Using Computer Simulation to Predict the Performance of Multithreaded Programs

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

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

ABSTRACT

Predicting the performance of a computer program facilitates its efficient design, deployment, and problem detection. However, predicting performance of multithreaded programs is complicated by complex locking behavior and concurrent usage of computational resources. Existing performance models either require running the program in many different configurations or impose restrictions on the types of programs that can be modeled.

This paper presents our approach towards building performance models that do not require vast amounts of training data. Our models are built using a combination of queuing networks and probabilistic call graphs. All necessary information is collected using static and dynamic analyses of a single run of the program. In our experiments these models were able to accurately predict performance of different types of multithreaded programs and detected those configurations that result in the programs' high performance.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of systems—*Modeling techniques*; D.2.9 [**Software Engineering**]: Management—*Software configuration management*; I.6 [**Computing Methodologies**]: Simulation and Modeling

General Terms

Performance

Keywords

Simulation, performance, modeling, configuration

1. INTRODUCTION

Performance is an important characteristic of any software system. It depends on various factors, such as the architecture of the program, properties of the underlying hardware, and characteristics of the system's workload. The performance is also strongly influenced by values of configuration options of the program. Examples of such options can be the size of the internal cache for input-output (I/O) operations or the number of working threads.

Proper understanding of how the configuration of the system affects its performance is essential for many applications. Usually this requires building a *performance prediction model* of the system. This model should be able to predict performance characteristics of the system for different configurations, which include variations in workload, configuration options, and characteristics of the hardware.

Performance prediction models can be useful in various scenarios. During the program's development, such models can estimate the program's performance characteristics and thus help detecting potential problems early [11]. Once the program is deployed, performance models can discover configurations of the system that result in its high performance on a particular platform. Similarly, models may be used for answering "what-if" questions, such as "how will the throughput change if I doubled the number of working threads?"

Performance models become a central component in building self-configuring and autonomic systems [15]. Here prediction results are used to dynamically reconfigure the system to achieve a higher performance. Performance prediction can be used to schedule tasks on high-performance computer systems [12] and large clusters [7]. This allows for both reducing the running time of the program and increasing utilization of the computation resources. Finally, the performance model can be useful in detecting run-time problems with the application. A large discrepancy between predicted and actual performance measurements may be a sign of an anomalous behavior of the system.

Building models of multithreaded programs is not easy. It requires carefully modeling the complex locking behavior of the application and concurrent usage of various computational resources such as the CPU and the I/O subsystem. As a result, existing performance models either impose restrictions on the types of programs that can be modeled or require collecting vast amounts of data about the performance of the system in different configurations. Such limitations often make these models impractical.

Our work attempts to overcome these limitations. We develop an innovative technique that *requires less data to build the model and allows modeling of a wider range of applications.* These models can predict performance of multithreaded programs running in various configurations under the established workload. Our models also predict performance of individual program components and utilization of compu-

ICPE'12, April 22-25, 2012, Boston, Massachusetts, USA

tational resources, which facilitates performance analysis of the system and bottleneck detection.

In our research we use a combination of static and dynamic analyses to collect information about the program. We analyze the program, instrument it at the key points, and run it in a certain configuration to collect the necessary data.

We use a the discrete event approach to simulate computer programs. At the high level we represent a program using a queuing model whose queues correspond to queues and buffers in the program, and whose servers correspond to the program's threads. Each thread is simulated using a probabilistic call graph whose vertices correspond to fragments of the program's code.

For modeling purposes we split the program into fragments; each fragment performing an elementary operation such as CPU-bound computations, I/O operations, synchronizations, buffering, etc. These fragments correspond to appropriate components in the model. This approach contrasts with less generic models, where the program is viewed as a combination of high level constructs specific to the kind of the program being simulated, for example an MPI call.

As a result, our models allow simulating almost any multithreaded program. To further increase flexibility of the modeling, components that simulate underlying OS and hardware are independent of the model of the program. To verify feasibility of our approach we have built models of the simple scientific computing application and the web server running under the Linux OS.

Our work extends the existing state of the art in the area of performance modeling in several aspects:

- our modeling framework is not restricted to simulation of a single class of multithreaded applications and can be used to simulate a wider range of programs;
- we propose a simple yet powerful technique to model I/O operations in the program, including simulation of both hardware and software components of the I/O subsystem;
- we pay strong attention to the proper simulation of the concurrent usage of computation resources.

