Automated Analysis of Multithreaded Programs for Performance Modeling

Alexander Tarvo
Brown University
Providence, RI, USA
alexta@cs.brown.edu

Steven P. Reiss
Brown University
Providence, RI, USA
spr@cs.brown.edu

ABSTRACT

The behavior of multithreaded programs is often difficult to understand and predict. Synchronization operations and limited computational resources combine to produce complex non-linear dependencies between a program's configuration parameters and its performance. Performance models are used to understand these dependencies. Such models are complex, and constructing them requires a solid understanding of the program's behavior. As a result, building models of complex applications manually is extremely time-consuming and error-prone. In this paper we demonstrate that such models can be built automatically.

This paper presents our approach for automatically modeling multithreaded programs. Our framework uses a combination of static and dynamic analyses of a single representative run of a system to build a model that can then be explored under a variety of configurations. We show how the models are constructed and show they accurately predict the performance of various multithreaded programs, including complex industrial applications.

Categories and Subject Descriptors

F.3.2 [**General**]: Logics and Meaning of Programs—*Program analysis*; C.4 [**Computer Systems Organization**]: Performance of systems—*Modeling techniques*

Keywords

Program analysis; performance; modeling

1. INTRODUCTION

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

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642979 .

Performance models are essential for various applications [20],[10],[28]. For example, a model may help finding a good configuration for deploying the Tomcat web server. For each combination of configuration parameters, including the number of CPU cores, the number of Tomcat working threads, or the rate of incoming connections, the model will predict response time, throughput, and resource utilization for Tomcat. A configuration that utilizes resources efficiently and satisfies the service agreement can be used for deployment. Performance models can be also used to detect performance anomalies and discover bottlenecks in the program.

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

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

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

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

While working on the automatic model building we made important findings. First, the analysis of a program is greatly simplified if that program relies on well-defined implementation of high-level locks (semaphores, barriers, blocking queues etc.). Second, in order to be fast and easy to understand the resulting model must be simple and compact. Building compact models requires identifying program constructs that do not have significant impact on performance,

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

2. SCOPE AND CHALLENGES

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

Processing tasks is parallelized across thread pools. A *thread pool* is a set of threads that have same functionality and can process tasks in parallel. Multiple threads rely on synchronization to ensure semantic correctness (e.g. the thread may start executing only after a barrier is lifted) and to protect shared data. This results in the parallel execution of some computations and the sequential execution of others. Threads also use shared hardware resources, such as the CPU, disks, and the network simultaneously, which may lead to their saturation. This combination of locking and simultaneous resource usage leads to complex non-linear dependencies between configuration parameters of the program and its performance. As a result, even an expert may not understand such dependencies on a quantitative level. The best approach is to build a performance prediction model.

We focus on following aspects of performance modeling:

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

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

Accurate performance prediction for a range of configurations. This lets our model to answer "what-if" questions about the program's performance, detect performance anomalies in the running program, and be used as a decision-making element of a self-configuring data center.

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

Building performance models of complex, multithreaded systems is challenging. The primary challenges are:

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

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

3. MODEL DEFINITION

Below we briefly describe the model we build automatically. We use discrete-event simulation models [23] that consist of three tiers.

The high-level tier simulates the flow of tasks processed by the program. It is a queuing network model whose queues correspond to program's queues and buffers as well as to some OS queues. The service nodes correspond to the program's threads and thread pools.

The mid-level tier simulates the *delays* that occur in the program's threads as they process tasks. Thread models are probabilistic call graphs (PCGs), where each vertex $s_i \in S$ corresponds to a piece of the thread's code – *a code fragment (CF)*. Edges represent possible transitions of control flow between the CFs and are labeled with their probability. Transition probabilities are defined by the mapping $\delta : S \to P(S)$.

We distinguish three major sources of delays in processing tasks, which correspond to three *classes of code fragments*: I/O code fragments (denoted as c_{io}) represent I/O operations; synchronization (c_{sync}) CFs represent synchronization operations; computation (c_{cpu}) CFs represent computations and memory operations. In addition, c_{in} and c_{out} CFs communicate with the high-level queuing model. c_{in} CFs fetch tasks from the queues of the queuing model, while c_{out} CFs send tasks to the queuing model.

The lower-tier model simulates the system's shared resources: the CPU and the OS thread scheduler, the disk I/O subsystem, and the set $L = \{l_1, ..., l_m\}$ of locks in the program. These models are part of $Q(t)$ – the state of the whole simulation at each moment of time t.

As an example, a model of a simple web server is shown in Figure 1. The accept thread listens for incoming connections (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 sends (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 if from the disk, and sends it to the client. Finally, the thread closes the connection and fetches the next task from the queue.

Execution of each CF results in the delay τ. While the call graph structure $\langle S, \delta \rangle$ does not generally change between different configurations, *execution times for code fragments can be affected by resource contention.* To accurately simulate the time delays τ we rely on the lower-tier models.

