodies. To improve the performance it uses the Barnes-Hut [23] algorithm which involves building an octree. A single iteration of Galaxy involves three consecutive actions: building the octree, computing the forces acting on bodies and updating their positions, and detecting collisions between bodies. The time taken by an iteration is the most important performance metric of the Galaxy.

The Galaxy parallelizes computations across two thread pools. The first thread pool ("force threads") computes forces and updates positions of bodies. The second thread pool ("collision threads") detects body collisions. Pools communicate through the synchronized queues. The order of operations is enforced by the main thread. The main thread is also responsible for rebuilding the octree. The number of force threads and the number of collision threads are the two parameters affecting the performance of the Galaxy.

The high-level model of the Galaxy is depicted on the Figure 6.13. In this model the request corresponds to the Java object that represent a celestial body. The leftmost blocks are models of program-wide locks. The `main_0` block is the model of the main thread; the `CalcForcesThread_1_3` and `CalcForcesThread_1_4` are models of the force threads; the `TrackCollisionsThread_1_1` and `TrackCollisionsThread_1_2` are models of the collision threads; the `XMLDumpThread_5` is the model of the thread that periodically dumps simulation results into the .xml file.

The models of Galaxy threads are depicted at the Figure 6.14. The model of the force thread is shown on the left. Upon the thread start, the CFM is sent to the `blk35_36 barrier_await` block. There it waits until all the requests are placed in the `blkBLCKQUEUE1` queue by the main thread. Once the initialization is complete, the barrier is lifted by the main thread, and the force threads iterate

through the requests in the blkBLCKQUEUE1. The thread attempts to fetch the request by using the blk41_42 s_{in} block. If the queue is not empty, the request is fetched and the CFM is routed to the blk42_39 block that simulates force computations. Next, the blk39_40 block emits the request to the blkBLCKQUEUE2 queue, and the CFM reaches back the blk41_42 block that attempts to fetch the next request. However, if the queue is already empty, the CFM is routed to the blk37_38 block. There it waits until all the force threads finish their computations. Once this happens, the CFM is sent to the blk35_36 block where it awaits for the next round of force computations.

The model of the collision thread shown on the right. Its structure and operations are similar to the model of the force thread. The blk21_22 block waits until all the force threads finish their computations and requests are placed into the blkBLCKQUEUE2 queue. Then the thread iterates through requests in the blkBLCKQUEUE2 queue. It fetches the request using the blk29_30 block, checks for collisions with other bodies (blk30_27), and writes the request it to the blkBLCKQUEUE1 queue (blk27_28). Once the queue is empty, the thread waits on the blk23_24 block until all its peers finish computations. Finally, the CFM is sent back to the blk21_22 block where it awaits for the next round of collision checks.

Running Galaxy in hardware Setup I with 4 CPU cores.

We built the model of the Galaxy using a configuration with 2 force and 2 collision threads. We have run the Galaxy with the 1,2,3,4,8,12,16 force threads and collision threads (total 7*7=49 configurations). In every configuration the Galaxy was used to simulate motion of 5000 bodies.

The comparison of the measured versus predicted performance is depicted at the Figure 6.15. Numeric results are presented in the Table 6.8. The relative prediction error varies in $\varepsilon \in (0.002, 0.291)$ with average error $\overline{\varepsilon} = 0.075$. The predicted CPU utilization is reported in the Table 6.9.

The model correctly predicts that the number of working threads have a relatively low influence over the performance of the Galaxy. Furthermore, it points to the non-linear dependency between the number of force threads and iteration length, which occurs because rebuilding the octree is not parallelized. For the same reason, Galaxy never fully utilizes all the four CPU cores (see Figure 6.16 and Table 6.9 for predicted CPU utilization).

Running Galaxy in hardware Setup II with 16 CPU cores.

We built the model of the Galaxy using a configuration with 2 force and 2 collision threads and used this model to predict performance of the Galaxy with the 1,2,3,4,8,12,16 force threads and collision threads (total 7*7=49 configurations). In every configuration the Galaxy was used to simulate motion of 5000 bodies.

The comparison of the measured versus predicted performance is depicted at the Figure 6.17. Numeric results are presented in the Table 6.10. The relative prediction error varies in $\varepsilon \in (0.004, 0.358)$ with average error $\overline{\varepsilon} = 0.092$, which is almost as accurate as the prediction for 4 CPU cores

The predicted CPU utilization is depicted at the Figure 6.18. The numeric results for predicting CPU utilization are reported in the Table 6.11. For this hardware setup we could also estimate the accuracy of prediction for CPU utilization. According to this data, the relative prediction error varies in $\varepsilon \in (0.004, 0.191)$ with average error $\overline{\varepsilon} = 0.080$.

Table 6.8: Predicted and measured iteration length for Galaxy on a 4-core machine

Measured iteration length, sec							
Num. force threads	The number of collision threads						
	1	2	3	4	8	12	16
1	0.230	0.226	0.245	0.250	0.243	0.245	0.274
2	0.139	0.136	0.126	0.127	0.146	0.149	0.154
3	0.096	0.083	0.088	0.089	0.106	0.105	0.109
4	0.068	0.066	0.074	0.068	0.070	0.070	0.076
8	0.072	0.067	0.071	0.071	0.070	0.071	0.069
12	0.070	0.067	0.071	0.068	0.069	0.071	0.069
16	0.074	0.066	0.070	0.069	0.069	0.068	0.068
Predicted iteration length, sec							
Num. force threads	The number of collision threads						
	1	2	3	4	8	12	16
1	0.261	0.248	0.244	0.242	0.242	0.242	0.242
2	0.146	0.133	0.129	0.127	0.127	0.127	0.127
3	0.108	0.095	0.090	0.088	0.088	0.088	0.088
4	0.088	0.076	0.071	0.069	0.069	0.069	0.069
8	0.088	0.076	0.071	0.069	0.069	0.069	0.069
12	0.088	0.076	0.071	0.069	0.069	0.069	0.069
16	0.089	0.076	0.072	0.069	0.069	0.069	0.069
Relative error							
Num. force threads	The number of collision threads						
	1	2	3	4	8	12	16
1	0.131	0.099	0.006	0.032	0.004	0.015	0.117
2	0.050	0.025	0.026	0.002	0.135	0.150	0.178
3	0.123	0.148	0.024	0.010	0.169	0.161	0.192
4	0.291	0.147	0.037	0.018	0.007	0.016	0.088
8	0.230	0.135	0.007	0.020	0.008	0.025	0.003
12	0.255	0.128	0.011	0.016	0.005	0.021	0.003
16	0.194	0.142	0.026	0.010	0.010	0.013	0.019

Table 6.9: Predicted CPU utilization for Galaxy on a 4-core machine, percent

Num. force threads	The number of collision threads						
	1	2	3	4	8	12	16
1	100.0	106.5	108.8	110.1	109.8	110.0	109.2
2	180.7	197.0	203.6	207.1	207.0	207.2	207.1
3	246.0	278.4	290.4	298.0	297.1	271.1	297.1
4	301.1	349.4	369.8	381.7	381.0	381.0	381.3
8	300.8	349.2	369.5	343.5	381.4	381.2	379.5
12	301.7	349.8	369.5	380.8	380.8	381.1	381.8
16	302.2	349.7	369.3	380.4	380.7	380.2	380.7

The non-linear dependency between the number of threads and performance becomes even more evident on a 16-core system. In fact, increasing the number of threads from 1 to 8 results in 5-fold improvement in performance. However, further increasing the number of threads from 8 to 15 improves performance only by 35%. This phenomenon is explained by the Amdahl's law [20]: as the number of active threads increase, the performance of the system becomes dominated by those computations that are not parallelized.

It might seem that the performance of the system can be easily predicted solely by the means of the Amdahl's law. However, in reality doing such prediction can be difficult. It requires discovering which computations are parallelized across working threads and which are not, and measuring CPU demand for both sequential and parallel computations.

One computation that is not parallelized is rebuilding the octree by the main thread. However, it is not the only sequential computation present in the program. Accessing the blocking queues by the multiple threads is also a sequential computation because it requires acquiring the mutex for the queue. As a result, fetching and putting data to the queues are performed sequentially. When the number of working threads is low, the overall impact of these queuing operations on the performance is negligible and they do not affect performance. However, as the number of threads increases, queuing operations introduce a noticeable effect on the program's performance and must be simulated for the accurate prediction.

6.1.5 Tornado: a simple web server

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. The behavior of the Tornado is discussed in general terms in the section 3; it is schematically depicted at the Figure 3.1.

The Figure 6.19 depicts the high-tier model of Tornado. The server does not have any synchronization primitives. However, the model contains two queues. The `blkACCEPT__QUEUE__25_26` queue represents the queue used by the operating system to store incoming TCP connections (connections themselves are generated by the `blkACCEPT__SOURCE__25_26` block). The `ListenThread_3` is a model of the accept thread, which listens on a socket and accepts incoming connections. This thread generates tasks that correspond to HTTP requests and stores them in the `blkBLCKQUEUE1`. Working threads represented by the `ServerThread0_1` and `ServerThread0_2` blocks fetch the task from the `blkBLCKQUEUE1` and process them.

The Figure 6.20 depicts the mid-tier models of Tornado threads. The model of the accept thread is shown on the right. The `blk25_26` block corresponds to the `ServerSocket.accept` operation. `blk26_35` simulates forming the task object, and the block `blk35_36` simulates storing the task object in the `blkBLCKQUEUE1` queue. Block `blkStartTimer__1` is used to timestamp the PERSIK request message, which is necessary to collect performance measurements.

The model of the working thread is depicted on the right. Here the `blk35_36` reads the request from the `blkBLCKQUEUE1`. `blk13_14` disk I/O block represents the call to the `File.exists()` method. If case if the requested page was not found on the disk, the execution flow is rerouted to the block

Table 6.10: Predicted and measured iteration length for Galaxy on a 16-core machine

Measured iteration length, sec								
Num. force threads	The number of collision threads							
	1	2	4	6	8	10	12	15
1	0.265	0.263	0.263	0.267	0.266	0.268	0.271	0.267
2	0.147	0.144	0.143	0.145	0.145	0.145	0.147	0.146
4	0.083	0.081	0.082	0.082	0.082	0.082	0.082	0.083
6	0.061	0.059	0.059	0.060	0.060	0.060	0.060	0.061
8	0.049	0.048	0.048	0.048	0.049	0.049	0.049	0.049
10	0.040	0.037	0.039	0.042	0.042	0.042	0.038	0.039
12	0.035	0.033	0.035	0.036	0.038	0.038	0.035	0.037
15	0.032	0.030	0.032	0.033	0.034	0.032	0.033	0.035
Predicted iteration length, sec								
Num. force threads	The number of collision threads							
	1	2	4	6	8	10	12	15
1	0.291	0.283	0.280	0.278	0.277	0.277	0.277	0.277
2	0.158	0.150	0.146	0.144	0.144	0.144	0.143	0.143
4	0.090	0.083	0.079	0.078	0.077	0.076	0.076	0.076
6	0.069	0.061	0.057	0.056	0.055	0.055	0.054	0.054
8	0.058	0.050	0.046	0.045	0.044	0.043	0.044	0.043
10	0.051	0.043	0.039	0.038	0.038	0.037	0.037	0.037
12	0.047	0.039	0.035	0.033	0.033	0.032	0.033	0.032
15	0.043	0.034	0.030	0.029	0.028	0.028	0.028	0.028
Relative error								
Num. force threads	The number of collision threads							
	1	2	4	6	8	10	12	15
1	0.099	0.077	0.063	0.041	0.042	0.034	0.025	0.038
2	0.076	0.040	0.018	0.005	0.006	0.004	0.031	0.021
4	0.095	0.032	0.037	0.057	0.059	0.069	0.071	0.078
6	0.127	0.036	0.034	0.082	0.082	0.094	0.102	0.106
8	0.178	0.038	0.042	0.079	0.107	0.121	0.119	0.126
10	0.286	0.164	0.003	0.105	0.109	0.134	0.048	0.060
12	0.358	0.173	0.010	0.085	0.133	0.154	0.070	0.126
15	0.315	0.150	0.053	0.113	0.171	0.101	0.162	0.194

Table 6.11: Predicted and measured CPU utilization for Galaxy on a 16-core machine

Measured CPU utilization, percent								
Num. force threads	The number of collision threads							
	1	2	4	6	8	10	12	15
1	104.3	190.7	328.0	444.3	533.5	614.6	681.2	776.3
2	108.5	200.8	347.4	465.0	581.1	651.4	737.7	832.1
4	112.6	209.8	362.6	490.9	597.8	688.6	761.4	860.9
6	114.9	210.4	368.1	488.7	595.2	689.4	759.8	843.6
8	119.1	213.3	375.6	494.9	597.4	687.8	770.9	845.7
10	121.6	218.3	369.2	503.3	605.1	695.7	766.4	869.1
12	117.3	218.3	371.2	504.1	612.9	699.1	786.9	845.6
15	117.7	222.6	373.7	501.4	611.7	689.3	754.6	876.6
Predicted CPU utilization, percent								
Num. force threads	The number of collision threads							
	1	2	4	6	8	10	12	15
1	100.0	184.6	320.9	423.0	502.5	567.7	618.3	684.5
2	102.8	194.6	349.6	476.3	583.7	674.8	745.8	849.5
4	104.2	199.7	368.8	511.7	633.3	743.8	831.9	961.8
6	104.7	201.7	374.1	523.3	652.2	770.1	881.9	1001.8
8	104.9	202.2	376.9	528.5	661.2	770.1	892.2	1025.8
10	105.1	202.5	379.8	532.5	671.0	789.1	898.2	1024.1
12	105.1	203.4	380.8	537.5	666.6	793.5	888.3	1045.3
15	105.2	203.2	380.6	536.5	672.3	789.3	901.6	1024.5
Relative error								
Num. force threads	The number of collision threads							
	1	2	4	6	8	10	12	15
1	0.044	0.033	0.022	0.050	0.062	0.084	0.102	0.134
2	0.058	0.032	0.006	0.022	0.006	0.035	0.012	0.023
4	0.081	0.050	0.017	0.040	0.056	0.074	0.083	0.106
6	0.097	0.043	0.016	0.065	0.087	0.107	0.139	0.158
8	0.135	0.056	0.004	0.065	0.097	0.108	0.137	0.175
10	0.159	0.078	0.028	0.057	0.098	0.119	0.148	0.152
12	0.114	0.073	0.025	0.062	0.082	0.119	0.114	0.192
15	0.120	0.095	0.019	0.065	0.091	0.128	0.163	0.145

blk12__23_24, which represents sending the "404" HTTP error code to the user and closing the socket. Otherwirse the model executes blocks blk8__10202_29 or blk12__10204_9 and blk30_10205 (here 8, 12, and 30 denote the prefix of the library CF). The loop consisting of I/O block blk30_-_10205_10206 and computation block blk30__10206_10205 simulates reading the file from the disk in chunks. Finally, blk30__10206_10203 and blk30__23_24 blocks simulate preparing the HTTP response and sending it to the user.

The blocks blkStopTimer__1 and blkStopTimer__2 measure the response time for each request. Each block covers one possible path that the request processing can take. blkStopTimer__1 block measures the response time when the requested page was found on the disk and the request was processed successfully. The blkStopTimer__2 block measures the response time when the corresponding page was not found.

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.

In our experiments we used Tornado to host about 200000 Wikipedia web pages. We used a separate client computer (an Intel 2.4 GHz dual-core CPU, 4 GB RAM, 250 GB HDD) to simulate the incoming connections. The client was connected to the web server over the 100 MBit LAN. The client was running the http_load software which accesses web pages from a supplied list with a specified IRR.

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). To ensure accurate measurements the IRR was measured on the server side; the client was configured to issue 20 to 140 rps which could not be sustained due to computational overhead. Our configuration space includes all possible combinations of these parameters. During each run 20000 requests were issued.

The model of the web server was built using a configuration with IRR=57.30 requests per second (RPS) and 1 working thread. The models of Tornado threads are depicted at the figure 6.20. The behavior of these models is described in sections 4.2.1 and respectively.

The prediction of the response time is shown at the Figure 6.21. Predictions of the throughput are shown at the Figure 6.22. Numeric results are presented in tables 6.12 and 6.13 correspondingly.

The relative prediction error for R varies in $\varepsilon(R) \in (0.017, 1.583)$ with average error $\overline{\varepsilon}(R) = 0.249$. One cause for the relatively high error terms for predicted response time is the variance in the disk I/O resource demand across different configurations. Another cause is the simplistic model of networking operations, which are currently simulated as CPU computations. A more accurate networking model remains the subject of future work.

Predictions of the throughput and hard drive utilization are significantly more accurate The error for the througput T $\varepsilon(T) \in (0.000, 0.051)$ and $\overline{\varepsilon}(T) = 0.012$. The error for the predicted hard drive utilization $\varepsilon(U) \in (0.000, 0.077)$, while $\overline{\varepsilon}(U) = 0.025$.