The remainder of this paper is organized as follows. Section 2 surveys the related work in the area. Section 3 outlines the general approach towards building the model. Section 4 describes the model itself. Section 5 focuses on data collection. Section 6 presents experimental results. Section 7 concludes and outlines directions for future work.

2. RELATED WORK

Existing performance models can be divided into three main classes according to the method they use: analytical models, black-box models, and simulation models. Despite implementation differences, at the high level all of them represent the system as a function $y = f(x)$. Here x are metrics describing system's configuration and workload, and y is some measure of the system's performance.

Analytical models explicitly specify the function $f(x)$ using a set of formulas. N. Bennani and A. Menasce [15] used analytical model in the controller for the autonomic data center. E. Thereska and D. Narayanan [23], [16] used such models to predict performance of the DBMS depending on the size of the buffer pool. These studies report relative error $\varepsilon \leq 0.1$ for predicting throughput and $\varepsilon \in (0.33, 0.68)$ for predicting response time. However, it is not clear if the presented methodology can be easily applied to other applications or if it can incorporate other parameters.

Analytical models are compact and expressive; however, their development requires considerable mathematical skills and deep understanding of the system. Moreover, complex behavior is difficult to describe with analytical models.

Black-box or statistical models are used to alleviate these problems. They utilize no information about the internal structure of the system and do not formulate the function $y = f(x)$ explicitly. Instead, the program is executed in different configurations and its performance is measured. Then some machine learning technique is used to learn the $y = f(x)$ dependency from the data.

In particular, B. Lee et al. [14] used linear regression and neural networks to predict performance of the linear equation solver running on the grid; they report $\varepsilon \in (0.02, 0.11)$. A. Ganapathi et al. [7] used a technique based on the k-NN method to predict performance of the Hadoop queries. They achieved high correlation (R^2 =0.87–0.93) between predicted and actual running times. Their work [8] uses machine learning techniques to predict the execution time of queries in the DBMS.

Disadvantages of the black-box models are the need for large amount of data to train the model and the lack of flexibility. The system must be run in many different configurations in order to collect sufficient amount of training data. Any change to the software or hardware of the system requires re-training the whole model [20]. Furthermore, these models can predict only high-level performance metrics and cannot give an insight into details of the system's performance.

There are attempts to work around these limitations. In particular, large amounts of data can be available directly from a user base [21] or from a large number of runs [19]. Unfortunately, these solution works only for very popular applications and can raise privacy concerns. Chun et al. [6] uses internal program features instead of configuration metrics x to build performance models of CPU-bound programs. This allows reducing the amount of training data for the model, but requires complex program analysis.

Simulation is another popular approach towards performance modeling. The structure of simulation models follows the structure of the system, where components of the system directly correspond to the components of the model. Building these models doesn't require as much data as black-box models, but requires an expert knowledge of the system.

An example of a simulation model is the PACE framework [18] that predicts performance of CPU-bound applications running on a high-performance computer system. PACE was used to predict the execution time of the `nreg` image processing application with the $\varepsilon <= 0.1$ [12]. However, PACE is limited to simulation of MPI-based programs.

One traditional tool used for simulation is Petri networks. Nguen and Apon [17] rely on Petri nets to predict throughput of Linux file system with $\varepsilon \in (0.12, 0.34)$. Gilmore et al. [9] use PEPA networks, a combination of Petri network and PEPA algebra, to build a model of a secure Web service.

Another well-established simulation methodology is queuing networks, which are extensively used for performance modeling of computer networks. However, van der Mei et al.

[24] used queuing network to study impact of networking parameters at the performance of the web server. IRONModel [22] simulates performance of the experimental cluster-based storage system using the combination of queuing model, black-box, and analytical models.

Layered Queue Networks (LQN) extend the queuing networks by introducing the hierarchy of the model components. Xu et al. [27] uses LQN to build a model of a distributed JavaBeans application with $\varepsilon = (0.02, 0.25)$. Israr et al. [11] attempts to automatically build LQN models of commercial message-passing programs from their traces.

Petri nets, queuing models, and LQN are successful in predicting performance of computer networks and distributed programs at the high level. Unfortunately, these models in their classical form fail to properly simulate complex synchronization operations and concurrent usage of computational resources such as CPU and hard drives by multiple threads [26]. Consequently, their applicability to modeling multithreaded applications is limited. In our work we attempt to overcome these limitations.