For each code fragment we define a set of parameters Π (see Table 1). The parameter of a computation CF $\Pi_{cpu} = \langle \tau_{cpu} \rangle$ is the *CPU time* for that fragment. The parame-

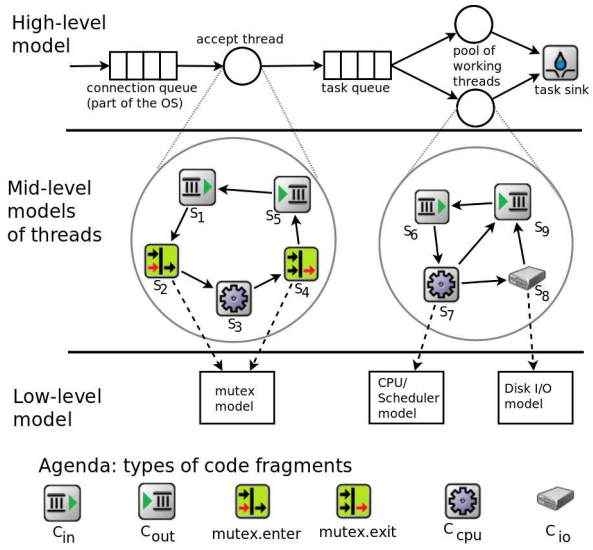

Figure 1: A model for a web server

ters of a disk I/O CF is a sequence $\Pi_{disk} = \langle dio_1, ..., dio_k \rangle$ of low-level disk I/O operations initiated by that CF. The number k of I/O requests allows to implicitly simulate the OS page cache. It was shown [15] that after serving a sufficient number of requests (10^4 to 10^5 in our experiments), the cache enters a steady state, where the probability of cache hit converges to a constant. In terms of our model, k follows a stationary distribution, where $k = 0$ indicates a cache hit.

Values of Π_{cpu} and Π_{disk} vary across different executions of CFs, so we represent them as distributions $\mathbb{P}^{\Pi}_{cpu}$ and $\mathbb{P}^{\Pi}_{disk}$.

The parameters $\Pi_{sync} = \langle lid, optype, \tau_{out} \rangle$ of a synchronization CF are the ID of the lock being called, the type of synchronization operation (e.g. barrier.await, mutex.enter, or mutex.exit), and the timeout.

When the thread model needs to compute the τ_i for the CF s_i, it retrieves the parameters Π_i and calls the corresponding low-tier resource model: c_{cpu} CFs call the model of the CPU and OS scheduler, c_{io} CFs call the model of disk I/O subsystem, and c_{sync} CFs call the model of a corresponding lock $l_j \in L$, which simulates that lock semantically. The resource model computes τ as a function $\tau = f(\Pi, Q(t))$. Once the delay τ is over, the resource model notifies the thread model, which resumes its execution.

Low-level resource models have parameters too. In particular, the parameter of the CPU is the number of cores. The parameters $\langle lid, ltype, lparam \rangle$ of the lock $l_j \in L$ are the lock ID, lock type (e.g. semaphore, barrier, or mutex), and the additional parameters specific to the type of the lock. For example, the parameter of the barrier is the barrier capacity, and the parameter of the semaphore is its count.

Low-level models are implemented as a combination of queuing and statistical models. Their detailed description is beyond the scope of this paper. Information on modeling hardware and locks and can be found in [35],[38],[22].

4. AUTOMATIC MODEL GENERATION

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

Table 1: Model components and their parameters

Entity	Description
$S = \{s_1 ... s_n\}$	The set of all nodes (code fragments) in the PCG
$\delta : S \to P(S)$	Transition probabilities for PCG nodes
τ_i	Delay caused by executing CF $s_i \in S$
$\Pi_{disk} = \langle dio_1, ..., dio_k \rangle$	I/O CF parameters: a sequence of low-level I/O operations
$\Pi_{cpu} = \langle \tau_{cpu} \rangle$	Computation CF parameters: the amount of CPU time
$\Pi_{sync} = \langle lid, optype, \tau_{out} \rangle$	Synchronization CF parameters: an ID of the lock called, operation type, timeout
$L = \{l_1 ... l_m\}$	The set of all locks in a program
$\Pi_{lock} = \langle lid, ltype, lparam \rangle$	Lock parameters: an ID of the lock, lock type, type-specific parameters

- The set of queues, threads (correspond to service nodes in the upper-tier model), and knowledge of their interactions (correspond to c_{in}/c_{out} CFs in the middle-tier model);
- The set of thread pools. The sizes of thread pools are configuration parameters that impact performance;
- The computations, I/O, and locking operations (corresponding to the set S of CFs) and the sequence of their execution (corresponding to transition probabilities δ);
- The parameters of CFs, required to model delays τ;
- The set L of locks, their types, and parameters Π_{lock}.

We collect the required data in four stages (see Figure 2) using a combination of static and dynamic analysis. Each stage saves intermediate results into files that are used as input to subsequent stages.

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

Below we describe these stages in more details.

4.1 Collecting stack samples

During the stack sampling stage our framework finds thread pools, frequently called functions and methods in the program, and frequently called library functions. Identifying libraries is essential for generating correct probabilistic call graphs (see Section 4.3.1).

As the program is being executed, the framework periodically takes "snapshots" of the call stack of the running program, which are merged to build a *call trie* of the program. In a call trie each leaf node contains the code location being executed, which includes the name of a function or a method being executed, and a line number. The non-leaf nodes provide a call stack for that code location. For each leaf the framework maintains the list of pairs $\langle t_1, c_1 \rangle, \ldots \langle t_n, c_n \rangle$, where the c_i is the number of executions of that code location by the thread t_i.

Thread groups are detected in two stages. First a map $\mathbb{T}$ is created. Its keys are thread tuples discovered by sampling,

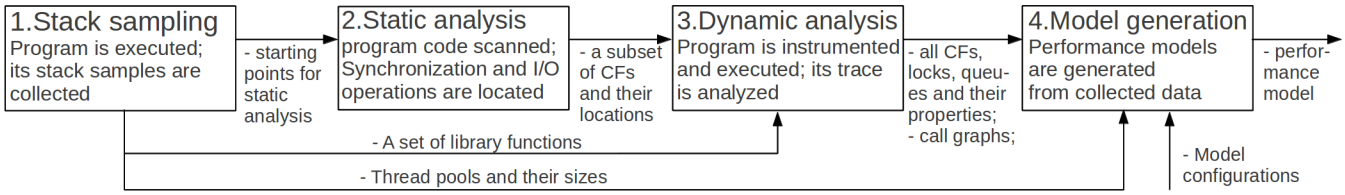

Figure 2: Model creation stages and intermediate results

and its values are execution counts. For each leaf in the trie the framework gets a tuple $Ti = \langle t_1, \dots t_k \rangle$ of threads that executed the node along with the total number of executions $Ci = \sum(c_1, \dots, c_k)$. If $\mathbb{T}$ does not contains the tuple Ti, the pair $\langle Ti, Ci \rangle$ is inserted into $\mathbb{T}$. Otherwise the number of executions for the existing tuple is increased by Ci.