Since the server is equipped with a single hard drive, the increase in the number of parallel I/O

Table 6.12: Experimental results for Tornado (response time)

Measured response time, sec													
Num. threads	IRR, req/sec												
	19,75	29,43	38,97	48,27	57,31	65,88	74,22	82,69	88,35	91,17	94,67	97,02	97,19
1	0.013	0.013	0.014	0.015	0.016	0.020	0.030	0.814	6.573	12.545	15.002	21.677	23.477
2	0.013	0.013	0.013	0.015	0.016	0.025	0.052	1.573	10.642	14.824	20.263	24.257	25.392
4	0.013	0.013	0.014	0.015	0.017	0.022	0.040	0.474	8.599	13.846	17.898	22.872	24.293
8	0.013	0.013	0.013	0.015	0.016	0.021	0.032	0.079	2.740	9.871	12.571	17.815	20.697
Predicted response time, sec													
Num. threads	IRR, req/sec												
	19,75	29,43	38,97	48,27	57,31	65,88	74,22	82,69	88,35	91,17	94,67	97,02	97,19
1	0.012	0.013	0.014	0.018	0.024	0.035	0.078	1.165	10.811	14.378	18.336	24.810	24.725
2	0.012	0.013	0.014	0.017	0.022	0.035	0.059	0.324	3.160	8.847	14.860	19.121	21.207
4	0.012	0.013	0.014	0.017	0.021	0.030	0.055	0.271	5.171	11.100	14.419	18.493	20.520
8	0.012	0.012	0.013	0.015	0.017	0.022	0.033	0.090	1.849	6.492	9.482	16.341	17.375
relative error													
Num. threads	IRR, req/sec												
	19,75	29,43	38,97	48,27	57,31	65,88	74,22	82,69	88,35	91,17	94,67	97,02	97,19
1	0.035	0.031	0.051	0.221	0.516	0.760	1.583	0.432	0.645	0.146	0.222	0.145	0.053
2	0.048	0.022	0.050	0.128	0.338	0.414	0.134	0.794	0.703	0.403	0.267	0.212	0.165
4	0.043	0.040	0.022	0.116	0.217	0.386	0.365	0.429	0.399	0.198	0.194	0.191	0.155
8	0.075	0.078	0.023	0.017	0.047	0.074	0.050	0.135	0.325	0.342	0.246	0.083	0.161

Table 6.13: Experimental results for Tornado (throughput)

Measured throughput, req/sec													
Num. threads	IRR, req/sec												
	19,75	29,43	38,97	48,27	57,31	65,88	74,22	82,69	88,35	91,17	94,67	97,02	97,19
1	19.77	29.45	39.02	48.36	57.44	66.01	74.51	82.87	85.56	84.99	85.50	84.47	84.82
2	19.76	29.48	39.03	48.32	57.43	66.05	74.28	81.68	80.98	81.40	81.58	81.34	81.95
4	19.76	29.47	39.00	48.34	57.47	66.02	74.38	82.88	82.82	83.03	83.23	82.92	83.10
8	19.77	29.45	39.03	48.38	57.37	66.03	74.49	83.11	88.82	88.17	88.36	88.40	87.61
Predicted throughput, req/sec													
Num. threads	IRR, req/sec												
	19,75	29,43	38,97	48,27	57,31	65,88	74,22	82,69	88,35	91,17	94,67	97,02	97,19
1	19.76	29.41	38.96	48.29	57.45	65.86	74.40	81.95	81.94	82.66	82.80	81.99	83.18
2	19.76	29.44	38.97	48.22	57.29	65.89	74.01	82.40	85.13	84.39	84.66	84.40	84.70
4	19.75	29.44	38.97	48.30	57.41	65.86	74.20	82.30	84.78	84.81	85.03	85.57	85.14
8	19.75	29.42	39.00	48.32	57.31	65.87	74.19	82.98	89.21	90.26	90.36	89.04	90.03
Relative error													
Num. threads	IRR, req/sec												
	19,75	29,43	38,97	48,27	57,31	65,88	74,22	82,69	88,35	91,17	94,67	97,02	97,19
1	0.001	0.001	0.002	0.001	0.000	0.002	0.001	0.011	0.042	0.027	0.032	0.029	0.019
2	0.000	0.001	0.002	0.002	0.003	0.003	0.004	0.009	0.051	0.037	0.038	0.038	0.034
4	0.001	0.001	0.001	0.001	0.001	0.002	0.002	0.007	0.024	0.021	0.022	0.032	0.025
8	0.001	0.001	0.001	0.001	0.001	0.003	0.004	0.002	0.004	0.024	0.023	0.007	0.028

operations is negated by the proportional increase in the average execution time for each I/O request. As a result, the number of working threads has a relatively small influence on the performance of the

server (see Figure 6.23 and Table 6.14 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.

Table 6.14: Experimental results for Tornado (hard drive utilization)

Measured hard drive utilization, percent													
Num. threads	IRR, req/sec												
	19,75	29,43	38,97	48,27	57,31	65,88	74,22	82,69	88,35	91,17	94,67	97,02	97,19
1	20.02	30.20	40.04	50.02	59.02	68.48	78.05	90.02	92.17	91.53	91.51	91.31	90.88
2	20.29	29.68	39.93	50.37	59.77	73.17	86.76	99.33	99.37	99.37	99.29	99.23	99.25
4	20.35	30.22	40.12	50.78	60.94	71.08	84.55	98.74	99.69	99.71	99.74	99.74	99.75
8	20.33	30.24	40.18	50.36	60.09	71.28	84.14	95.10	99.74	99.71	99.73	99.73	99.72
Predicted hard drive utilization, percent													
Num. threads	IRR, req/sec												
	19,75	29,43	38,97	48,27	57,31	65,88	74,22	82,69	88,35	91,17	94,67	97,02	97,19
1	20.95	30.26	40.02	50.28	59.96	68.72	78.07	85.93	86.46	86.40	86.39	86.45	86.25
2	20.95	31.12	41.64	52.75	64.38	76.30	85.49	96.52	98.99	99.04	99.04	99.05	99.07
4	21.18	30.90	41.72	52.93	64.08	75.20	86.42	96.83	99.94	99.98	99.98	99.96	99.98
8	20.41	29.59	39.63	49.50	58.93	69.00	78.98	90.66	99.63	99.94	99.99	99.99	99.98
Relative error													
Num. threads	IRR, req/sec												
	19,75	29,43	38,97	48,27	57,31	65,88	74,22	82,69	88,35	91,17	94,67	97,02	97,19
1	0.046	0.002	0.001	0.005	0.016	0.004	0.000	0.045	0.062	0.056	0.056	0.053	0.051
2	0.033	0.048	0.043	0.047	0.077	0.043	0.015	0.028	0.004	0.003	0.003	0.002	0.002
4	0.041	0.023	0.040	0.042	0.051	0.058	0.022	0.019	0.003	0.003	0.002	0.002	0.002
8	0.004	0.021	0.014	0.017	0.019	0.032	0.061	0.047	0.001	0.002	0.003	0.003	0.003

6.2 Automated performance modeling of large applications

In the previous section we described building performance models of small- to medium-size applications, which demonstrate overall feasibility of our approach. However, modern multithreaded applications are significantly larger and more complex than programs we have studied in a previous section. In order to prove the practical value of our methodology we must demonstrate that our framework is capable of building accurate models of large industrial applications.

To prove the practicality of our approach we built performance models of industrial open source Java programs: Sunflow 0.07 3D renderer and Apache Tomcat 7.0 web server. These are large applications that consists of tens of thousands and hundreds of thousands of lines of code (LOC). The summary information on programs and their models is provided in the Table 6.15.

We predicted performance of Tomcat in two setups: as a standalone web server that hosts static web pages and as a servlet container that hosts an iText library for text conversion. Because of the difference in Tomcat functionality between these setups, there is a significant dissimilarity between the corresponding models.

Instrumenting these programs required inserting hundreds and thousands of probes into the

Table 6.15: Summary information on large applications 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

bytecode of these applications. Instrumentation did not alter semantics of our test programs, but it introduced some overhead. We measured overhead as a relative difference between the time required to process a task by an instrumented and non-instrumented program. According to this methodology, the overhead constituted 2.5%-7.6%.

Program analysis identified a large number of code fragments in the code of test programs: from 380 CFs in Sunflow 3D renderer to 3926 CFs in the Tomcat servlet container. The complexity reduction algorithm eliminated 99.3% (11124 out of 11206) to 99.5% (9944 out of 9993) CFs as insignificant in Tomcat and Tomcat+iText models correspondingly. Most of these insignificant CFs were detected during startup or shutdown stages. No startup or shutdown stages were detected in the Sunflow; and only 80% (167 out of 209) of Sunflow CFs were eliminated as insignificant.

Our models run 8-1000 times faster than the actual program (see Table 6.15). The actual speedup depends 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 run slower than models that execute few "long" CFs. Simulating intense synchronization operations slow down models as well. However, compared to benchmarking the actual system performance models offer two additional sources of speedup. 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. We plan to increase performance of our models by optimizing our modeling framework.

The approach we used for measuring the prediction accuracy of large applications is the same as we used for measuring the accuracy of models of small- to medium-size programs. Namely, we computed the relative error ε between predicted and measured performance across a number of different configurations. To get reliable performance measurements we performed three runs of both the actual program and its model in each configuration. All our experiments were executed on the hardware Setup I, which includes Intel Q6600 quad-core CPU, 4GB RAM, and 160 GB HDD. The computer was running Ubuntu Linux OS.

Below we describe our experiments in detail. We discuss the behavior of each program at the high level, describe the structure of its model, and present results of our experiments. Experimental results include predicted values of the program's performance metrics and their comparison to the values of performance metrics measured by running the actual program.

6.2.1 Sunflow: a 3D renderer

Sunflow is an open-source 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 [13].

The Sunflow offers a wide range of features including an API for embedding Sunflow functionality into the user programs, the API for creating scenes, the run-time compilation of shaders, the adaptive sampling of area light sources, the variable depth of field, and different types of renderers. It supports various file formats for importing scenes and textures, and for exporting data. Sunflow offers rich set of primitives for scene description, various types of cameras, surface shaders and modifiers, light sources and image filters. A full list of Sunflow features can be found on its website [13].

Although the Sunflow has a wide variety of options that can significantly alter the appearance of the resulting scene, at the high level it follows the same algorithm. Upon the start of the Sunflow the main thread parses the configuration options of the program, reads a scene specification from the disk, splits the frame into multiple tiles, and stores tile coordinates in the queue. These tiles correspond to tasks in the context of the multithreaded program. Then the main thread starts working threads. These working threads read tile coordinates from the queue, render the image tiles, and synthesizes the resulting image. The time required to render the image is the most important performance metric of Sunflow.

The high-level model of Sunflow is shown at the Figure 6.24. The `main_2` block is the model of the main thread. The `blkBLCKQUEUE1` represents the queue used to save tile coordinates. The working threads are represented by the `Thread_2_0` and `Thread_2_1` are the blocks in the high-level model. These blocks belong to the high-level formal model of Sunflow.

The high-level PERSIK model of Sunflow also contain blocks that simulate hardware: `blkDiskIOModule` that simulates the disk I/O subsystem, `blkCPUScheduler` that simulates the CPU and the thread scheduler, and `blkOSLimits` that simulates certain limitations imposed by the OS. These blocks formally belong to the low-level formal model of the system. But as we noticed earlier (see Section 4.2), placing these blocks into the high-level PERSIK model simplifies implementation and does not violate the semantics of the formal model.

Sunflow is a larger and more complex than the programs we studied before, thus we could not identify the exact function of each block in its thread models. Although the ability of the model to describe performance of the system in the programmer's terms would be a nice feature, it was not the main goal of our work. Indeed, the whole point of the automatic program analysis was to build models of large programs without knowing details of their structure. Nevertheless below we provide a brief description of key elements of these models.

The model of the main thread is depicted at the Figure 6.25 (top). The set of computational blocks on the left simulates initializations performed at the startup of the program, in particular – reading the program's configuration, loading the scene, and splitting the image into tiles. The loop `blkLOOP__45_46` fills the queue `blkBLCKQUEUE1` with tasks (image tiles). Then the thread relies on the barrier.await block `blk_49_50` to signal working threads that the image rendering can be started.

The model of the working thread is depicted at the Figure 6.25 (bottom). Upon its start the working thread waits on the barrier represented by the `blk_49_50` block until the main thread finishes initialization. Then the working thread enters the main loop, where it fetches tasks from the queue (this operation is modeled using the `blk37_38` block) and renders them. The block `blk70_37` represents the rendering operations itself. Although rendering is a complex operation accounting for a significant portion of the program's code, it involves only CPU computations and thus is represented with the single computation block.

The thread also contains the model of a synchronized region represented by the `blk73_74` and `blk75_76` blocks. These blocks corresponds to the $\langle 73, 74\rangle$ and $\langle 75, 76\rangle$ CFs, which represent entering and exiting the synchronized region respectively. In terms of our model, $\langle 73, 74\rangle$ and $\langle 75, 76\rangle$ CFs denote the data-guarding region (see Section 5.4.2). Computations within that region consume very little resources. As a result, these computations were considered as insignificant and not represented in the model.

The presence of data-guarding regions in the model does not alter the behavior of the model or affect prediction accuracy. However, their presence introduces additional complexity into the model and decreases the simulation performance. Thus in future we plan to introduce a more sophisticated logic for dealing with those regions. One approach is to detect data-guarding regions and eliminate them from the model altogether. An alternative is to decrease the value of the threshold used to decide if CFs residing within data guarding regions are insignificant. The consequence of this will higher chances for considering these CFs as significant and retaining them in the model. The latter option might theoretically improve the prediction accuracy in configurations with the high number of working threads, where the data-guarding regions can have a noticeable impact on performance. However, in our experiments we did not encounter scenarios when dropping CFs within the data-guarding regions caused a significant drop in the model accuracy.

Given the constant size of the image, the number of working threads is one factor that determines the performance of the Sunflow. The hardware used to deploy the program also has strong influence over its performance. In particular, the number of CPU cores poses a practical limit on the Sunflow performance.

We built the model of Sunflow v. 0.07 using a configuration with 2 working threads and 4 CPU cores. We used it to predict Sunflow performance in hardware Setup I 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 6.26 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 $\overline{\varepsilon} = 0.032$. Numeric results are presented in the table 6.16.

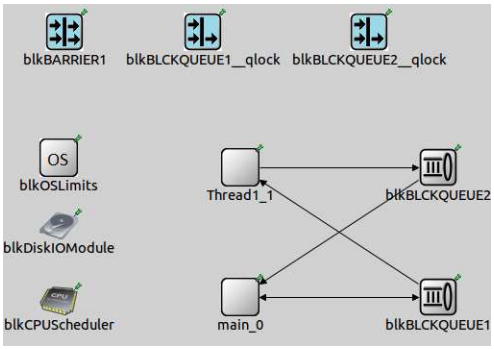

Figure 6.9: The high-level PERSIK model of the Moldyn

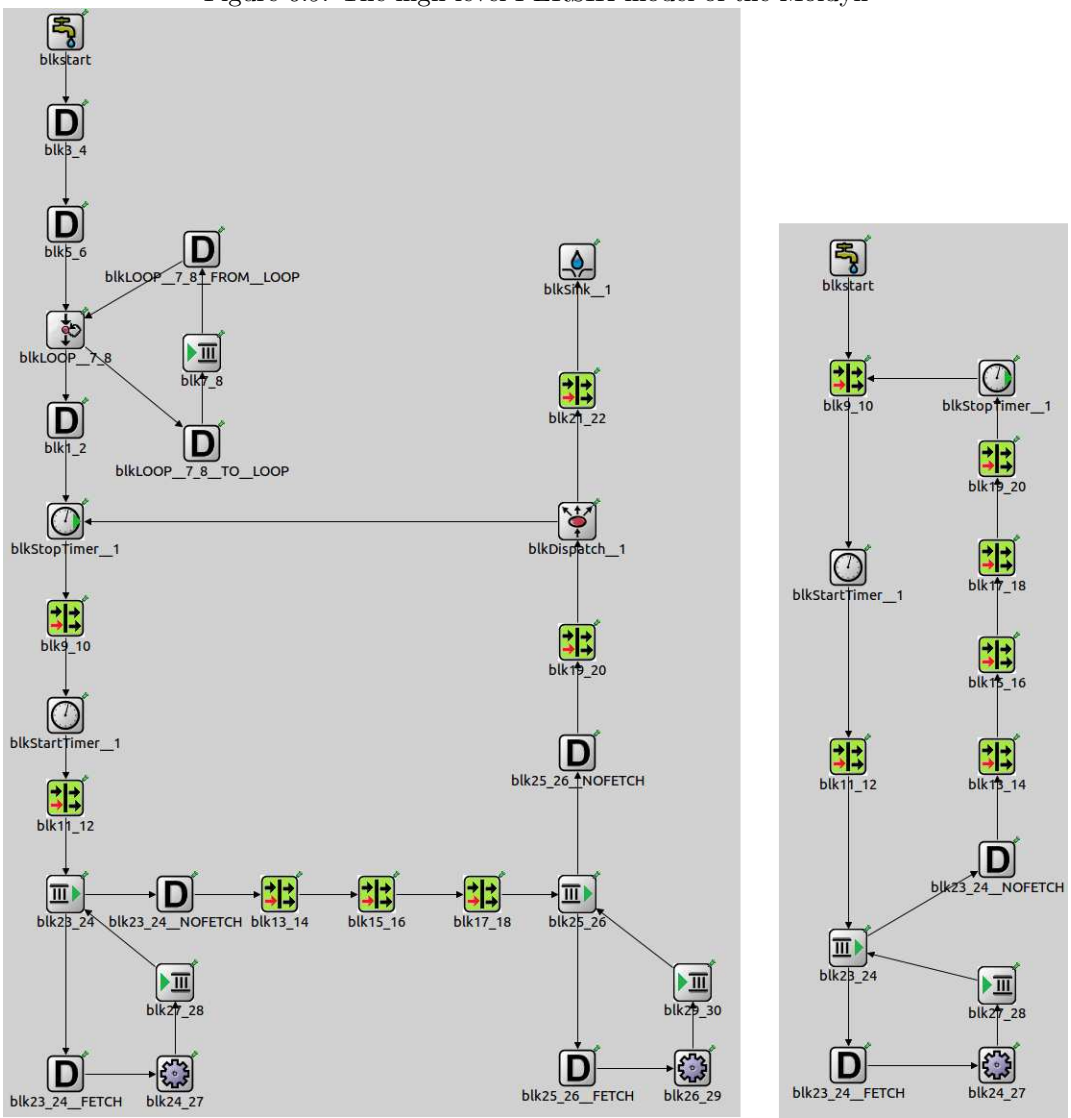

Figure 6.10: PERSIK models of the Moldyn threads: the main thread (left) and the working thread (right).