Second, thread tuples in $\mathbb{T}$ are merged. The tuple $\langle T1, C1 \rangle$ is merged with $\langle T2, C2 \rangle$ if and only if all threads in $T2$ also present in $T1$ and $C1 \gg C2$. The resulting tuple is formed as $\langle T1, C1 + C2 \rangle$. After merging, the tuples $T1 \dots Tm \in \mathbb{T}$ represent the thread pools detected in the program.

Stack samples are also used to identify program's libraries. For every function f the framework generates the set of functions $\langle f_1, ..., f_n \rangle$ that called f. If the number of callees $n > 1$, f is added to the set of *library functions*. Although the stack sampling may not detect some rarely executed library functions, this does not affect correctness of our models.

4.2 Static analysis

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

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

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

There are numerous ways to implement synchronization in a program. As a result, detecting c_{in}, c_{out} and synchronization CFs and determining their operation types *optype* may require complex analysis that is very hard to automate. We therefore assume the program has used specific implementations of locks and queues for thread interactions. Examples of such implementations are the `java.util.concurrent` package in Java, the `System.Threading` namespace in C#, and the boost threading library in C/C++.

The analyzer considers calls to specific functions that perform synchronization operations and access program's queues as c_{sync}, c_{in}, and c_{out} CFs appropriately. Typically, these are the functions that constitute the API of the corresponding thread and locking library. The class of the CF and the type of synchronization operation *optype* are inferred from the name and the signature of the called function.

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

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

To discover the c_{io} code fragments, the analyzer tracks API functions that can perform disk I/O. Calls to the functions that may access the file system metadata are considered as I/O CFs as are the bodies of low-level functions that perform file I/O.

4.3 Dynamic analysis

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

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

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

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

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

Figure 3: A fragment of the trace for a thread.

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

The Figure 3 depicts an example of such trace. Here the CF ⟨10, 11⟩ is a c_{in} CF. The object ID=7683745 recorded by the probe 10 identifies the queue, while the argument value 0 correspond to the timeout of 0 milliseconds. The probe 11 reports the return value 4387459, which is an ID of the retrieved object. ⟨27, 28⟩ and ⟨6, 7⟩ are synchronization CFs corresponding to the entry and exit from the synchronized region. The object ID=87235467 identifies the monitor associated with that region. Two instances of ⟨10205, 10206⟩ I/O CF correspond to two (unrelated) file read operations from the disk. Their object IDs identify the instances of the corresponding file objects. Pairs ⟨11, 27⟩, ⟨28, 10205⟩, ⟨10206, 6⟩, and ⟨7, 10205⟩ are the computation CFs.

4.3.1 Construction of probabilistic 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 $(s_i, s_k), \dots, (s_i, s_m)$. The probability of transition from the node s_i to $s_j, j \in (k \dots m)$ is calculated as

$$p(s_i, s_j) = \frac{count(s_i, s_j)}{\sum_{l=k}^{m} count(s_i, s_l)} \quad (1)$$

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

The naïve approach may not represent calls to the program's libraries correctly and generates overly complex PCG. To become practical, this approach must be improved.

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

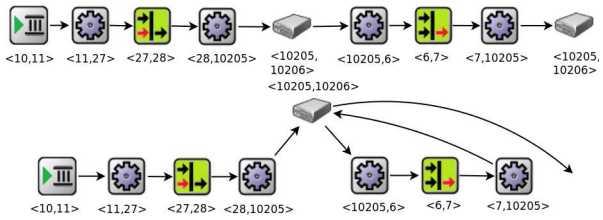

Figure 4: Top: the ground truth PCG from the thread trace. Bottom: the incorrect PCG generated from the trace that contains a library call.

For example, according to the trace shown on the Figure 3 the program enters the synchronized region, reads data from a file, exits the synchronized region, and performs another unrelated file read. The "ground truth" call graph has no loops or branches (see Figure 4, top). However, both I/O operations will eventually call the same read() I/O API that contains an ⟨10205, 10206⟩ I/O CF. As a result, the generated PCG will contain a loop in it (see Figure 4, bottom). While simulating this loop the model may not exit the synchronized region, or may attempt exiting it multiple times. In both cases the behavior of the model will be incorrect.

To address this problem the dynamic analyzer represents separate calls to the library CFs as separate PCG nodes using the node splitting technique described in [33]. For every CF located within one of the program's libraries, the analyzer adds a context information describing the origin of the call to that library.

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

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

Reducing the complexity of the model. According to the naïve approach, all the computations between c_{io}, c_{sync}, c_{in}, and c_{out} CFs are represented as c_{cpu} CFs, even if their impact on performance is negligible. Similarly, every synchronization region is represented as a pair of CFs, even if it is very short and never becomes contended in practice. This leads to an unnecessary complex PCG, consisting of thousands of CFs (see Table 2). Such complex models have low performance and are hard to analyze. To simplify the model we remove all the *insignificant CFs* that have negligible impact on the program's performance.

Model optimization is performed in two steps. First, the whole timeline of the program's execution is split into three phases: the startup phase, when the program doesn't pro-

cess tasks yet; the work phase, when the program processes tasks; and 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, we assume these stages are absent in the trace.

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

Second, the insignificant CFs executed during the work phase are removed from the model. These are c_{cpu} CFs whose summary CPU times amounts to less than t% of the overall CPU time for the thread, and c_{io} CFs whose summary data transfer amounts to less than t% of data transferred by the thread. Setting $t = 3-5$% allows shrinking the PCG by 50-70% without noticeable impact on the accuracy.

Accounting for determinism in the program behavior. Some program behaviors are difficult to represent accurately using a probabilistic model. First, the execution flow may take different paths depending on the availability of the task in the queue. To account for this c_{in}^{fetch} and $c_{in}^{nofetch}$ "virtual" nodes are inserted after each c_{in} node in the PCG. The c_{in}^{fetch} node is executed when the c_{in} CF was able to fetch the task from the queue. $c_{in}^{nofetch}$ is executed if c_{in} did not fetch the task and exited by the timeout.

Second, representing 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 can be shown that X will rather be a random variable that follows a geometric distribution with mean n and a probability mass function $Pr(X = k) = \frac{1}{n} \cdot (1 - \frac{1}{n})^{k-1}$. In most cases this representation has a minor effect on the prediction accuracy. However, if the program's performance y strictly follows the function $y = f(n)$, the predicted performance y' will be a function of a random variable $y' = f(X)$, whose parameters (mean, standard deviation) may differ noticeably from y.

In our experiments such mispredictions occurred if the loop performed an initial population of the program's queues with tasks. To address this issue the dynamic analyzer detects loops in the trace using the algorithm [27]. If the loop contains the c_{out} node, the model explicitly simulates it. Otherwise the loop is represented as a cycle in the PCG.

4.3.2 Retrieving parameters of code fragments

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

Locks and task queues. Parameters of locks and queues are obtained from the arguments passed to constructors and intializers of these locks and queues, and from their return values. The lock type *ltype* is inferred from the signature of the constructor/intializer. The type-specific parameters *lparam* are retrieved from the values of arguments passed to that constructor. The lock ID *lid* is obtained from the reference to the lock returned by the constructor; it uniquely identifies each lock $l_i \in L$. Queues and their parameters are obtained in the same manner.

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

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

c_{cpu} CFs. The parameter of the c_{cpu} CF is the distribution $\mathbb{P}^{\tau}_{cpu}$ of CPU times τ_{cpu}. τ_{cpu} can be accurately measured when the execution time of a thread can be determined. When this is not the case, τ_{cpu} is measured as the difference between the timestamps of start and end probes of the CF, substituting clock time for CPU time. However, in order to use the latter approach we need to avoid configurations where CPU congestion is likely.

c_{io} CFs. The parameters of the c_{io} CF are the number k and properties (the type of I/O operation and the amount of data transferred) of low-level disk I/O requests $\{dio_1, ..., dio_k\}$ initiated by that c_{io} CF. This request-specific data can be retrieved only from the OS kernel. We used the blktrace [1] to retrieve the log of all kernel-mode disk I/O operations initiated by the program.

Generally, the timestamps and thread IDs in the kernel-mode I/O log might not match the timestamps and thread IDs in the instrumentation log. To match blktrace log to the instrumentation log the dynamic analyzer uses cross-correlation – a technique used in signal processing [36]. The cross-correlation $(f \star g)[t]$ is a measure of similarity between signals f and g, where one of the signals is shifted by the time lag Δt. The result of a cross-correlation is also a signal whose maximum value is achieved at the point $t = \Delta t$. The magnitude of that value depends on similarity between f and g. The more similar are those signals, the higher is the magnitude of $(f \star g)[\Delta t]$.

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

Figure 5 depicts the cross-correlation between signals $u(t)$ and $b(t)$. The cross-correlation signal $(u(t) \star b(t))[t]$ reaches its maximum value at the point $\Delta t = 324$, which means that the user signal $u(t)$ is shifted forwards by $\Delta t = 324$ ms with relation to the kernel signal $b(t)$.

The dynamic analyzer matches user to the kernel I/O signals using a greedy iterative procedure. For each pair of signals $\langle u(t)_i \in U,\ b(t)_j \in B \rangle$ the analyzer computes a cross-correlation signal $xcorr_{ij} = b(t)_i \star u(t)_j$ and the value $\Delta t_{ij} = \arg\max_t(xcorr_{ij})$. The user signal $u(t)_i$ matches the kernel signal $b(t)_j$ if the maximum value of the cross-

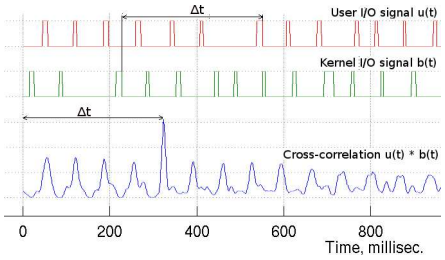

Figure 5: Cross-correlation between I/O signals

correlation signal $xcorr_{ij}[\Delta t_{ij}]$ is the highest across the signal pairs.

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

4.4 Constructing the performance model

The result of the program analysis is a set of text and xml files, which contain all the information required to generate the model: the list of threads, thread pools, and queues in the high-level model; the set S of CFs, their classes and properties Π; transition probabilities δ; the set of locks L and their properties Π_{lock}. This information is used to generate the three-tier performance models described in the Section 3. The models are implemented using the OMNeT simulation toolset [2] and can reviewed in the OMNeT IDE.

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

5. MODEL VERIFICATION

We implemented our approach as a tool for automatically building models of Java programs. The tool uses ASM [3] framework for bytecode analysis and instrumentation.