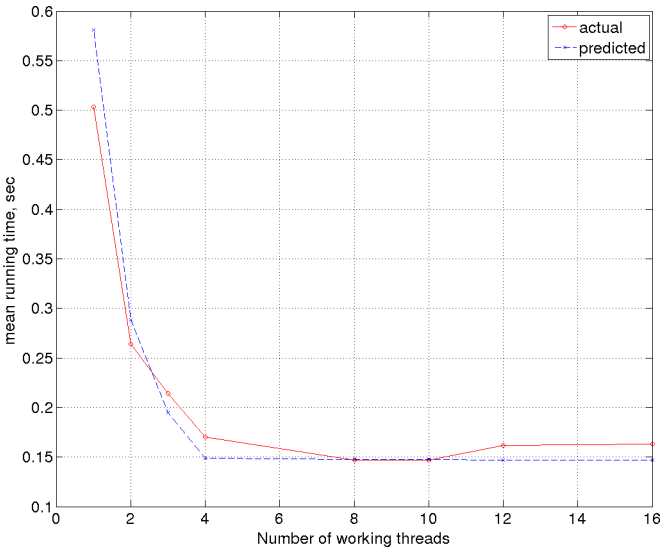

Figure 6.11: Experimental results for Moldyn

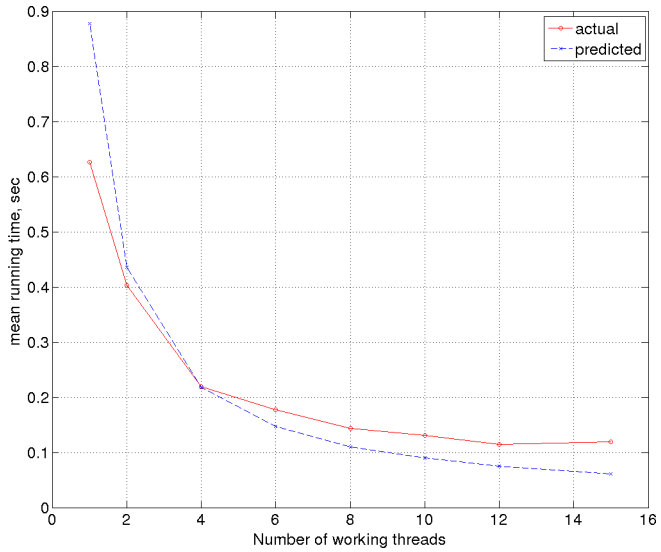

Figure 6.12: Experimental results for Moldyn

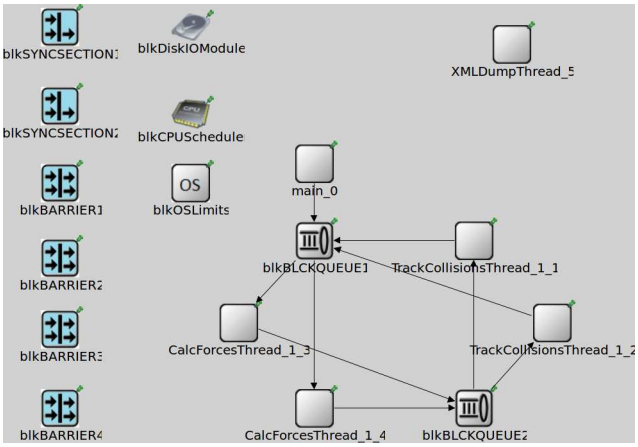

Figure 6.13: The high-level PERSIK model of the Galaxy

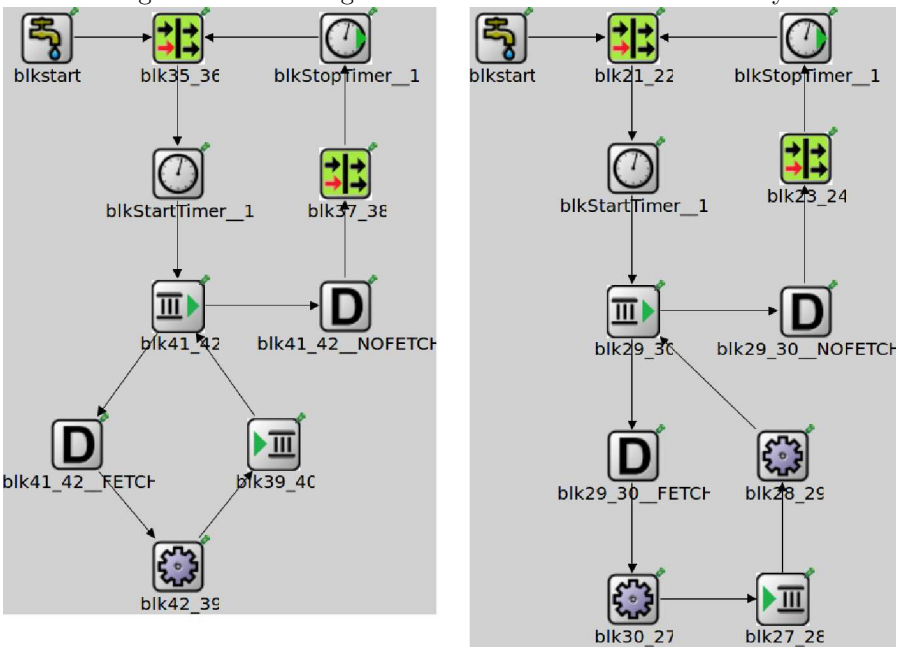

Figure 6.14: PERSIK models of the Galaxy threads: force thread (left) and collision thread (right).

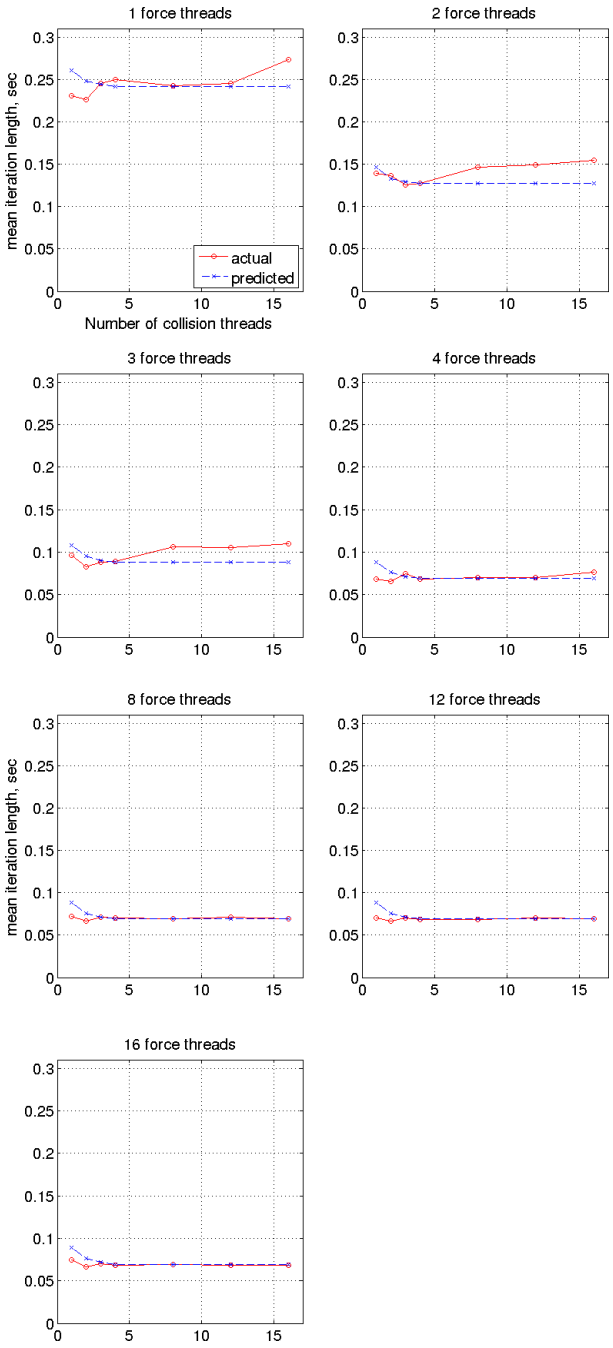

Figure 6.15: Predicted and measured iteration length for Galaxy program on a 4-core machine

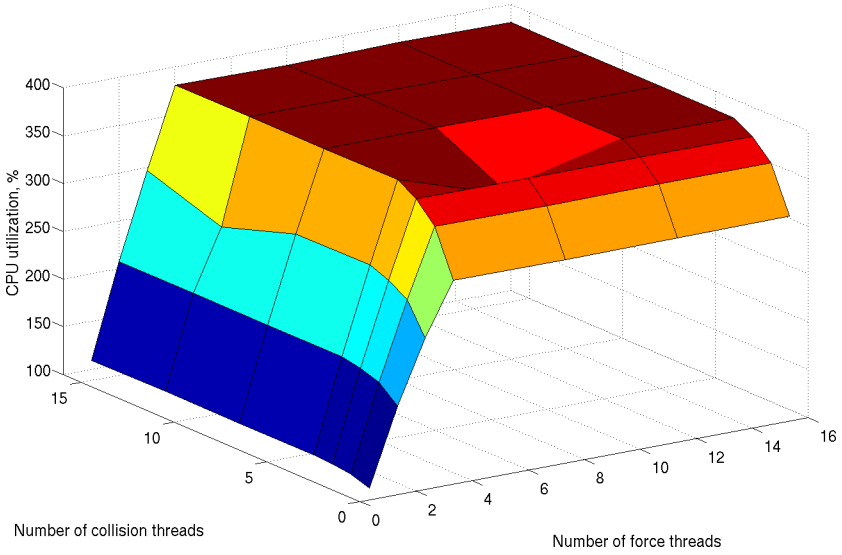

Figure 6.16: Predicted CPU for Galaxy utilization on a 4-core machine

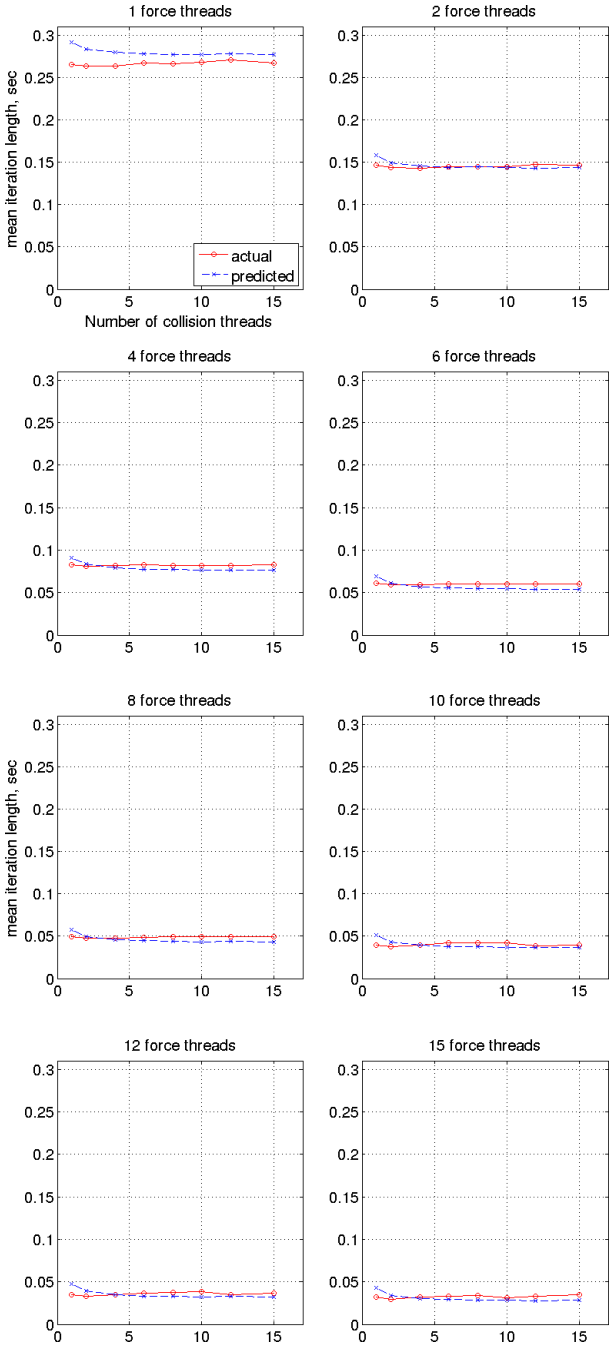

Figure 6.17: Predicted and measured iteration length for Galaxy program on a 16-core machine

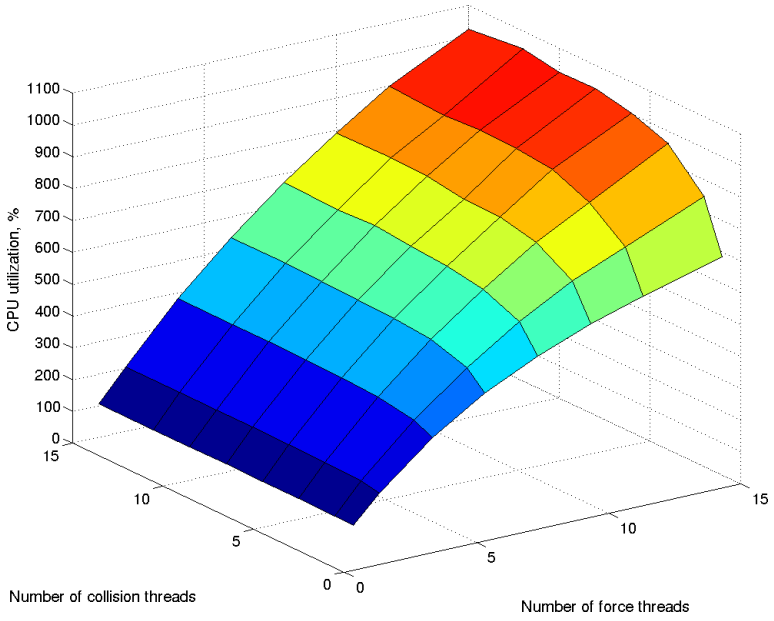

Figure 6.18: Predicted CPU for Galaxy utilization on a 16-core machine

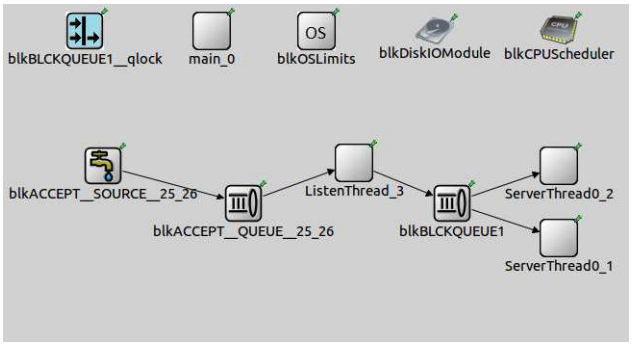

Figure 6.19: The high-level PERSIK model of the Tornado.

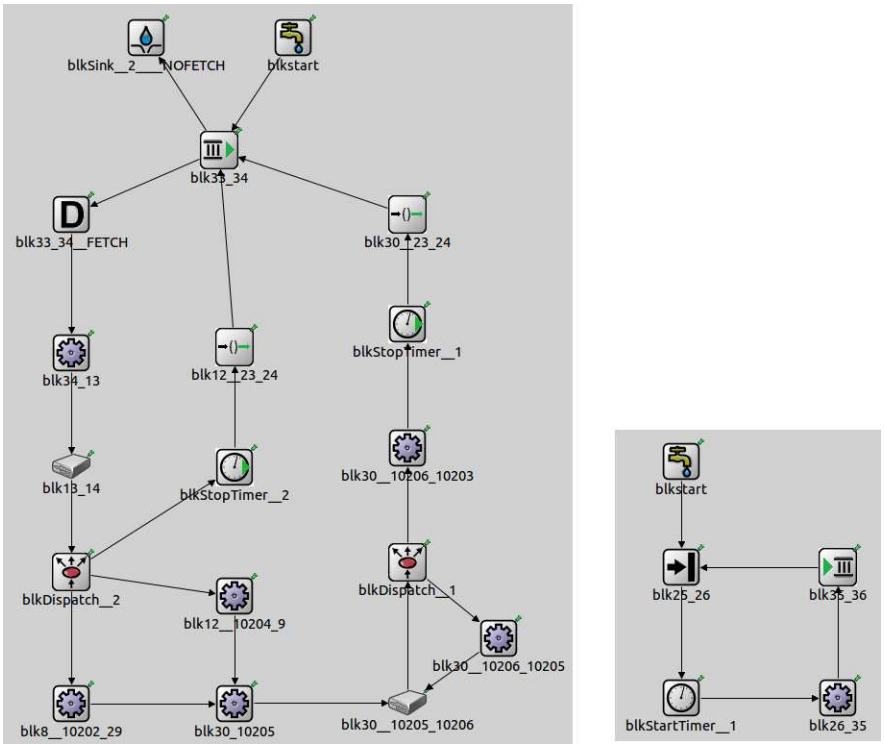

Figure 6.20: PERSIK model of Tornado threads.