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

$$\varepsilon = \frac{|\overline{measured} - \overline{predicted}|}{\overline{measured}} \quad (2)$$

We conducted our experiments on a PC equipped with the 2.4 GHz Intel quad-core CPU, 8GB RAM, and 250 Gb HDD running Ubuntu Linux. To uncover potential artifacts in the performance of the test programs our configurations cover a variety of program's behaviors, ranging from under-utilization of resources (e.g. when the number of active threads is less than CPU cores) to their over-utilization. Below we describe our simulations in detail.

5.1 Modeling large industrial applications

We built performance models of two industrial open source Java programs: Sunflow 0.07 3D renderer and Apache Tomcat 7.0 web server. We predicted the performance of Tomcat in two setups: as a standalone web server hosting static web pages and as a servlet container. Considering difference in Tomcat functionality over these setups, corresponding models are significantly different. Table 2 provides information on programs and their models.

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

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

Our models run 8-1000 times faster than the actual program (see Table 2). The actual speedup depends not on the size of a program, but on a ratio between the times required to simulate CFs by the model and times required to execute these CFs by the program. Simulating a CF requires a (roughly) constant amount of computations, regardless of its execution time. Thus models that invoke many CFs with short execution times or simulate intense locking operations tend to run slower. As a result, eliminating insignificant CFs is essential for high performance of the model.

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

Sunflow 3D renderer. Sunflow uses a ray tracing algorithm for image rendering [4]. The main thread splits the frame into multiple tiles and stores them in the queue. The pool of working threads reads tile coordinates from the queue, renders the image tiles, and synthesizes the resulting image. Given the constant size of the image, the number of working threads and the number of CPU cores are two main factors that determine the performance of the Sunflow. The

Table 2: Large programs and their models

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

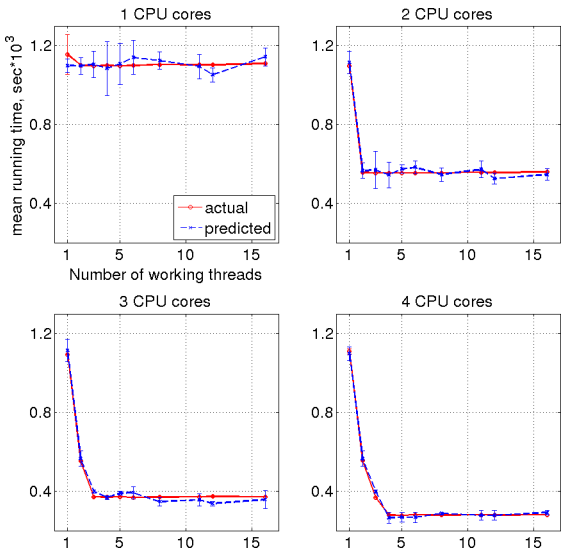

Figure 6: Predicted and measured performance of Sunflow. Good accuracy for configurations involving under- and over-utilization of resources

time required to render the image is the main performance metric.

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

Our experiments demonstrate the ability of our framework to predict performance of a program across different hardware configurations. This does not yet translate into an accurate prediction of the program running on a totally different hardware. Differences in characteristics of CPU, memory, and cache will result in different execution times for individual CFs. Nevertheless, it opens a path for such a prediction because CF timing can be estimated analytically or using microbenchmarks on the target architecture.

Apache Tomcat as a web server. Apache Tomcat is a widely used web server and Java servlet container. In our experiments Tomcat relies on a single blocking queue to pass incoming HTTP requests to a fixed-size thread pool. The performance of the Tomcat was influenced by the size of the thread pool and by the the workload intensity (the number of requests the server receives in a second, req/s). The performance metrics are response time R and throughput T.

We used Tomcat to host about 600000 Wikipedia web pages. We predicted performance of Tomcat with with 1,3,5, and 8 working threads and workload intensity ranging from 48.3 to 156.2 req/s (measured on the server side).

The prediction results for R and T are depicted at the Figure 7. The relative prediction error $\varepsilon(T) \in (0.001, 0.087)$ with average error $\overline{\varepsilon}(T) = 0.0121$. In non-saturated configurations throughput is roughly equal to the incoming request rate, thus the relative error for saturated configurations is a more informative accuracy metric: $\overline{\varepsilon}(T_{sat}) = 0.027$.

The error for R is $\varepsilon(R) \in (0.003, 2.452)$ and $\overline{\varepsilon}(R) = 0.269$. The relatively high error terms are attributed to fluctuations of the page cache hit rate represented by k. According to our measurements, mean $\overline{k} = 0.755$ with standard deviation $\sigma(k) = 0.046$. Overall, precise data collection proved to be essential for accurate performance prediction. Introducing an artificial 15% bias in the value of k resulted in $\varepsilon(R) \in (0.015, 3.109)$ with $\overline{\varepsilon}(R) = 0.882$. This experiment demonstrates the importance of the accurate measurement of CF resource demands.

In a web server setup Tomcat expresses a mixed behavior. 81% of computational resources consumed during processing the HTTP request is the I/O bandwidth, and 19% is CPU time. As a result, the single hard drive becomes the bottleneck that prevents performance from growing significantly as the number of working thread increases. At the same time, remaining CPU computations are parallelized across four CPU cores, resulting in small but noticeable performance improvement.

Apache Tomcat as a servlet container. Tomcat is more frequently used as a servlet container. We used Tomcat to host a web application that reads a random passage from the King James bible, formats it, and converts into the PDF using the iText [5] library.