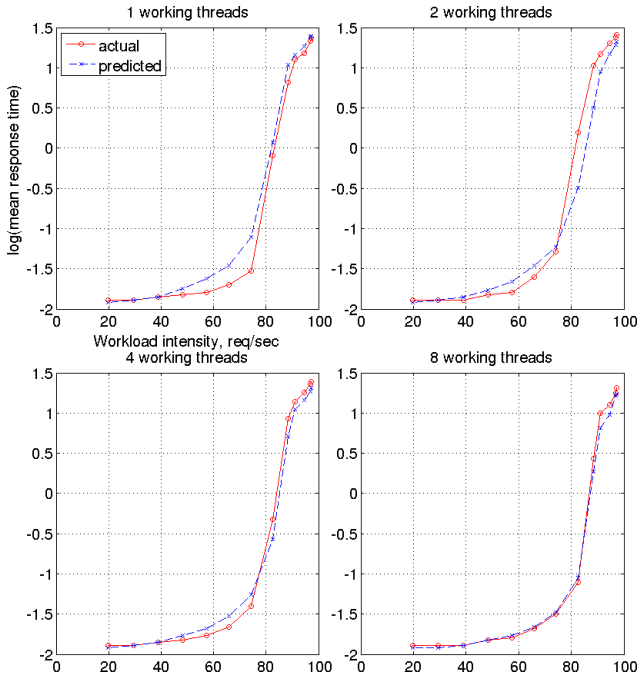

Figure 6.21: Predicted and measured response time for Tornado

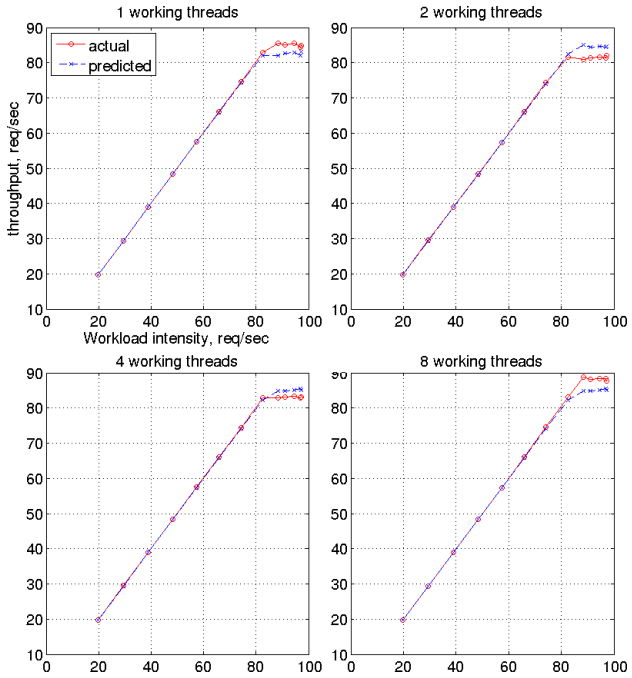

Figure 6.22: Predicted and measured throughput for Tornado

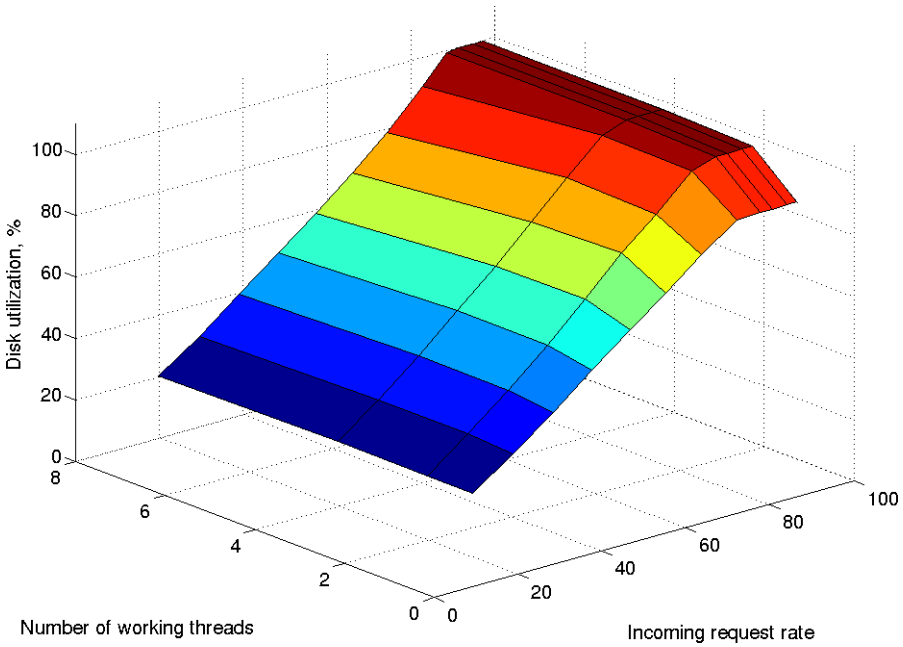

Figure 6.23: Experimental results for Tornado (hard drive utilization)

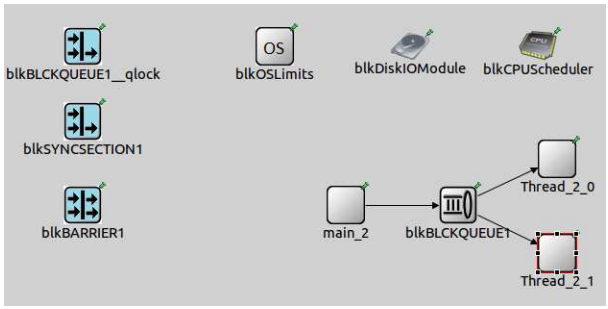

Figure 6.24: The high-level PERSIK model of the Sunflow.

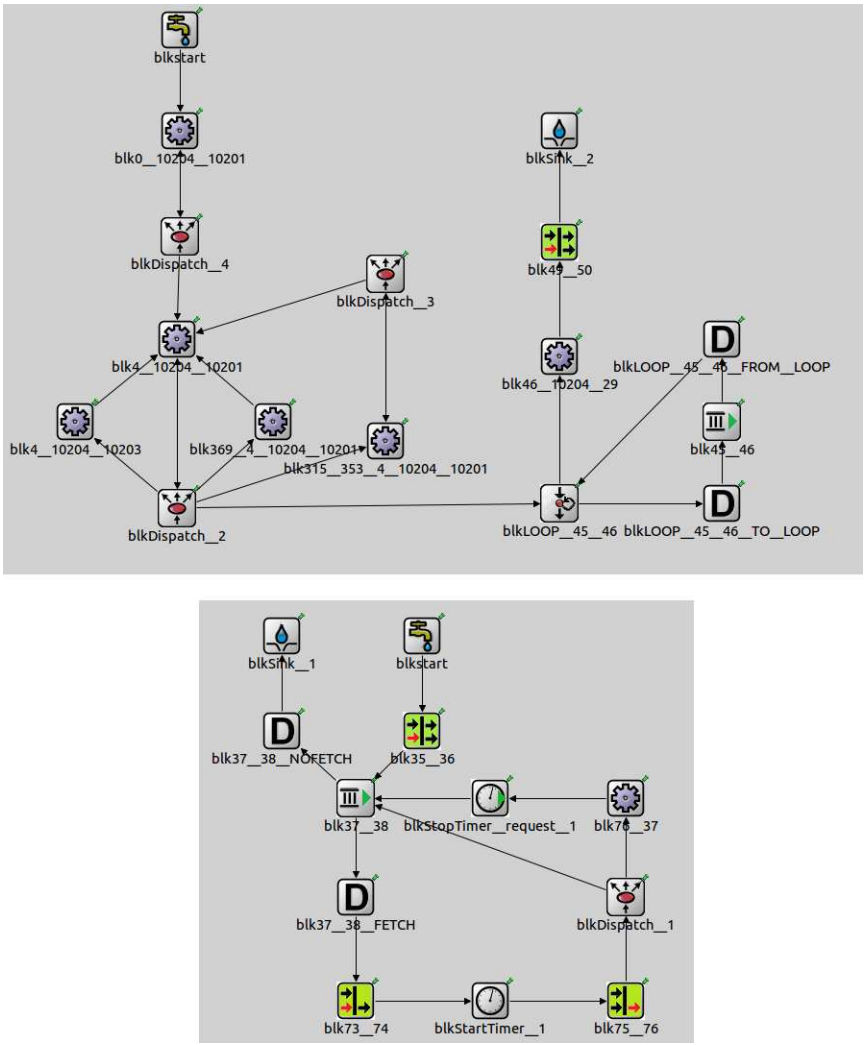

Figure 6.25: PERSIK models of Sunflow threads: the model of the main thread (top) and the model of the working thread (bottom).

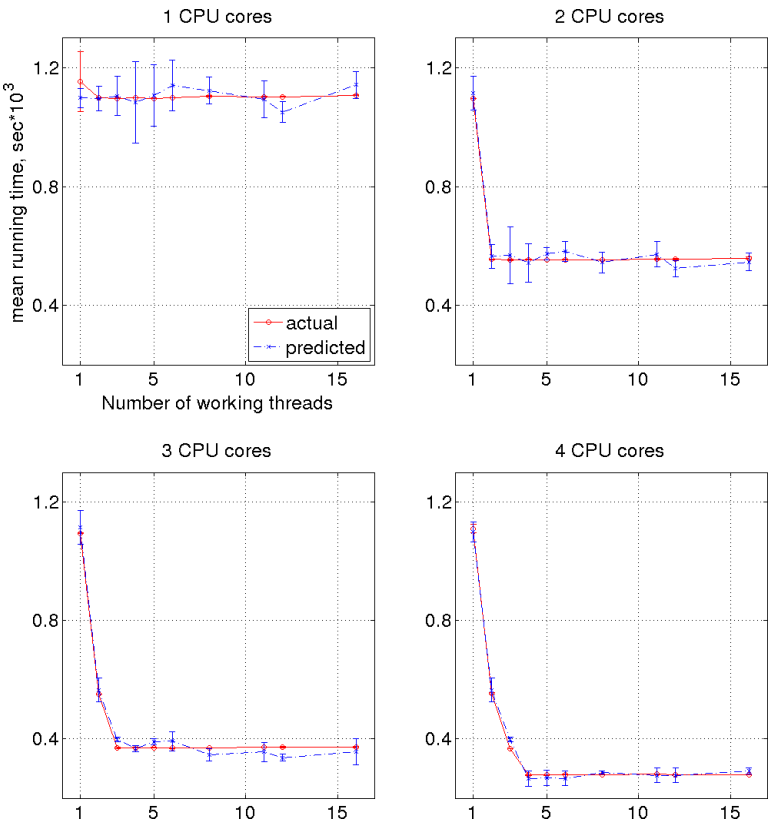

Figure 6.26: Predicted and measured response time for Sunflow program

Table 6.16: Predicted and measured running time for the Sunflow

Measured running time, sec

Num. CPUs	Number of working threads									
	1	2	3	4	5	6	8	11	12	16
1	1154.3	1099.8	1097.4	1099.3	1097.9	1099.6	1104.6	1102.9	1103.0	1108.1
2	1096.4	554.86	553.51	554.23	553.51	553.96	553.98	556.77	555.74	559.28
3	1094.7	552.44	371.07	370.48	371.02	368.96	369.35	371.73	373.33	372.39
4	1110.0	554.77	367.52	278.13	278.01	280.37	278.58	281.12	278.96	280.42

Predicted running time, sec

Num. CPUs	Number of working threads									
	1	2	3	4	5	6	8	11	12	16
1	1098.7	1096.9	1104.8	1084.1	1106.6	1140.4	1123.9	1094.5	1051.5	1143.0
2	1114.4	565.59	568.71	543.45	573.76	582.30	545.18	571.96	524.46	545.77
3	1114.4	565.59	398.67	367.87	389.86	392.42	346.79	356.04	337.13	357.14
4	1098.7	565.59	398.67	265.55	268.19	266.79	286.82	277.77	277.47	292.02

relative error

Num. CPUs	Number of working threads									
	1	2	3	4	5	6	8	11	12	16
1	0.048	0.003	0.007	0.014	0.008	0.037	0.017	0.008	0.047	0.032
2	0.016	0.019	0.027	0.019	0.037	0.051	0.016	0.027	0.056	0.024
3	0.018	0.024	0.074	0.007	0.051	0.064	0.061	0.042	0.097	0.041
4	0.010	0.019	0.085	0.045	0.035	0.048	0.030	0.012	0.005	0.041

Our experiments with Sunflow prove the capability of our framework to analyze complex industrial applications and to predict their performance across different hardware configurations. This does not yet translate into an accurate prediction of the program running on a totally different hardware as 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 using either an analytic model (e.g. by applying scaling coefficients) or by running microbenchmarks on the target architecture.

6.2.2 Tomcat: a web server and a servlet container

Apache Tomcat is a web server and Java servlet container [14]. Tomcat was initially developed by the Sun Microsystems as a reference implementation of Java Servlet and Java Server Page technologies, and later released under the Apache open-source license. In addition to being an application server, Tomcat provides a full support of all the HTTP protocol features, making it a very robust and feature-rich web server. Thanks to its reliability, flexibility, and high performance Tomcat is widely used in industry. It has been reported that more than the half of Fortune 500 companies use Tomcat in their business, including the Wallmart chain of retail stores, The Weather Channell, and E-Trade financial services company [15].

A core component of Tomcat is the set of `Connectors` objects. Connectors implement the external interfaces that allows clients access Tomcat using HTTP, AJP, and HTTPS protocols [35]. A connector accepts the incoming connection, perform some initial processing on it, and creates the `Request` object to represent the incoming request. The request object, which corresponds to the task in our formal model, is then passed to the `Processor` component for processing. The `Processor`

component in the Tomcat implements a thread pool, which can be configured both as a dynamic thread pool and as a producer-consumer pool with the fixed number of working threads.

The request processing itself is performed by the hierarchy of `Containers` objects called by the Processor. The top-level entry of this hierarchy is the `Engine` component. The engine contains one or more `Host` components that correspond to individual web applications. Hosts, in turn, contain the web application `Contexts`. Finally, Contexts contain `Wrappers` for individual servlets. The result of processing the request is represented by the `Response` object, which is returned through the same `Connector` used to accept the incoming network connection. Containers rely on a number of additional components, such as `Valves`, `Listeners`, `Loaders`, `Resources`, and `Security Realms`, for logging, class loading, authentication and authorization, and for other purposes.

In terms of threading, Tomcat uses up to 10 different thread pools to start up and shut down the program, to accept incoming connections, to process timeouts, to serve incoming HTTP requests, and for other purposes. Web applications hosted by the Tomcat can perform synchronization and start additional threads, further increasing complexity of the system.

Tomcat proved itself as a highly flexible web server. It provides a number of configuration options, which allow modifying the behavior of the Tomcat to the concrete customer scenario. In the unlikely case when the desired behavior cannot be achieved by adjusting values of the configuration parameters, the behavior of the Tomcat can be modified by developing a server extension and integrating it with the server's code using a plugin mechanism. Tomcat achieves high performance by means of object pooling, extensive caching of the results, and parallelizing the workload across different machines (e.g. using the Apache HTTP server to serve requests to static web pages, so Tomcat will be used only for hosting servlets).

However, flexibility, extensibility, and performance of Tomcat comes at the cost of the high internal complexity. Tomcat is a large and complex program, consisting of about 200000 lines of Java code and hundreds of Java classes. Considering the size and complexity of the Tomcat, manually building and configuring performance model of Tomcat is not feasible. The only way to build a performance model of Tomcat and servlets hosted by it is to use an automatic model generator.

We relied on our framework to build the performance model of Tomcat in two different setups. In the first setup the Tomcat was deployed as a pure web server, serving requests to the static web pages. In the second setup we used Tomcat as a servlet container. In this setup Tomcat was hosting a web application that used `iText` library to generate the PDF document and send it to the user. The functionality of Tomcat was very different in both of these setups. The code of the system in the servlet container was different as well, since it included the code of the web application. As a result, simulating performance of Tomcat in these different setups required building two different models.

We will describe out experimentation with Tomcat performance models in a following way. First we will describe parts of the Tomcat model that are common for both of experimental setups. Then we will discuss each of these setups in more detail. This will include the description of the difference

between the corresponding models as well as the description of the experiments themselves.

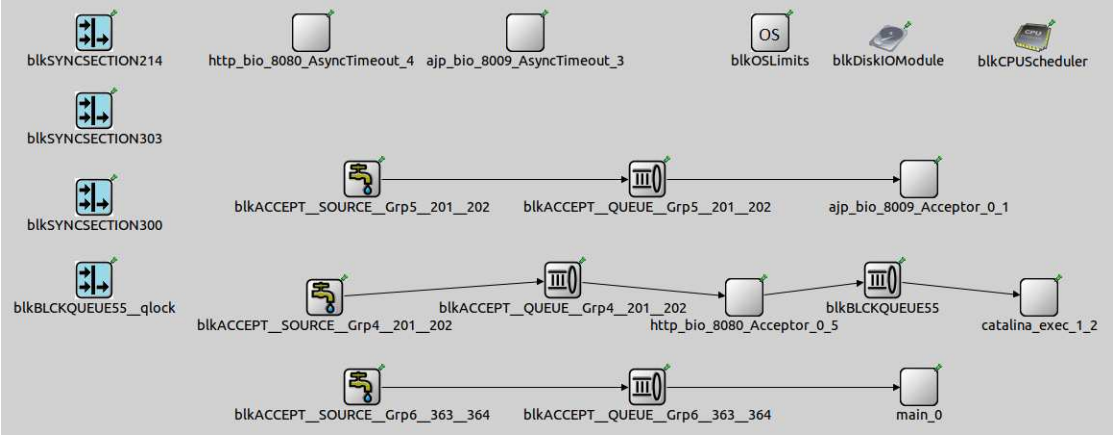

Figure 6.27: The high-level PERSIK model of the Tomcat.

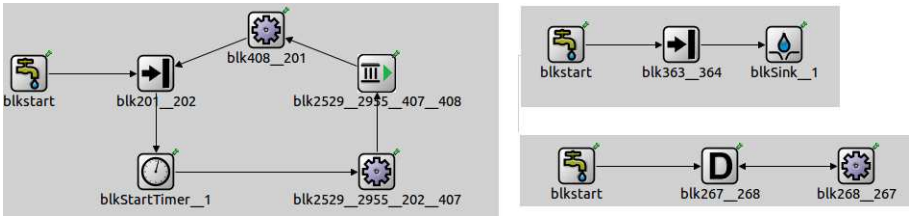

Figure 6.28: The PERSIK model of Tomcat threads. Left: the Acceptor thread for accepting incoming HTTP connections. Upper right: the main thread. Lower right: the connection timeout thread.

The Figure 6.27 depicts the high-level model of Tomcat with one working thread. This model remains unchanged across the different setups. From the point of performance modeling, the most important threads in it are the HTTP acceptor thread (represented by the the `http_bio_8080_Acceptor` block in the high-level model) and the working thread (represented by the `catalina_exec_1_2` block).

The acceptor thread listens for incoming HTTP connections, which are the main type of requests processed by Tomcat in our experiments. The incoming HTTP connections are generated by the `blkACCEPT__SOURCE__Grp4__201__202` source block. The source block passes the connection to the `blkACCEPT__QUEUE__Grp4__201__202` block, which represents the queue used by the operating system to store incoming TCP connections.

Incoming connections are processed by the `http_bio_8080_Acceptor` HTTP acceptor thread, whose mid-level model is depicted at the Figure 6.28 (left). The `blk201__202` represents the Socket.accept operation, which fetches the incoming connections from the `blkACCEPT__QUEUE__Grp4__201__202` queue. Once the connection has been accepted, the `blk2529__2955__202__407` block performs computations pertaining to generating the `HTTPRequest` object. The block `blk2529__2955__407__408` sends the `HTTPRequest` object to the `blkBLKQUEUE55`, which is the part of the

Processor component in the Tomcat.

The working thread fetches tasks from blkBLKQUEUE55 queue and processes them. The behavior of the working thread is highly specific to the setup in which the Tomcat is running. Models of the Tomcat working threads will be discussed below when we describe different Tomcat setups.

The ajp_bio_8009_Acceptor_0_1 is an AJP acceptor thread that processes the incoming AJP connections. AJP (Apache JServ Protocol) is the binary protocol that can pass the HTTP requests from a web server to an application server. The purpose of the AJP is to distribute processing of the static and dynamic web content across different computers. The static content can be processed by the specialized web server such as Apache HTTP server, while dynamic content is processed by Tomcat. Using such distributed processing can reduce load on Tomcat and thus improve overall performance of the system.

The blkACCEPT__SOURCE__Grp5__201__202 block of the high-level model represents the source of AJP connections. Incoming AJP connections should be placed in the blkACCEPT__QUEUE__Grp4_-_201__202 queue, from which the ajp_bio_8009_Acceptor_0_1 thread will fetch them and process. However, in our setup we used Tomcat as a standalone web server that did not receive any AJP connections. As a consequence, the blkACCEPT__SOURCE__Grp5__201__202 block did not generate any requests and the AJP acceptor thread did not perform any operations.

The main_0 block is the model of the Tomcat's main thread, whose function is to manage the server lifecycle. Upon the startup the main thread initializes the web server. Server initialization is a complicated process, considering the size and complexity of the Tomcat. Once the server is fully initialized, the main thread opens a network socket, and listens on that socket for the shutdown command. When the shutdown command is received, the main thread shuts down the Tomcat gracefully. In our model the source of shutdown command is represented by the blkACCEPT__-SOURCE__Grp6__363__364 source block.

The mid-level model of the main thread is shown at the Figure 6.28 (upper right). The main thread perform very complex operations during the startup phase of the program, and representing these operations in the model would require using thousands of code fragments. However, all these operations have no influence on the performance of the server during its request processing phase, when the Tomcat actually serves HTTP requests. Consequentially, the corresponding code fragments were omitted from the model during the model simplification (see Section 5.4.2 for details). As a result, the model of the main thread contains only the blk363_364 block, which represents waiting for the incoming connection on the shutdown socket.

The http_bio_8080_AsyncTimeout_4 thread is used to track connection timeouts for the HTTP connections. The thread wakes up every second, checks if any connection has timed out, and goes to sleep again. Correspondingly the model of this thread is fairly simple (see Figure 6.28, lower right). The blk267__268 block is a Thread.Sleep operation, while the blk268__267 block represent computations performed when the thread checks for connection timeouts.

Similarly, the ajp_bio_8080_AsyncTimeout_3 thread tracks timeouts for AJP connections. The

structure and the behavior of its model are identical to the model of the `http_bio_8080_-AsyncTimeout_4` thread.

Tomcat as a web server

In this setup we used Tomcat to host about 600000 Wikipedia web pages. A client computer (an Intel 2.4 GHz dual-core CPU, 4 GB RAM) connected to the web server over the 100 MBit LAN was used to simulate the incoming requests. The client was running the `http_load` software which accesses web pages from a supplied list with a specified IRR.

The performance of Tomcat server is influenced by two parameters: the incoming request rate (IRR), which represents the intensity of the workload, and the number of working threads. The performance of the web server is characterized by two main metrics: its response time R and throughput T.

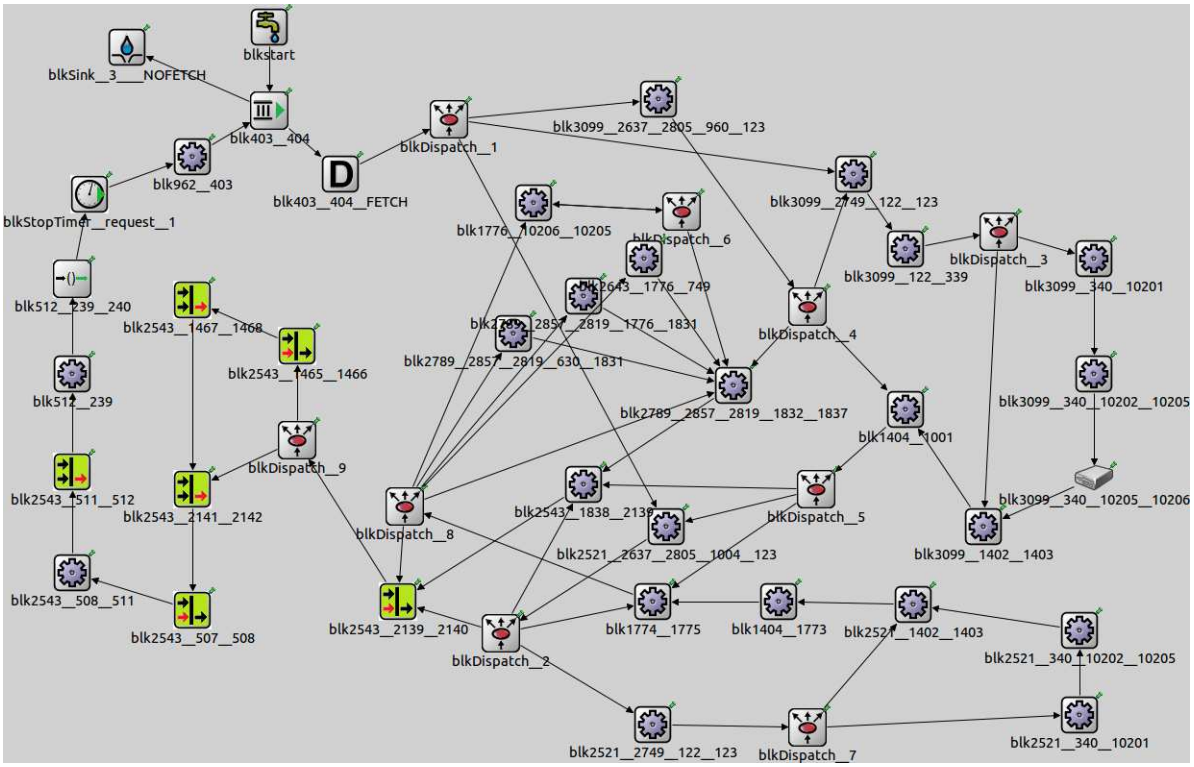

Figure 6.29: The PERSIK model of the Tomcat thread for processing HTTP requests.

The model of the web server was built using a configuration with workload intensity 92 requests per second (req/s) and 1 working thread. The model of the working thread in the web server setup is depicted at the Figure 6.29. Because of high complexity of the Tomcat we could not precisely determine the functionality of each of the model's block. Although such analysis would improve the understanding of the system and facilitate the debugging of the model, it was never the ultimate goal of our model. Instead, our primary focus was automatically building models that could predict

the performance of the Tomcat.

Nevertheless, we could determine the functionality of certain blocks that constitute the mid-level model of the Tomcat working thread. In particular, the `blk403_404` fetches the `HTTPRequest` object from the `blkBLKQUEUE55` queue. Blocks `blk3099_2637_2805_960_123`, `blk3099_2749_122_123`, and `blk3099_122_339` are executed after fetching the request object. They manage the Tomcat cache, where it stores the contents of the web pages accessed previously. The block `blk3099_340_-_10205_10206` reads the web page from the disk. Finally, block `blk512_239_240` sends the data back to the client and closes the socket.

We ran Tomcat with with 1,2,3,4,5,6,8 and 10 working threads and workload intensity ranging from 48.30 to 156.25 requests per second (measured on the server side). During each run 10000 requests were issued. The workload is characterized with the large number of disk I/O operation.

The prediction results for the response time R are depicted at the Figure 6.30. The relative prediction error $\varepsilon(R)$ varies within $(0.003, 2.452)$ with average error $\overline{\varepsilon}(R) = 0.269$. The numeric results for the response time across all the configurations are presented in the table 6.17.

The relatively high error terms in predicting the R are attributed to the fluctuations of the page cache hit rate k. When measured across all the experimental configurations, the mean number of low-level I/O operations issued by the block `blk3099_340_10205_10206` is $\overline{k} = 0.755$ with standard deviation $\sigma(k) = 0.046$. From a statistical point of view this mean that in 95% of cases the true value of k will vary between $(0.663, 0.847)$. 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 hypothesis we introduced a 15% artificial bias in the number of I/O operations k performed by Tomcat. This resulted in increasing the relative error to $\varepsilon(R) \in (0.015, 3.109)$ with $\overline{\varepsilon}(R) = 0.882$ (a single outlier configuration resulting in $\varepsilon(R) = 49.446$ was removed from the analysis). We believe this experiment proves our assumption about the causes of the errors in predicting R. Moreover, it illustrates difficulties associated with predicting the variability inherent to the hard drive I/O operations.

The prediction for throughput T is shown at the Figure 6.31. The relative prediction error $\varepsilon(T)$ varies in $(0.001, 0.087)$ with average error $\overline{\varepsilon}(T) = 0.0121$. However, in non-saturated configurations throughput is roughly equal to the incoming request rate. Thus a more informative metric for the accuracy of throughput prediction is the relative error for saturated configurations, which is $\overline{\varepsilon}(T_{sat}) = 0.027$. The numeric results for the throughput across all the configurations are presented in the table 6.18.

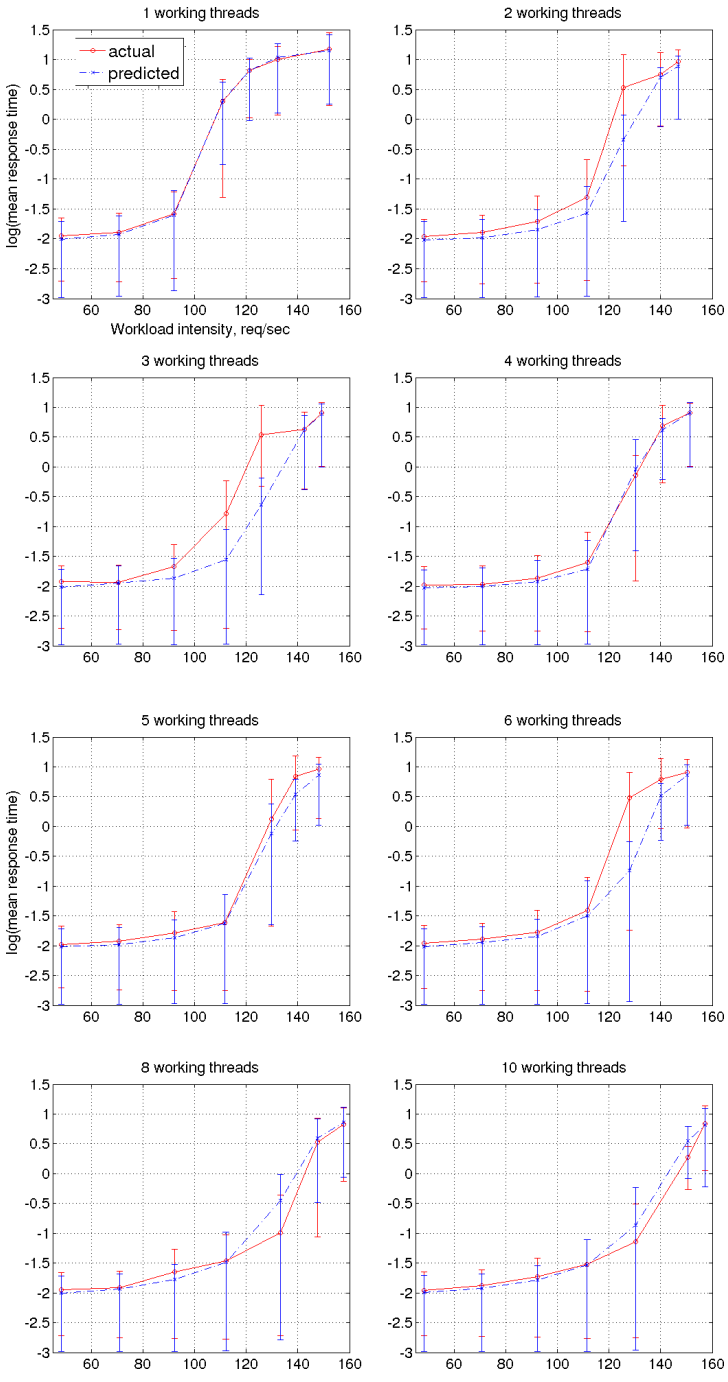

Figure 6.30: Predicted and measured response time for the Tomcat in a web server configuration.

Table 6.17: Predicted and measured response time for the Tomcat in a web server configuration.

Measured response time, sec

Num. threads	Average workload intensity, req/sec						
	48.29	70.74	92.10	111.8	127.5	140.4	151.8
1	0.011	0.013	0.026	1.995	6.575	10.14	14.93
2	0.012	0.012	0.022	0.166	3.471	4.313	8.065
3	0.010	0.012	0.017	0.025	1.346	6.810	9.292
4	0.011	0.012	0.022	0.035	0.102	3.377	6.772
5	0.010	0.012	0.017	0.025	1.346	6.810	9.292
6	0.011	0.013	0.017	0.039	3.048	6.247	8.135
8	0.011	0.012	0.022	0.035	0.102	3.377	6.772
10	0.011	0.013	0.019	0.030	0.073	1.865	6.951

Predicted response time, sec

Num. threads	Average workload intensity, req/sec						
	48.29	70.74	92.10	111.8	127.5	140.4	151.8
1	0.010	0.012	0.025	2.055	6.558	10.93	14.12
2	0.010	0.011	0.014	0.028	0.235	4.302	7.699
3	0.010	0.010	0.013	0.023	0.754	3.505	7.358
4	0.010	0.011	0.017	0.032	0.352	3.898	7.284
5	0.010	0.010	0.013	0.023	0.754	3.505	7.358
6	0.010	0.011	0.014	0.032	0.183	3.305	7.199
8	0.010	0.011	0.017	0.032	0.352	3.898	7.284
10	0.010	0.012	0.016	0.030	0.136	3.539	6.537

relative error

Num. threads	Average workload intensity, req/sec						
	48.29	70.74	92.10	111.8	127.5	140.4	151.8
1	0.130	0.058	0.055	0.030	0.003	0.079	0.054
2	0.184	0.034	0.363	0.832	0.932	0.002	0.045
3	0.082	0.111	0.186	0.045	0.440	0.485	0.208
4	0.121	0.050	0.246	0.063	2.452	0.154	0.075
5	0.082	0.111	0.186	0.045	0.440	0.485	0.208
6	0.132	0.124	0.163	0.190	0.940	0.471	0.115
8	0.121	0.050	0.246	0.063	2.452	0.154	0.075
10	0.087	0.078	0.125	0.020	0.859	0.898	0.059

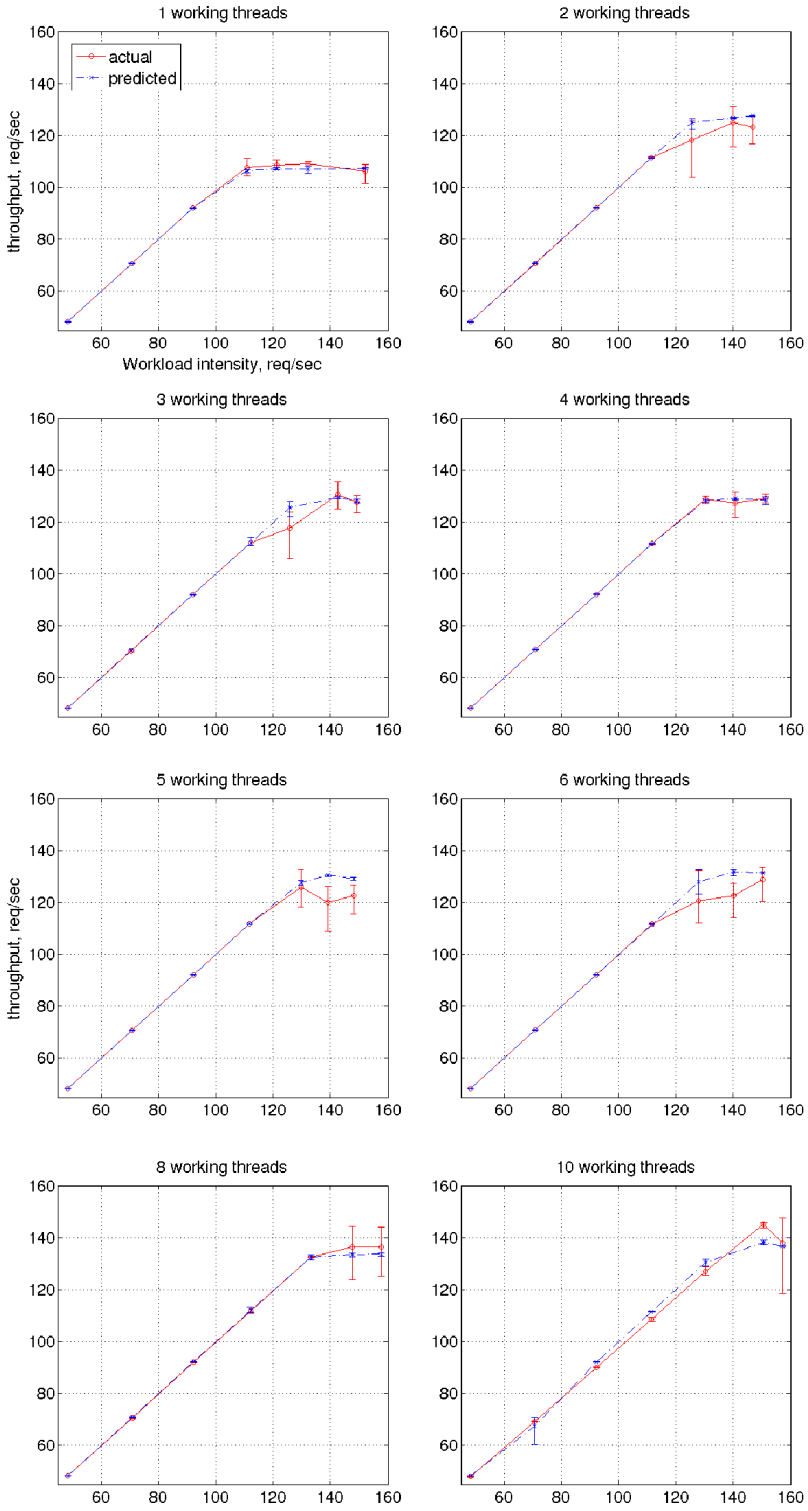

Figure 6.31: Predicted and measured throughput for the Tomcat in a web server configuration.