The prediction results for R is depicted at the Figure 7 (T is not shown due to space limitations). The relative prediction error $\varepsilon(R) \in (0.000, 0.375)$ with the average error $\overline{\varepsilon}(R) = 0.122$. The error for T $\varepsilon(T) \in (0.000, 0.356)$ and $\overline{\varepsilon}(T) = 0.053$, with $\overline{\varepsilon}(T_{sat}) = 0.099$. The CPU time τ_{CPU} fluctuates less than the demand for I/O bandwidth, which leads to the lower prediction error.

The model correctly predicts the workload intensity at which the server saturates. PDF conversion is a CPU-heavy task, thus performance of the server is bounded by the number and performance of CPU cores. Since there are four CPU cores available, the actual saturation point depends on the number of threads. It ranges from 21.4 req/sec for a configuration with 1 thread to 85.5 req/sec for 10 threads.

5.2 Modeling small- to medium-size programs

We built models of the following applications: Montecarlo (a financial application), Moldyn and Galaxy (scientific computing applications), and Tornado (a Web server). Although smaller in size, these programs express functionalities peculiar to a wide range of multithreaded programs. They implement thread interaction in different ways and use a great variety of synchronization mechanisms to enforce a correct order of computations across multiple threads. Table 3 present a summary on these programs and their models.

Montecarlo and Moldyn are parts of the Java Grande benchmark [12] suite. Montecarlo simulates price of market derivatives by generating a number of time series reflecting

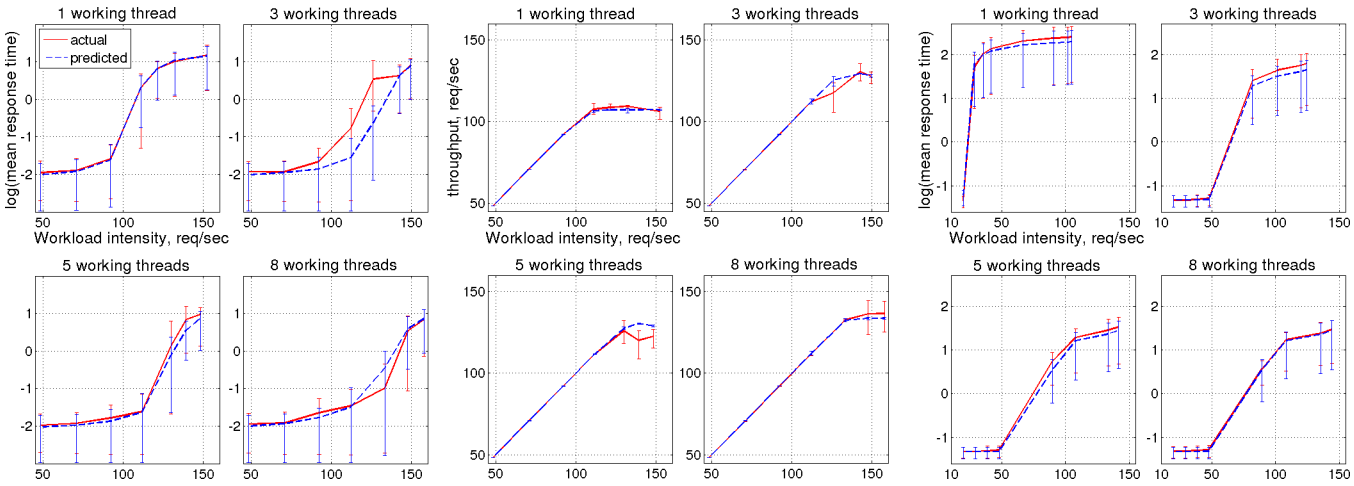

Figure 7: Predicted and measured performance of Tomcat. Left: response time in a web server setup. Small variation in demand for I/O bandwidth lead to large changes in the response time. Center: throughput in a web server setup. Configurations leading to server saturation are detected accurately. Right: response time in a servlet container setup. Consistent demand for the CPU time leads to an accurate prediction.

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

	Montecarlo	Moldyn	Galaxy	Tornado
Size, LOC	3207	1006	2480	1705
Number of probes	18	30	72	40
Number of CFs	17	72	124	88
Number of nodes in the model	24	46	59	36
Mean error $\overline{\varepsilon}$	0.062	0.083	0.075	0.262 (R) 0.010 (T)

prices of the underlying assets. Time series are generated independently using a pool of working threads.

Moldyn simulates motion of argon atoms in a cubic volume. The time is discretized into small steps. During each step (iteration) working threads compute forces acting on atoms, and then the main thread updates positions of atoms.

Galaxy simulates the gravitational interaction of celestial bodies using the Barnes-Hut [8] algorithm. During each iteration the main thread rebuilds the octree, the pool of "force threads" computes forces and updates positions of bodies, and the pool of "collision threads" detects body collisions.

Tornado is a simple web server, whose structure and behavior are described as an example in the Section 3. Unlike Moldyn, Montecarlo, and Galaxy, which engage the CPU-intense computations, Tornado workload is dominated by disk I/O operations.

One configuration parameter common to all these programs was the size of their thread pools. For Montecarlo, Moldyn, and Galaxy we experimented with 1,2,3,4,8,10,12,16 working threads (Galaxy parameters included the number of both force threads and collision threads). Parameters of Tornado were the number of working threads (1,3,5, and 10) and the workload intensity (ranged from 19.8 to 99.6 req/s). The relative prediction error for each program is provided in the Table 3 (detailed results are not shown due to space constraints).

6. DISCUSSION AND LIMITATIONS

Although our framework is capable of building performance models automatically, it imposes certain limitations on the programs we can model. First, our high-level models represent computations as task processing. This approach does not cover all possible programs, but covers most programs of interest for performance purposes.