Table 6.18: Predicted and measured throughput for the Tomcat in a web server configuration.

Measured troughput, req/sec

Num. threads	Average workload intensity, req/sec						
	48.29	70.74	92.10	111.8	127.5	140.4	151.8
1	48.24	70.77	91.99	107.6	108.6	109.2	106.1
2	48.25	70.52	91.96	112.2	117.6	130.4	127.7
3	48.33	70.79	92.12	111.7	125.9	120.0	122.7
4	48.26	70.71	92.05	111.7	132.6	136.4	136.5
5	48.33	70.79	92.12	111.74	125.94	119.99	122.72
6	48.33	70.82	92.13	111.69	120.59	122.56	128.68
8	48.26	70.71	92.05	111.73	132.63	136.38	136.51
10	47.95	69.24	90.10	108.66	126.99	145.08	137.88

Predicted troughput, req/sec

Num. threads	Average workload intensity, req/sec						
	48.29	70.74	92.10	111.8	127.5	140.4	151.8
1	48.25	70.78	92.02	106.4	107.1	107.1	107.3
2	48.26	70.69	92.00	112.2	125.5	129.3	128.1
3	48.34	70.81	92.13	111.8	127.5	130.4	128.9
4	48.32	70.87	92.25	112.1	132.5	133.6	133.7
5	48.34	70.81	92.13	111.78	127.48	130.38	128.92
6	48.30	70.86	92.17	111.57	127.88	131.69	131.48
8	48.32	70.87	92.25	112.09	132.51	133.58	133.68
10	48.31	67.30	92.19	111.54	130.38	138.03	136.70

relative error

Num. threads	Average workload intensity, req/sec						
	48.29	70.74	92.10	111.8	127.5	140.4	151.8
1	0.000	0.000	0.000	0.011	0.014	0.019	0.011
2	0.000	0.002	0.000	0.000	0.067	0.009	0.004
3	0.000	0.000	0.000	0.000	0.012	0.087	0.051
4	0.001	0.002	0.002	0.003	0.001	0.021	0.021
5	0.000	0.000	0.000	0.000	0.012	0.087	0.051
6	0.001	0.000	0.000	0.001	0.060	0.074	0.022
8	0.001	0.002	0.002	0.003	0.001	0.021	0.021
10	0.007	0.028	0.023	0.026	0.027	0.049	0.009

Our model correctly predicts that the number of working threads has a minor impact on performance of Tomcat in this setup. When executed with 1 working thread, Tomcat saturates at 110 req/s, but with 8 threads it saturates at 130 req/s (see Figure 6.31). The reason for this behavior lays in the nature of the workload and the hardware configuration of the experimental PC, which is equipped with a single hard drive and a quad-core CPU.

The workload of the Tomcat in the web server setup is mostly disk I/O-bound. In particular, out of all the computational resources consumed during the processing of the HTTP request, 81% constitute the disk I/O bandwidth (see Figure 6.32 and Table 6.19) and remaining 19% are the CPU time (see Figure 6.20 and Table 6.20). Since the server is equipped with only one hard drive, this hard drive becomes the point of contention in the system. Any increase in the number of working threads is compensated by the increase in the average execution time for each I/O request. At the same time, CPU computations are parallelized across four CPU cores, resulting in small but noticeable increase in the server performance.

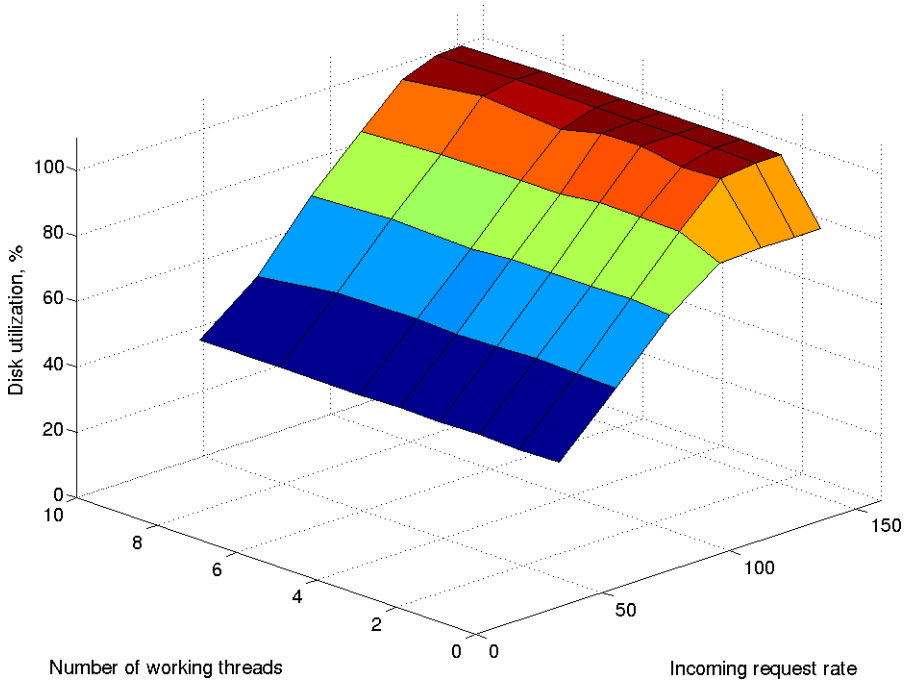

Figure 6.32: Predicted utilization of the hard drive for the Tomcat in a web server configuration.

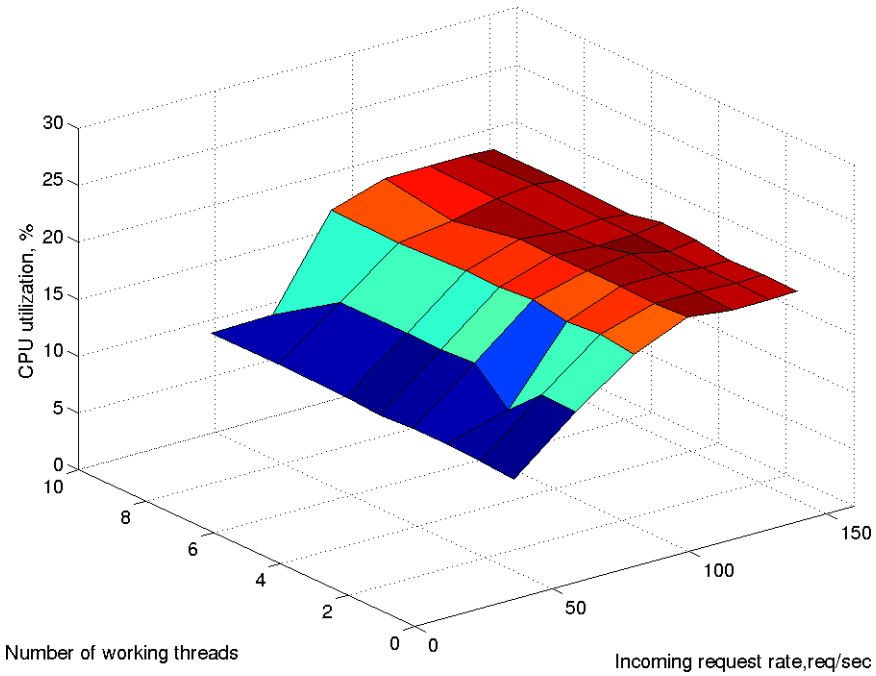

Figure 6.33: Predicted utilization of the CPU for the Tomcat in a web server configuration.

Tomcat as a servlet container

Although Tomcat can serve requests to static web pages, more frequently it is used as a servlet container. In order to ensure that performance models are capable of predicting performance of the

Table 6.19: Predicted utilization of the hard drive for the Tomcat in a web server configuration.

Measured hard drive utilization for the Tomcat
in a web server configuration, percent

Num. threads	Average workload intensity, req/sec						
	48.29	70.74	92.10	111.8	127.5	140.4	151.8
1	37.83	52.63	70.44	82.14	81.89	80.74	80.53
2	38.45	58.50	73.04	89.79	98.77	98.68	98.51
3	41.43	51.20	77.66	91.24	99.88	99.47	99.85
4	36.97	52.52	71.88	87.34	99.01	99.85	99.86
5	35.23	54.29	72.70	86.38	99.06	99.15	99.11
6	37.74	54.02	71.16	88.63	98.64	98.96	98.85
8	38.05	51.98	76.72	89.35	97.33	99.06	98.57
10	36.76	53.17	71.61	85.22	97.02	100.03	99.98

Predicted hard drive utilization for the Tomcat
in a web server configuration, percent

Num. threads	Average workload intensity, req/sec						
	48.29	70.74	92.10	111.8	127.5	140.4	151.8
1	36.23	53.08	70.02	80.66	81.17	81.12	81.01
2	35.97	53.21	70.62	86.35	98.17	99.54	99.50
3	36.30	53.69	70.33	86.65	97.65	100.01	100.02
4	35.99	53.03	70.18	86.86	99.77	100.01	100.01
5	36.24	52.73	70.11	85.13	99.55	100.00	100.00
6	35.92	53.34	69.24	85.12	96.59	100.00	100.02
8	36.00	52.94	70.14	84.83	98.53	99.97	100.01
10	35.82	49.65	68.84	83.50	95.30	99.07	99.48

relative error

Num. threads	Average workload intensity, req/sec						
	48.29	70.74	92.10	111.8	127.5	140.4	151.8
1	0.042	0.009	0.006	0.018	0.009	0.005	0.006
2	0.064	0.091	0.033	0.038	0.006	0.009	0.010
3	0.124	0.049	0.094	0.050	0.022	0.005	0.002
4	0.027	0.010	0.024	0.006	0.008	0.002	0.002
5	0.029	0.029	0.036	0.015	0.005	0.009	0.009
6	0.048	0.013	0.027	0.040	0.021	0.010	0.012
8	0.054	0.018	0.086	0.051	0.012	0.009	0.015
10	0.026	0.066	0.039	0.020	0.018	0.010	0.005

Table 6.20: Predicted utilization of the CPU for the Tomcat in a web server configuration.

Predicted CPU utilization for Tomcat
in a web server configuration, percent

Num. threads	Average workload intensity, req/sec						
	48.29	70.74	92.10	111.8	127.5	140.4	151.8
1	8.40	12.79	16.46	18.49	17.98	18.03	18.09
2	8.61	12.92	16.85	18.26	18.18	18.06	18.17
3	8.78	10.20	16.57	18.43	18.45	18.04	18.09
4	8.70	13.00	17.11	18.36	18.78	18.14	18.43
5	8.49	12.74	16.86	18.34	18.11	18.26	18.61
6	8.74	12.82	16.92	18.37	18.01	18.21	17.98
8	8.86	12.78	16.60	17.25	18.12	18.51	18.15
10	8.83	8.99	16.70	18.20	18.21	18.26	18.16

server in this realistic configuration, we used Tomcat to host a web application. The web application reads a random passage from the King James bible, formats the passage, and converts it to PDF using the iText [16] 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. The model of the working thread in the servlet container setup is depicted at the Figure 6.34. As a consequence of hosting a web application the code of the system under the study has become more voluminous and complex. Nevertheless, the mid-tier model of the working thread has become noticeably simpler. In fact, all the functionality related to creating PDF document is captured by four computation blocks in the model: `blk2927_-_1460_781`, `blk2687_1460_2243`, `blk3411_2851_3357_1460_1935`, and `blk3917_3807_3291_-_1460_659`. We believe such simplification occurs because the Tomcat no longer manages the cache for static web pages and does not perform disk I/O operations. And all the operations associated with generating a PDF document, although complex, involve only CPU-bound computations. These computations are represented as a small number of CPU code fragments, which, in turn, are translated into the computational blocks in the thread model.

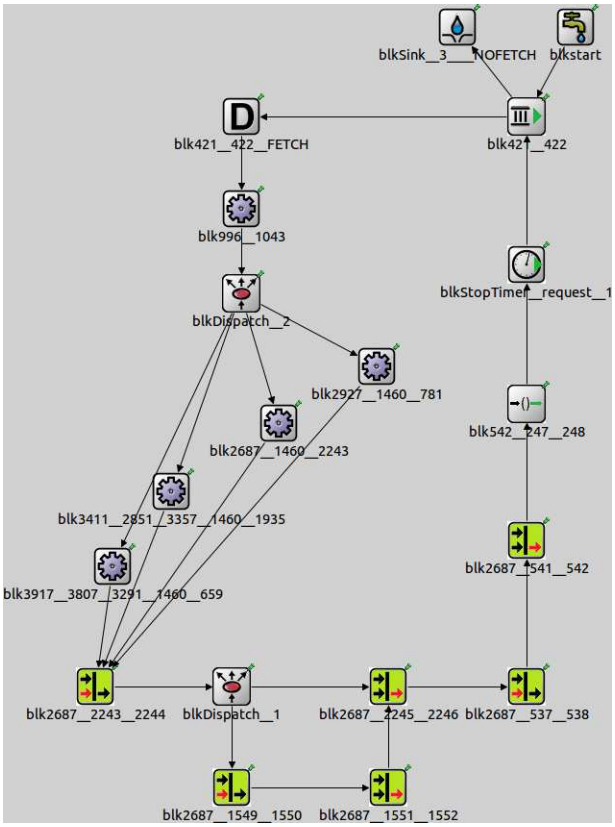

Figure 6.34: The PERSIK model of the Tomcat request processing thread in a servlet container configuration.

We ran Tomcat with with 1,2,3,4,5,6,8 and 10 working threads and workload intensity ranging from 19.67 to 132.68 requests per second. The workload intensity was measured on the server side and also averaged for each number of working threads. During each run 10000 requests were issued.

The prediction results for the response time R are depicted at the Figure 6.35. The relative

Table 6.21: Predicted and measured response time for the Tomcat in a servlet container configuration.

Measured response time, sec

Num. threads	Average workload intensity, req/sec							
	19.67	29.10	38.16	46.87	83.09	102.99	125.44	132.68
1	0.045	51.591	104.267	136.564	204.969	238.726	246.826	255.443
2	0.045	0.046	0.058	10.868	71.546	92.162	109.935	113.411
3	0.046	0.047	0.048	0.052	25.625	44.379	57.593	64.686
4	0.047	0.048	0.049	0.052	9.601	26.190	36.521	43.847
5	0.048	0.047	0.049	0.051	5.415	19.499	28.585	33.817
6	0.048	0.049	0.051	0.052	3.684	17.698	26.663	32.624
8	0.048	0.048	0.050	0.053	3.715	17.175	24.432	29.652
10	0.047	0.048	0.050	0.054	3.050	15.490	21.712	28.889

Predicted response time, sec

Num. threads	Average workload intensity, req/sec							
	19.67	29.10	38.16	46.87	83.09	102.99	125.44	132.68
1	0.053	61.571	99.727	122.127	167.665	186.862	194.538	198.092
2	0.047	0.047	0.050	11.923	54.125	70.153	78.952	83.793
3	0.047	0.047	0.047	0.048	19.814	32.025	41.218	45.096
4	0.047	0.047	0.047	0.048	2.728	13.907	23.010	28.295
5	0.047	0.047	0.047	0.048	3.478	16.380	23.524	28.304
6	0.047	0.047	0.047	0.048	3.720	16.013	23.053	28.403
8	0.047	0.047	0.047	0.048	3.580	16.470	23.142	28.156
10	0.047	0.047	0.048	0.048	4.194	17.239	23.278	28.635

relative error

Num. threads	Average workload intensity, req/sec							
	19.67	29.10	38.16	46.87	83.09	102.99	125.44	132.68
1	0.172	0.193	0.044	0.106	0.182	0.217	0.212	0.225
2	0.037	0.019	0.136	0.097	0.243	0.239	0.282	0.261
3	0.018	0.010	0.020	0.069	0.227	0.278	0.284	0.303
4	0.009	0.008	0.033	0.082	0.716	0.469	0.370	0.355
5	0.006	0.001	0.036	0.069	0.358	0.160	0.177	0.163
6	0.016	0.028	0.064	0.078	0.010	0.095	0.135	0.129
8	0.016	0.024	0.059	0.088	0.036	0.041	0.053	0.050
10	0.004	0.009	0.053	0.108	0.375	0.113	0.072	0.009

prediction error $\varepsilon(R) \in (0.000, 0.716)$ with the average error measured across all the configurations $\overline{\varepsilon}(R) = 0.134$. The numeric results for the response time across all the configurations are presented in the table 6.21.

The model predicts the response time of Tomcat in the servlet container setup with significantly higher accuracy than in the web server configuration. We believe this is explained by the lower variability in a demand for CPU time compared to the higher variability in the demand for the hard drive in a web server setup.

The prediction results for response time T are shown at the Figure 6.36. The relative prediction error $\varepsilon(T) \in (0.000, 0.236)$, while the mean error for all configurations $\overline{\varepsilon}(T) = 0.053$. For saturated configurations, $\overline{\varepsilon}(T_{sat}) = 0.099$. The numeric results for the throughput across all the configurations are presented in the table 6.22.