Second, during data collection we use a single representative configuration, where the transition probabilities δ and CF parameters Π would be similar to δ and Π of a larger set of configurations. This requires the usage patterns for the program, such as the image resolution in Sunflow or probabilities of accessing individual web pages in Tomcat, to remain similar across the configuration space. Changing usage patterns may require reconstructing the model. One solution to this problem would be recollecting δ and Π directly from the running program. Another solution is building a hybrid of statistical and simulation model, where usage patterns are described using metrics X', and the dependency $(\delta, \Pi) = f(X')$ is approximated statistically.

Third, our current framework requires programs to implement multithreading using the well-defined synchronization operations. We do not see it as a major limitation as modern programming frameworks offer rich libraries of locks which programmers are encouraged to use [6]. Furthermore, the semantics of locks implemented using low-level constructs can be discovered using analysis described in [32].

Third, our models do not explicitly simulate calls made by the program to other systems, such as Web services or SQL databases. Timing of these calls can be simulated using statistical models. Alternatively, these systems can be modeled using their own performance models, combined into a model of a distributed system using INET or NS2/3 simulators.

Next, our models simulate memory operations as CPU computations. This didn't affect prediction accuracy in our experiments, but accurate modeling of certain programs or workloads may require explicit simulation of memory operations and corresponding OS and hardware components.

Finally, our framework currently does not include a network model, our static analysis and instrumentation methods are only implemented for Java applications, and we use clock time as a substitute for actual CPU time.

7. RELATED WORK

We divide the related work into two categories: (i) performance modeling and (ii) automated program analysis and model construction.

(i) At the high level the performance of the system can be represented as a function $y = f(\vec{x})$, where $\vec{x}$ is the configuration and y is the performance of the system.

Analytic models explicitly represent this dependency using a set of equations. An analytic model was used to predict the performance of the DBMS buffer pool [28]; the reported relative errors are $\varepsilon(T) \leq 0.1$ and $\varepsilon(R) \in (0.33...0.68)$. Analytic models were employed to study performance of certain multithreaded design patterns [37], and as a central element of the autonomic data center [10].

Building analytic models requires strong mathematical skills and is hard to automate. Moreover, analytically modeling even a simple multithreaded system is challenging [25].

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

Statistical models were used to predict performance of Hadoop tasks [17], SQL queries [14], and scientific applications running on a grid [24] with relative error $\varepsilon \in (0.01,...,0.25)$. CART trees predict performance of the SSD disk with $\varepsilon \in (0.17, ..., 0.25)$ [19] and the traditional hard drive with $\varepsilon \in (0.17, ..., 0.38)$ [40]. However, collecting the training set (X, Y) requires running the system in many different configurations, which is overly time-consuming and costly. The number of executions can be somewhat reduced by optimizing the search through the configuration space [41] or by complex program analysis [13]. Still, such experimentation may not be feasible on a production system.

Queuing networks, Petri nets, and their extensions can model complex behavior, but their construction requires extensive information about the system.

The Layered Queuing Network (LQN) represents the system as a hierarchy of layers, where each layer includes both a queue and a service node. LQNs can be solved analytically and are particularly useful for simulation of distributed systems, where their accuracy reaches $\varepsilon \leq 0.24$ [42],[34]. However, analytic modeling of complex threading behavior with LQN [16] may be challenging. Palladio Component Models (PCM) is another approach to simulation where the system is divided into a number of interconnected components [9].

Colored Petri nets (CPN) extend the traditional Petri nets by allowing multiple types of tokens. In [30] CPN predicted performance of a parallel file system with $\varepsilon \in (0.2...0.4)$, and in [35] CPN was used to simulating the complex locking constructs in a program with $\varepsilon \in (0.0...0.2)$.

Existing models such as CPN can simulate multithreaded systems to some extent. But typically they model a single aspect of the system's behavior, such as locks, an OS component, or a piece of a hardware. This limits applicability of these models to a specific scenario. Instead, our models simulate many factors that influence performance of a system: queuing, synchronization operations, and simultaneous usage of hardware by threads. This allows us to build accurate models of various multithreaded programs and workloads.

(ii) There is a significant amount of work on automated analysis of parallel an distributed programs. The THOR tool combines kernel and user-mode instrumentation to understand and visualize relations between the Java threads and locks [39]. In [32] the dynamic analysis was used to understand and visualize locks in a multithreaded program. The ETE framework uncovers the task flow through the distributed system using the correlation variables [18]. The Magpie performs the same task for a parallel program by tracking invocations of key API functions [7].

Program analysis techniques are also used to automatically construct performance models. LQN models of message-passing programs were automatically built by recovering the request flow from the application trace [26]. A PCM model of the distributed EJB application was constructed using program instrumentation; it demonstrated accuracy $\varepsilon \in (0.1...0.2)$ for the CPU-bound workload [11]. The PACE framework [31] uses static analysis to automatically build performance models of MPI applications with $\varepsilon <= 0.1$ [21]. In [43] authors construct models of distributed scientific computing applications with $\varepsilon \in (0.02...0.3)$.

However, automated construction of performance models is mostly limited to distributed and message passing programs. Accuracy of these models can decrease rapidly if the program performs complex threading operations. For example, the accuracy of the model [43] drops to $\varepsilon = 0.5$ for programs engaged in synchronization operations.

We address this limitation by developing innovative static and dynamic analyses for building performance models of multithreaded programs. Our analyses automatically discover resource demands and thread interactions and translate this information into an accurate model of the system.

The great variety in the types of performance models, programs, and workloads makes it difficult to establish a common baseline to compare accuracy of performance models. Thus we compare our approach to a wider group of models, including analytical and statistical models, and models of distributed systems. These models have $\varepsilon \in (0.02, ..., 0.15)$ for CPU-bound workloads [11],[24],[34],[42],[21],[43], and $\varepsilon \in (0.12, ..., 0.34)$ for I/O-bound workloads [40],[14],[19],[28],[29]. Our models have accuracy $\varepsilon \in (0.032, ..., 0.122)$ for CPU-bound and $\varepsilon \in (0.262, ..., 0.269)$ for disk I/O-bound workloads, which is comparable to the state of the art.