Surprisingly, the prediction accuracy for throughput in a servlet container configuration is lower than in the web server configuration. Although the model correctly predicts the point of saturation for the server, it predicts the absolute value of T less accurately in some configurations. This occurs

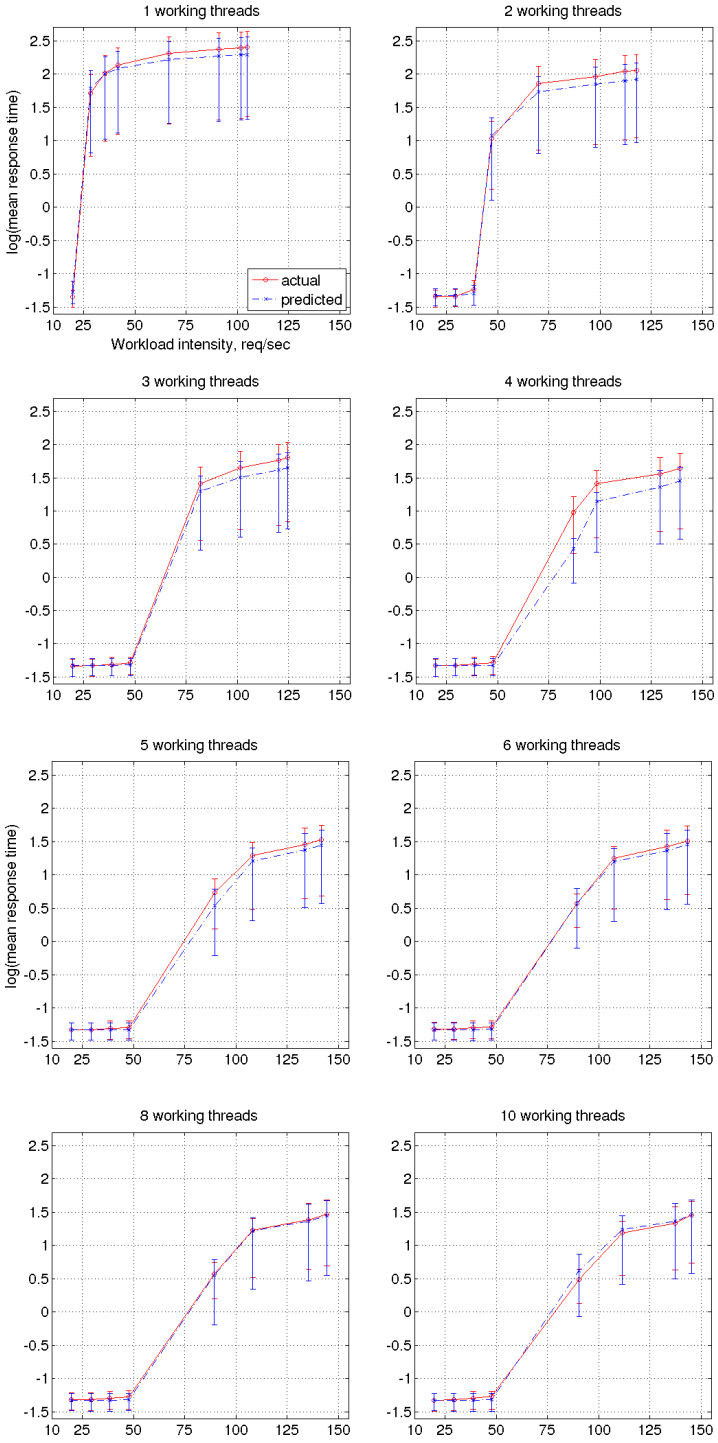

Figure 6.35: Predicted and measured response time for the Tomcat in a servlet container configuration.

because the amount of CPU time for the $\langle 2687, 1460, 2243 \rangle$ CF (represented by the `blk2687_1460_2243` block) somewhat varies across the configuration space. We believe this variation may be caused by the specifics of usage of the CPU cache by the application.

The model correctly predicts a strong dependency between the number of working threads and the performance of the Tomcat. In particular, the saturation point for a server depends on the number of threads: in a configuration with one working thread server becomes saturated at the IRR=21 req/sec, and in configurations with 4 or more working threads saturation occurs at the IRR=75–85 req/sec.

Again, the reason for this lies in the nature of interaction between the hardware and the workload. PDF conversion is a CPU-heavy application, so performance of the server in this setup is bounded by the number of the active CPU cores. In particular, performance characteristics of the Tomcat (response time, throughput) improve significantly when the number of working threads is increased from 1 to 4. But as the number of working threads further increases, all the four CPU cores become utilized (the predicted CPU utilization reaches 400%, see Figure 6.37 and Table 6.23 for predicted CPU utilization). As a result, further increase in the number of working threads does not result in the proportional performance increase.

Our experiments with models of Tomcat demonstrate the ability of our framework to predict performance of large and complex applications, expressing complex workload characteristics. We believe these results clearly demonstrate feasibility of our approach for performance modeling and prove that accurate performance models of large industrial applications can be built automatically.

Table 6.22: Predicted and measured throughput for the Tomcat in a servlet container configuration.

Measured troughput, req/sec

Num. threads	Average workload intensity, req/sec							
	19.67	29.10	38.16	46.87	83.09	102.99	125.44	132.68
1	19.656	22.036	20.792	20.153	19.029	18.267	18.426	18.164
2	19.641	29.184	38.450	43.098	37.479	36.799	34.908	35.360
3	19.640	29.141	38.483	47.843	59.223	55.678	53.901	52.624
4	19.631	29.211	38.485	47.509	76.368	70.534	69.776	68.488
5	19.616	29.141	38.474	47.718	83.314	81.141	78.957	78.901
6	19.617	29.228	38.532	47.561	85.688	83.533	81.168	80.533
8	19.635	29.164	38.481	47.559	85.797	84.760	84.632	84.404
10	19.625	29.144	38.577	47.732	87.852	88.762	89.073	86.371

Predicted troughput, req/sec

Num. threads	Average workload intensity, req/sec							
	19.67	29.10	38.16	46.87	83.09	102.99	125.44	132.68
1	19.665	21.241	21.225	21.228	21.245	21.254	21.257	21.267
2	19.676	29.213	38.459	42.491	42.516	42.523	42.535	42.427
3	19.678	29.164	38.579	47.917	63.590	63.638	63.726	63.815
4	19.684	29.235	38.540	47.609	84.498	84.388	84.508	84.642
5	19.655	29.158	38.547	47.783	84.874	84.707	84.462	84.605
6	19.671	29.248	38.584	47.612	84.477	84.810	84.773	84.642
8	19.671	29.196	38.511	47.667	84.660	84.531	84.822	84.802
10	19.659	29.160	38.649	47.801	84.604	84.562	84.765	84.469

relative error

Num. threads	Average workload intensity, req/sec							
	19.67	29.10	38.16	46.87	83.09	102.99	125.44	132.68
1	0.000	0.036	0.021	0.053	0.116	0.164	0.154	0.171
2	0.002	0.001	0.000	0.014	0.134	0.156	0.218	0.200
3	0.002	0.001	0.002	0.002	0.074	0.143	0.182	0.213
4	0.003	0.001	0.001	0.002	0.106	0.196	0.211	0.236
5	0.002	0.001	0.002	0.001	0.019	0.044	0.070	0.072
6	0.003	0.001	0.001	0.001	0.014	0.015	0.044	0.051
8	0.002	0.001	0.001	0.002	0.013	0.003	0.002	0.005
10	0.002	0.001	0.002	0.001	0.037	0.047	0.048	0.022

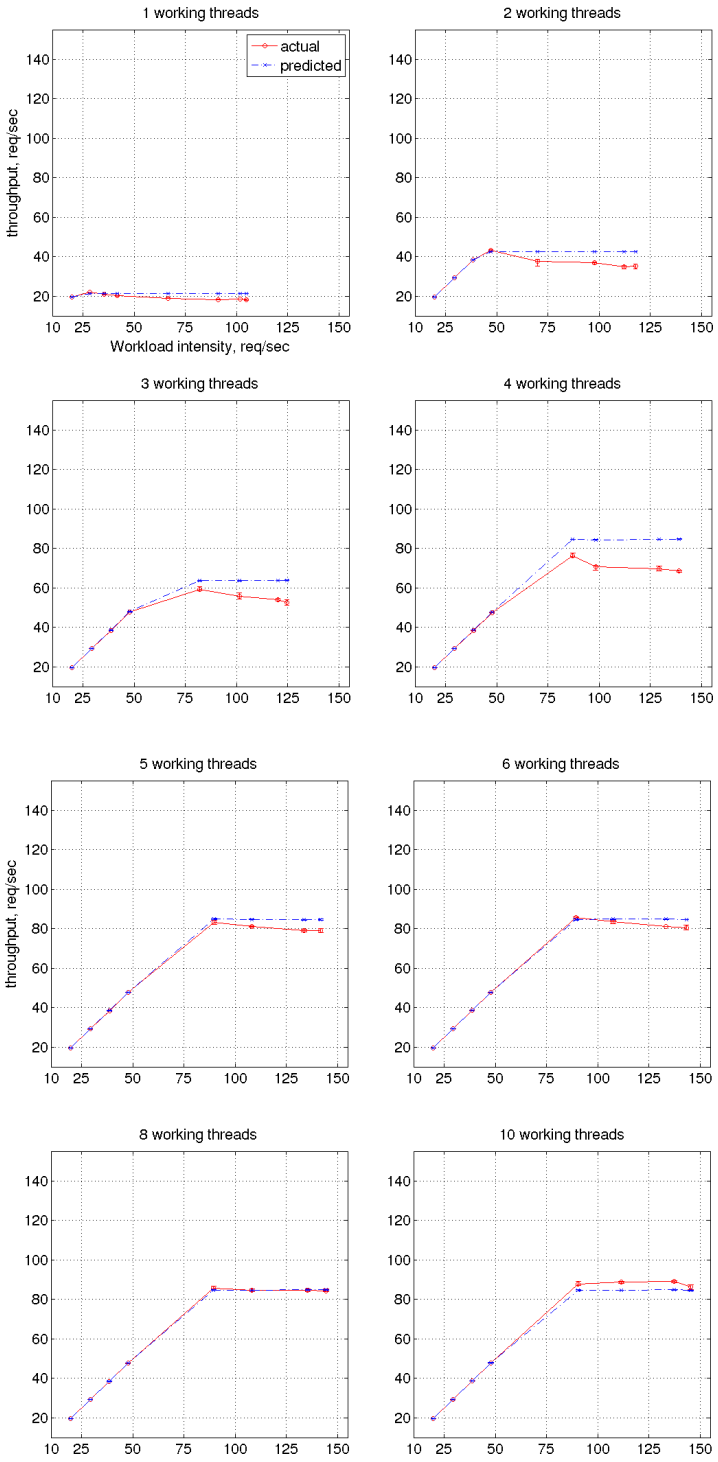

Figure 6.36: Predicted and measured throughput for the Tomcat in a servlet container configuration.

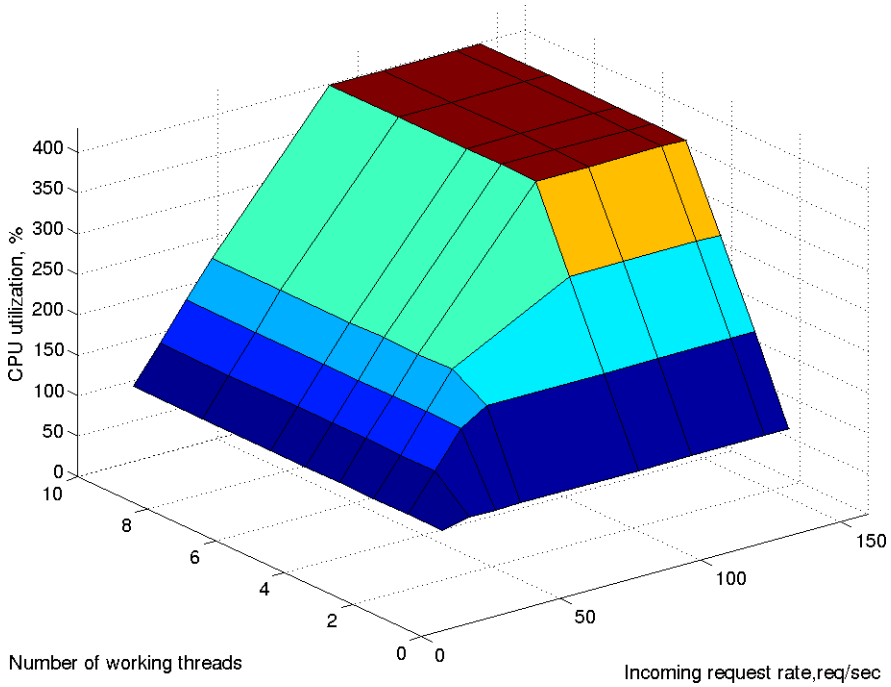

Figure 6.37: Predicted utilization of the CPU for the Tomcat in a servlet container configuration.

Table 6.23: Predicted utilization of the CPU for the Tomcat in a servlet container configuration.

Predicted CPU utilization for Tomcat in a servlet container configuration, req/sec

Num. threads	Average workload intensity, req/sec							
	48.29	70.74	92.10	111.8	127.5	140.4	151.8	
1	92.86	100.27	100.34	100.42	100.65	100.90	100.91	100.90
2	92.88	137.72	181.36	200.38	200.68	200.89	200.89	200.92
3	92.86	137.20	182.98	226.17	300.30	300.84	300.92	300.89
4	92.65	138.06	182.16	224.25	398.20	398.32	398.70	398.61
5	92.76	137.35	181.78	225.97	399.40	399.96	398.63	398.74
6	92.77	137.92	182.13	224.56	398.68	399.69	399.65	399.31
8	93.10	137.46	180.96	224.84	399.14	399.05	399.67	399.79
10	93.07	137.70	182.10	224.51	398.09	397.88	398.65	398.69

Chapter 7

Conclusion

In this section we present the summary of our work, describe lessons we have learned from it, and discuss limitations of our approach. Finally, we identify directions for the future research in the area of performance prediction and analysis of multithreaded programs.

7.1 Summary

This thesis presents our methodology for automatic performance modeling of multithreaded programs. Performance models have many important applications, including building autonomous data centers, provisioning computational resources on a cluster, and detecting performance anomalies in programs.

However, applying existing approaches for building models of multithreaded programs may not produce the desired results. Constructing some performance models require running the program in many different configurations, which may not be possible in a production environment. Other models do not simulate complex synchronization operations, and using these models for predicting performance of multithreaded programs result in inferior prediction accuracy.

This thesis makes two main contributions to the areas of performance modeling and program analysis. First, we propose a discrete-event model specially designed for predicting performance of multithreaded programs. Second, we develop an approach for building performance models automatically by running the program in a single representative configuration.

Performance modeling of multithreaded programs. We developed a hierarchical model to predict the performance of a multithreaded system. Different model tiers simulate different factors that influence performance of the multithreaded program. Interaction between the model tiers simulates joint influence of these different factors on the program's performance. This unique architecture allows our models to accurately predict performance of a wide range of multithreaded programs.

The upper tier of the model is a queuing network, whose queues represent buffers and queues in the program, and service nodes represent program's threads. The upper-tier model simulates delays

that occur because of queuing, and delays associated with processing individual tasks by threads. The exact amount of delay introduced by each thread is simulated by the mid-tier model.

The mid-tier model represents program's threads as probabilistic call graphs. Nodes of the graph correspond to the fragments of the program's code. These code fragments represent CPU-bound computations, disk I/O, and synchronization operations performed by the thread. Edges represent sequence in which these operations may be executed. The amount of time required for each operation to complete is simulated by the low-tier model.

The low-tier model simulates shared resources of the program. This includes models of shared synchronization constructs (locks), hardware resources, and corresponding components of the operating system. Low-tier models simulate contention of computation resources by multiple threads, which is essential for an accurate performance prediction in a multithreaded program.

To implement our models we have developed a PERSIK framework – a discrete-event simulator written using a C++ language. PERSIK can be used for building performance models of programs written using a general-purpose framework for multithreaded programming such as `java.util.concurrent`, `System.Threading`, or `pthread`.

Our experiments showed that PERSIK models can accurately predict performance of multithreaded applications. However, manually building the model of even a small-size program proved to be time-consuming and error-prone activity. To make our models practical we developed a framework for building PERSIK models automatically.

Building performance models automatically. Building simulation models of multithreaded programs is a hard problem. It requires exhaustive information about the program, including semantics of thread interactions. However, there are numerous ways to implement locks and queues, and to expose their functionality to threads. In general, retrieving this information automatically requires very complex and potentially inaccurate program analysis.

We discovered that analysis of a program could be greatly simplified if that program relies on well-defined implementation of high-level locks and queues. By tracking calls to functions and methods that implements these locks we uncover semantics of thread interaction in the program.

Based on this general idea we developed a four-stage methodology to generate performance models automatically. First, our framework runs the program in a simple representative configuration and samples its stack. Results of this run are used to detect thread pools in the program and to discover frequently called library functions.

Second, the static analyzer scans the code of the program searching for specific constructs that perform synchronization and I/O operations. These constructs are represented as code fragments in the mid-tier models of the program's threads.

Third, the dynamic analysis of the program is performed. Code fragments (CFs) detected during the static analysis are instrumented, and the program is executed in the same configuration as during the initial stack sampling run. By analyzing the resulting trace the framework discovers structure of probabilistic call graphs that represent program's threads, locks and queues and their parameters, and parameters of CFs. The probabilistic call graphs are further trimmed to ensure the compactness

and high performance of the thread models.

Finally, the discovered information is translated into the PERSIK model of the multithreaded program.

Results. We implemented our methodology as a tool for building performance models of Java programs. We used this tool to automatically build performance models of large industrial programs, which demonstrates the practicality of our approach. We also built models of various small- to medium-size programs, demonstrating our ability to model different types of multithreaded applications.

We verified our approach by building models of different Java programs. These include web servers, scientific computing applications, multimedia programs, and financial applications. The average prediction error of our models ranges in $(0.062 \ldots 0.117)$ for computation-intense and $(0.249 \ldots 0.269)$ for I/O-intense workloads, which is comparable to the results reported by other studies and other approaches [44],[64],[103],[73]. Furthermore, our framework can predict usage of computational resources such as the hard drive and the CPU.

7.2 Lessons From Building Performance Models

While working on our approach for automated building we made a few important findings:

- 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. For example, barriers enforce order of computations in the Galaxy and Moldyn scientific computing applications. Failure to simulate these locks would result in a non-functional model of the program.

- It is difficult to build simulation models that can handle a broad range of multithreaded programs, even if these programs are written using the same programming language. 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.

- 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.

- Debugging performance models is a difficult task. Often the only manifestation of the bug is the deviation between the predicted and actual performance. Such deviation may be caused by various reasons. The most common causes of bugs in the model are the lack of care in experiment design, incorrect structure of the probabilistic call graph, inaccurate measurements

of code fragments parameters, and spurious computations that occur during the experiments. 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.

- Accurate prediction of performance requires precise measures of resource demands for the elements of the program. In certain cases 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 a difficult task. Measuring the resource demand for the hard drive is especially problematic, as it requires collecting data both in the user and kernel mode.

7.3 Assumptions and Limitations

Our framework is capable of building performance models automatically, but it imposes certain assumptions on the programs it can handle. These assumptions can limit the range of programs and workloads that can be simulated. Below we discuss limitations of our framework. Our plans for addressing these limitations are discussed in the Section 7.4.

The following limitations are inherent to the architecture of our models (see Section 3):

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

- Our models assume that the structure of the probabilistic call graphs δ and parameters Π of code fragments remain same across the configuration space of the program. Changes in workload characteristics, such as the resolution of an image in the Sunflow 3D renderer or probabilities of accessing individual web pages by the Tomcat web server, would require recollecting these parameters of the model. However, in prospective such parameter recollection can be done on-line.

- Our models simulate performance of multithreaded, but not distributed systems. In particular, they do not explicitly simulate calls made by the program to applications located at different hosts, such as Web services or SQL databases.

- Our framework in its present state can build models of only those programs that implement multithreading using the well-defined synchronization operations. Dynamic analysis described in [76] can be used to discover semantics of locks implemented using low-level constructs, such as monitors. However, programs that implement "custom" locks that cannot be assigned to

one of existing lock types (semaphore, barrier, mutex, read-write lock 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.

Other limitations more reflect the current state of our modeling framework and can be addressed by refining its implementation:

- PERSIK 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 performance artifacts related to maintaining cache coherence, such as false sharing. As a result, the modeling accuracy can decrease for certain workloads and hardware platforms.

- Automatic program analysis is currently implemented for Java applications that use locks and queues from the `java.util.concurrent` library. Because this package provides a rich set of primitives and is the recommended approach to synchronization [9], we don't consider the latter the major restriction.

- Currently Java does not provide means for accurate, precise, and low-overhead measurements of CPU times for a thread. Thus during the analysis of Java programs wallclock time is used as a substitute for the actual CPU time. This limits the set of configurations that can be used for building models;

- PERSIK can not model multiple programs that are deployed on the same hardware. Our program analysis can reliably measure resource demands, such as CPU time and I/O operations, only if there is no other process that extensively uses hardware resources of the system. Similarly, PERSIK model can predict performance of the program accurately only if no other program is co-located on the same physical hardware as the given one.

7.4 Future Work

We see two main directions for extending our research in the areas of performance modeling and analysis of multithreaded programs. The first direction is improving the flexibility of performance models, so they could predict performance for a wider range of workloads and hardware configurations. This may require both refinement of our existing performance models, as well as search for new, more agile approaches to performance modeling and program analysis. The second direction is applying our performance modeling and program analysis techniques to a wide range of problems.

7.4.1 Improving Flexibility of the Models

Our models can predict performance for different configuration options of the program (e.g the number of threads), workload intensities (the number of requests received per second), and the number of CPU cores. However, programs also face changes in the characteristics of workload, in the hardware used to deploy the program, and in the code of the program itself. These changes can significantly alter performance of the program. Below we discuss approaches towards developing more flexible performance models, which can predict performance for different workloads and hardware configurations.

Changes in the workload characteristics, such as the probabilities of accessing individual web pages by the web server or the image resolution in the 3D rendering programs, may cause changes in the behavior and resource demands of the program. This, in turn, will require updating the transition probabilities δ and resource demands of individual code fragments Π in the mid-tier model. We envision two possible solutions to this problem.

The first solution is to update the model on-line without stopping the program. This approach would require developing techniques for low-overhead program analysis that can be enabled and disabled dynamically during the progam's execution. Such analysis can be used to monitor the behavior and resource demands of the program and update resource demands Π and to probabilities δ if necessary.

The second solution is to use the hybrid of simulation and statistical modeling. The program's workload will be described using a subset of metrics X'. Then the dependency $(\delta, \Pi) = f(X')$ will be approximated using a statistical model. This approach may require running the model in few configurations in order to collect data required to approximate dependency $(\delta, \Pi) = f(X')$. However, the size of the training set is expected to be significantly smaller than if the pure statistical model was used [37].

We will predict performance of the program on different hardware by developing a set of microbenchmarks. Microbenchmarks will be quickly executed on a target platform, and their performance will be measured. Results will be used to estimate the running time of the program components on the target platform. The challenge here is to design a library of microbenchmarks representative of a range of different workloads and program's behaviors.

Changes in the program's code may impact the performance of the program by the different degree. Some changes may have no effect on performance at all, while other changes may cause a significant performance impact and thus require rebuilding the model. Rebuilding the model after every change to the program may be burdensome, thus we propose developing a criteria for rebuilding performance models. We will rely on techniques from the area of mining software repositories. We will mine the history of changes in the program, and evaluate the impact of these changes on the performance. Based on the properties of the change, such as the amount of changed code or the types of the changed program constructs, we will establish criteria for reconstructing the model.

A valuable extension to the PERSIK framework would be the models of network I/O, memory, and cache operations. Developing such models would enhance the prediction accuracy of PERSIK

models and allow simulating a wider range of workloads. Although models for predicting memory and cache performance are known [71] [48], these models either require data specific to a particular execution of the program or work significantly slower than the program itself. Thus we plan developing accurate and robust models for predicting performance of memory and cache.

7.4.2 Extending the Scope

In this thesis we presented an approach for building performance of multithreaded programs running on a single computer having a symmetric multiprocessor architecture. However, modern computer systems are becoming increasingly heterogeneous. This includes distributed systems, whose components are deployed at different computers. This also include systems that use heterogeneous hardware, such as general-purpose GPUs (GPGPU), to speed up computations. We plan to extend our models so they could simulate distributed and heterogeneous systems, as well as cloud-based applications.

An important practical application of our models is predicting performance of server-side applications, such as web servers or application servers. However, these programs often issue calls to remote applications running on different machines. As a result, the performance of the server is often determined by the timing of these calls. PERSIK models in their current form cannot simulate such distributed systems. We would like to extend our models so they could simulate distributed systems, whose hosts run multithreaded applications.

We can build models of distributed systems by introducing another layer in the hierarchy of models. This layer represents 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 [4] or NS3 [17] simulators. This layer will predict the performance of the distributed system at the global scale and also delays caused by network communication between individual hosts. Performance of each host will be simulated using a corresponding PERSIK model.

Another direction for improving PERSIK would be developing techniques for predicting performance of co-located programs. Co-located programs are either directly deployed on a same computer or are executing within different virtual machines that share the same physical hardware. Co-locating programs has become a common practice in the cloud environment, as it improves maintainability and utilization of the hardware resources.

Modeling co-located programs would require minor modifications to PERSIK framework. However analyzing such programs and measuring their resource demands can be challenging, especially in a virtualized environment. We plan to develop techniques for measuring resource demands of co-located applications directly or for inferring their resource demands using methods similar to described in [83][29].

Building models of distributed systems and co-located programs would enable predicting performance of applications in the cloud environment. Cloud-based systems becoming increasingly prevalent, and developing practical methodologies for predicting performance of these systems would be an important practical application of our framework.

Building models of applications that use GPGPUs is another area for the future research. Modeling of such programs would require extending our framework to simulate the GPU computations itself and delays caused by transferring data to the GPU. Automatic building of these models will require detecting corresponding constructs in the code of the program, such as OpenCL or AMP statements, and discovering their performance parameters.

One more application of performance models is simulating the mobile offloading systems. In these systems the smartphone offloads some computations to the cloud-based server and waits for results. Mobile offloading may decrease the power consumption of the device, but it may incur performance overhead. Building a performance model of such system will help finding an efficient way to implement offloading in a given mobile program. In prospective, developing such models would allow automatic detection of the components of the mobile program that can be offloaded.

Just as any successful research project, this thesis opens a range of opportunities for the future work in the area of performance modeling. Predicting performance for a variety of programs, workloads, and hardware setups; building models of cloud applications and heterogeneous systems are the main directions for my future research. Developing practical solutions for these problems would not only result in significant academic results, but will be beneficial for the computer industry in general.

Building performance models is a challenging research area. Developing scientifically sound and practical solutions for performance prediction require not only a thorough research work, but also significant implementation effort. Nevertheless, the knowledge and skills I gained while working on this thesis make me confident in my ability to overcome future challenges and to contribute to the exciting area of performance engineering.

Bibliography

[1] http://en.wikipedia.org/wiki/Resource_contention.

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

[3] http://www.mathworks.com/products/simevents/.

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

[5] http://sourceforge.net/apps/trac/tornado/.

[6] http://sourceware.org/systemtap/.

[7] http://sourceforge.net/projects/tinyhttpd/.

[8] http://acme.com/software/http_load/.

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

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

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

[12] http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/s3contents.html.

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

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

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

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

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

[18] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89, New York, NY, USA, 2003. ACM.

[19] Mert Akdere, Ugur etintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. Learning-based query performance modeling and prediction. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *ICDE*, pages 390–401. IEEE, 2012.

[20] Gene 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.

[21] Eric Anderson. Hpl–ssp–2001–4: Simple table-based modeling of storage devices, 2001.

[22] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard 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.

[23] Josh Barnes and Piet Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324:446–449, 1986.

[24] Falko 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.

[25] Falko Bause, Peter Buchholz, and Peter Kemper. Hierarchically combined queueing petri nets. In *In proc. of the 11th International Conference on Analysis and Optimization of Systems*, pages 176–182, 1994.

[26] Steffen Becker, Heiko Koziolek, and Ralf 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.

[27] Mohamed Bennani and Daniel 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.

[28] Fabian Brosig, Nikolaus Huber, and Samuel 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.

[29] Fabian Brosig, Samuel Kounev, and Klaus 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, October 2009.

[30] Paul D. Bryan, Jason A. Poovey, Jesse G. Beu, and Thomas M. Conte. Accelerating multi-threaded application simulation through barrier-interval time-parallelism. In *MASCOTS*, pages 117–126. IEEE Computer Society, 2012.

[31] 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.

[32] Jacob Burnim and Koushik 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.

[33] Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. Modeling program resource demand using inherent program characteristics. *SIGMETRICS Perform. Eval. Rev.*, 39(1):1–12, June 2011.

[34] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based faliure and evolution management. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 23–23, Berkeley, CA, USA, 2004. USENIX Association.

[35] Damodar Chetty. *Tomcat 6 Developer's Guide*. Packt Publishing, 2009.

[36] Leslie Cheung, Leana Golubchik, and Fei 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.

[37] Byung-Gon Chun, Ling Huang, Sangmin Lee, Petros Maniatis, and Mayur Naik. Mantis: Predicting system performance through program analysis and modeling. *CoRR*, abs/1010.0019, 2010.

[38] Emilio Coppa, Camil Demetrescu, and Irene 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.

[39] Witold Drytkiewicz, Steffen Sroka, Vlado Handziski, Vlado H, Andreas Kpke, Holger Karl, and Technische Universitt Berlin. A mobility framework for omnet++, 2003.

[40] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli 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.

[41] Wenying Feng and Yong 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.

[42] Domenico Ferrari, Giuseppe Serazzi, and Alessandro Zeigner. *Measurement and tuning of computer systems*. Prentice-Hall, 1983.

[43] Greg Franks and Murray 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.

[44] A. Ganapathi, Yanpei 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.

[45] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David 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.

[46] Stephen Gilmore, Jane Hillston, Leïla Kloul, and Marina Ribaudo. Software performance modelling using pepa nets. In *Proc. of of international workshop on Software and performance*, WOSP '04, pages 13–23, New York, NY, USA, 2004. ACM.

[47] Simon F. Goldsmith, Alex S. Aiken, and Daniel 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.

[48] Nagendra Gulur, Mahesh Mehendale, Raman Manikantan, and Ramaswamy 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.

[49] Jens Happe, Dennis Westermann, Kai Sachs, and Lucia Kapov. Statistical inference of software performance models for parametric performance completions. In *QoSA'10*, pages 20–35, 2010.

[50] Michael Hauck, Jens Happe, and Ralf H. Reussner. Automatic derivation of performance prediction models for load-balancing properties based on goal-oriented measurements. In *Proc. of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '10, pages 361–369, Washington, DC, USA, 2010. IEEE.

[51] Joseph L. Hellerstein, Mark M. Maccabee, W. Nathaniel Mills III, and John Turek. Ete: A customizable approach to measuring end-to-end response times and their components in distributed systems. In *ICDCS*, pages 152–162. IEEE, 1999.

[52] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[53] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, October 1974.

[54] Sofie Van Hoecke, Tom Verdickt, Bart Dhoedt, Frank Gielen, and Piet 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.

[55] Curtis E. Hrischuk, Jerome A. Rolia, and C. Murray 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.

[56] H.H. Huang, Shan 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.

[57] Tauseef A. Israr, Danny H. Lau, Greg Franks, and Murray 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.

[58] 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.

[59] Stephen A. Jarvis, Daniel P. Spooner, Helene N. Lim Choi Keung, Junwei Cao, Subhash Saini, and Graham R. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Gener. Comput. Syst.*, 22(7):745–754, August 2006.

[60] Kristján Valur Jónsson. Httptools: A toolkit for simulation of web hosts in omnet++. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 70:1–70:8, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[61] Samuel Kounev, Simon Spinner, and Philipp 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.

[62] Lars M. Kristensen, Sren Christensen, and Kurt Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132, 1998.

[63] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance, Computer System Analysis Using Queuing Network Models*. Prentice Hall, 1984.

[64] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally 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.

[65] Philipp Meier, Samuel Kounev, and Heiko Koziolek. Automated transformation of component-based software architecture models to queueing petri nets. pages 339–348, 2011.

[66] D. A. Menascé. Two-level iterative queuing modeling of software contention. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '02, pages 267–, Washington, DC, USA, 2002. IEEE Computer Society.

[67] Daniel A. Menasce and Mohamed N. Bennani. Analytic performance models for single class and multiple class multithreaded software servers. In *Int. CMG Conference*, 2006.

[68] Nick Mitchell and Peter 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.

[69] Tipp Moseley, Daniel A. Connors, Dirk Grunwald, and Ramesh 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.

[70] Dushyanth Narayanan, Eno Thereska, and Anastassia 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.

[71] Nicholas Nethercote, Robert Walsh, and Jeremy Fitzhardinge. Building workload characterization tools with valgrind. Invited tutorial, October 2006.

[72] Hai Q. Nguyen and Amy 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.

[73] Hai Quang Nguyen and Amy 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.

[74] 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.

[75] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[76] Steven Reiss and Alexander Tarvo. Automatic categorization and visualization of lock behavior. In *Proc. of the first IEEE Working Conference on Software Visualization*, VISSOFT '13. IEEE, 2013.

[77] Steven 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.

[78] Steven P. Reiss and Suman Karumuri. Visualizing threads, transactions and tasks. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 9–16, New York, NY, USA, 2010. ACM.

[79] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proc. of the 23rd International Conference on Software Engineering*, ICSE '01, pages 221–230, Washington, DC, USA, 2001. IEEE.

[80] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Softw. Eng.*, 21(8):689–700, August 1995.

[81] Jerome Rolia and Vidar Vetland. Parameter estimation for performance models of distributed application systems. In *Proc. of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '95, pages 54–. IBM Press, 1995.

[82] Jerry Rolia, Giuliano Casale, Diwakar Krishnamurthy, Stephen Dawson, and Stephan 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.

[83] Jerry Rolia, Amir Kalbasi, Diwakar Krishnamurthy, and Stephen 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.

[84] Luiz De Rose and Daniel A. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In *ICPP*, pages 311–318, 1999.

[85] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The simos approach. *IEEE Parallel Distrib. Technol.*, 3(4):34–43, December 1995.

[86] Nilabja Roy, Akshay Dabholkar, Nathan Hamm, Lawrence W. Dowdy, and Douglas C. Schmidt. Modeling software contention using colored petri nets. In Ethan L. Miller and Carey L. Williamson, editors, *MASCOTS*, pages 317–324. IEEE, 2008.

[87] Robert Sedgewick. *Algorithms in C.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[88] Jonathan Y. Stein. *Digital Signal Processing: A Computer Science Perspective.* John Wiley & Sons, Inc., New York, NY, USA, 2000.

[89] Marille Stoelinga. An introduction to probabilistic automata. *Bulletin of the European Association for Theoretical Computer Science*, 78:176–198, 2002.

[90] Nathan R. Tallent and John 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.

[91] 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, September 2010.

[92] Dharmesh Thakkar, Ahmed E. Hassan, Gilbert Hamann, and Parminder 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.

[93] Eno Thereska, Bjoern Doebel, Alice X. Zheng, and Peter 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.

[94] Eno Thereska and Gregory 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.

[95] R.D. van der Mei, R. Hariharan, and P.K. Reeser. Web server performance modeling. *Telecommunication Systems*, 16:361–378, 2001. 10.1023/A:1016667027983.

[96] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[97] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory 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.

[98] Elias Weingartner, Hendrik vom Lehn, and Klaus Wehrle. A performance comparison of recent network simulators. In *Proc. of the IEEE International Conference on Communications*, Dresden, Germany, 2009. IEEE.

[99] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh 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.

[100] C. Murray Woodside, Curtis E. Hrischuk, Bran Selic, and Stefan Bayarov. Automated performance modeling of software generated by a design environment. *Perform. Eval.*, 45(2-3):107–123, 2001.

[101] C. Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Trans. Comput.*, 44(1):20–34, January 1995.

[102] Xiuping Wu and Murray Woodside. Performance modeling from software components. In *Proc. of International Workshop on Software and Performance*, WOSP '04, pages 290–301, New York, NY, USA, 2004. ACM.

[103] Jing Xu, Alexandre Oufimtsev, Murray Woodside, and Liam 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.

[104] Qiang Xu and Jaspal 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.

[105] Dmitrijs Zaparanuks and Matthias Hauswirth. Algorithmic profiling. *SIGPLAN Not.*, 47(6):67–76, June 2012.

[106] Tao Zheng, Marin Litoiu, and Murray Woodside. Integrated estimation and tracking of performance model parameters with autoregressive trends. In *Proceedings of the 2Nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, pages 157–166, New York, NY, USA, 2011. ACM.