8. SUMMARY

In this paper we presented a novel methodology for automatic modeling of complex multithreaded programs. Our models accurately simulate synchronization operations and hardware usage by multiple threads. At the same time, our framework builds program models automatically and does not require running the program in many configuration.

We verified our approach by building models of various Java applications, including large industrial programs. Our models predicted performance of these programs across a range of configurations with a reasonable degree of accuracy.

Our next steps will be addressing limitations of our models discussed in Section 6, which will allow predicting performance for a wider range of applications and workloads.

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

9. REFERENCES

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

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

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

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

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

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

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proc. of the Symposium on Opearting Systems Design & Implementation*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

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

[9] S. Becker, H. Koziolek, and R. Reussner. Model-based performance prediction with the palladio component model. In *Proc. of the 6th international workshop on Software and performance*, WOSP '07, pages 54–65, New York, NY, USA, 2007. ACM.

[10] M. Bennani and D. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *Proc. of International Conference on Automatic Computing*, pages 229–240, Washington, DC, USA, 2005. IEEE.

[11] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *Proc. International Conference on Automated Software Engineering*, ASE '11, pages 183–192, Washington, DC, USA, 2011. IEEE.

[12] 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.

[13] B.-G. Chun, L. Huang, S. Lee, P. Maniatis, and M. Naik. Mantis: Predicting system performance through program analysis and modeling. *CoRR*, abs/1010.0019, 2010.

[14] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *Proc. of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 337–348, New York, NY, USA, 2011. ACM.

[15] W. Feng and Y. Zhang. A birth-death model for web cache systems: Numerical solutions and simulation. In *Proc. of International Conference on Hybrid Systems and Applications*, pages 272–284, 2008.

[16] G. Franks and M. Woodside. Performance of multi-level client-server systems with parallel service operations. In *Proc. of the 1st International Workshop on Software and Performance*, WOSP '98, pages 120–130, New York, NY, USA, 1998. ACM.

[17] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Proc. of International Conference on Data Engineering Workshops*, pages 87–92, 2010.

[18] J. L. Hellerstein, M. M. Maccabee, W. N. M. III, and J. 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.

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

[20] T. A. Israr, D. H. Lau, G. Franks, and M. Woodside. Automatic generation of layered queuing software performance models from commonly available traces. In *Proc. of International Workshop on Software and Performance*, WOSP '05, pages 147–158, New York, NY, USA, 2005. ACM.

[21] 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.

[22] T. Kelly, T. Kelly, I. Cohen, I. Cohen, M. Goldszmidt, M. Goldszmidt, K. Keeton, and K. Keeton. Inducing models of black-box storage arrays. Technical report, HP Laboratories Palo Alto, 2004.

[23] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Higher Education, 2nd edition, 1997.

[24] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proc. of SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM.

[25] D. A. Menasce and M. N. Bennani. Analytic performance models for single class and multiple class multithreaded software servers. In *Int. CMG Conference*, 2006.

[26] A. Mizan and G. Franks. An automatic trace based performance evaluation model building for parallel distributed systems. *SIGSOFT Softw. Eng. Notes*, 36(5):61–72, Sept. 2011.

[27] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proc. of the 4th international conference on Computing frontiers*, CF '07, pages 143–152, New York, NY, USA, 2007. ACM.

[28] D. Narayanan, E. Thereska, and A. Ailamaki. Continuous resource monitoring for self-predicting dbms. In *Proc. of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 239–248, Washington, DC, USA, 2005. IEEE.

[29] H. Q. Nguyen and A. Apon. Hierarchical performance measurement and modeling of the linux file system. In *Proc. of International Conference on Performance Engineering*, ICPE '11, pages 73–84, New York, NY, USA, 2011. ACM.

[30] H. Q. Nguyen and A. Apon. Parallel file system measurement and modeling using colored petri nets. In *Proc. of International Conference on Performance Engineering*, ICPE '12, pages 229–240, New York, NY, USA, 2012. ACM.

[31] 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.

[32] S. Reiss and A. Tarvo. Automatic categorization and visualization of lock behavior. In *Proc. of the first IEEE Working Conference on Software Visualization*, VISSOFT '13. IEEE, 2013.

[33] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. of the 23rd International Conference on Software Engineering*, ICSE '01, pages 221–230, Washington, DC, USA, 2001. IEEE.

[34] J. Rolia, G. Casale, D. Krishnamurthy, S. Dawson, and S. Kraft. Predictive modelling of sap erp applications: Challenges and solutions. In *Proc. of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '09, pages 9:1–9:9, ICST, Brussels, Belgium, Belgium, 2009. ICST.

[35] N. Roy, A. Dabholkar, N. Hamm, L. W. Dowdy, and D. C. Schmidt. Modeling software contention using colored petri nets. In E. L. Miller and C. L. Williamson, editors, *MASCOTS*, pages 317–324. IEEE, 2008.

[36] J. Y. Stein. *Digital Signal Processing: A Computer Science Perspective*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

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

[38] A. Tarvo and S. P. Reiss. Using computer simulation to predict the performance of multithreaded programs. In *Proc. of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE '12, pages 217–228, New York, NY, USA, 2012. ACM.

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

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

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

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

[43] Q. Xu and J. Subhlok. Construction and evaluation of coordinated performance skeletons. In *Proc. of International Conference on High performance computing*, HiPC'08, pages 73–86, Berlin, Heidelberg, 2008. Springer-Verlag.