3. MODEL DEFINITION

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. 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.

This approach naturally allows simulating reactive systems which constitute a majority of modern computer programs. For example, in the web server, the request corresponds to an incoming HTTP connection. The processing of the request includes reading a web-page from the disk and sending it to the user. In the model of the UI application, e.g. a text editor, the request corresponds to a keystroke or a mouse move. The response of the application includes updating its UI and underlying data. Scientific applications can be also simulated in this way. Here, the request can correspond to the object or set of objects in the program. The program responds to the request by performing computations and sending the request to the next component in the system.

We employ a two-tier simulation of computer programs. At the high level, we simulate the program using a queuing network model [13]. At the low level, we simulate threads as probabilistic call graphs. Both types of models are built using discrete-event principles, where the simulation time is advanced by discrete steps and the state of the system does not change between time advances.

The high-level model explicitly simulates the flow of the request as it is being processed by the program, from its arrival to completion. It is a queuing network model where queues correspond to program's queues and buffers, and servers correspond to the program's threads.

Our model differs from the classical queuing networks. First, it does not restrict the structure and properties of the model, such as the number of service nodes n or parameters of the arrival process λ. Second, the server nodes are models on their own that simulate the program's threads. Finally, the high-level model does not explicitly define service demand for requests; these are simulated by the lower-level thread models. Nevertheless, the high-level model is capable of collecting the same performance measures as queuing models, such as R, T, or the number of requests in the system N.

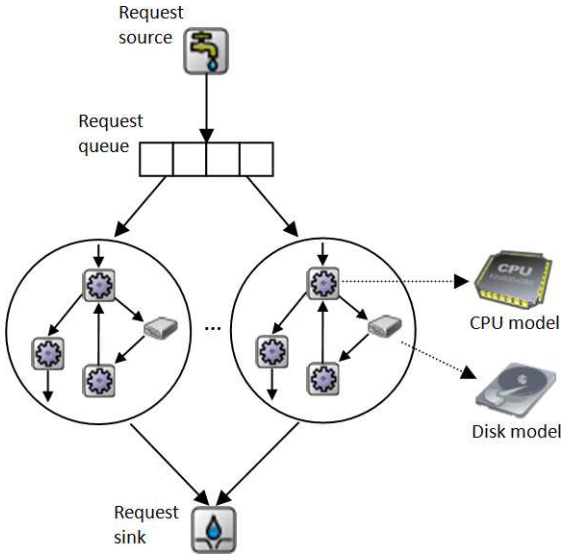

Figure 1: A high-level model for a web server

Figure 1 provides a (somewhat simplified) example of a high-level model for a multithreaded web server. The server works as follows: when the incoming HTTP request arrives, the server calls `accept()` to open a socket for communicating with the client. The request is placed into the queue for incoming connections. Once one of the working threads becomes available, it fetches the request for processing from that queue. The thread verifies that the requested page exists, reads it from the disk and sends it back to the client. When processing of the request is complete, the thread closes a connection and fetches another request from the queue.

The elements of the high-level model reflect basic stages of processing the request by the web server. The request itself corresponds to the socket ID. It is created by a request source, which corresponds to the `accept()` call in the server's code. Correspondingly, the server node (depicted as a circle) denotes a working thread. The server node itself is implemented as a lower-level thread model. Once the processing is over, the request is sent to the sink, which corresponds to closing the client socket by the server.

Lower-level thread models simulate delays that occur in threads when they process requests. Thread models are implemented as probabilistic call graphs, whose vertices S correspond to the pieces of the thread's code – *code fragments*. A special vertex $s_0 \in S$ is a starting code fragment, which is executed upon the thread start. Edges of the graph represent a possible transition of control flow between the code fragments. Probabilities of these transitions are defined with a mapping $\delta : S \to P(S)$.

We distinguish three basic types of code fragments: computation, I/O, and synchronization fragments. These fragments are represented with vertices of the corresponding type in the call graph. Furthermore, we define special vertices s_{in} and s_{out} to communicate with the higher-level queu-

ing model. The input vertices s_{in} fetch requests from the queues of the high-level queuing model. Output vertices s_{out} generate requests and send them to the queuing model.

Execution of each code fragment results in the delay τ. Whereas the call graph structure $< S, s_0, \delta >$ does not generally change in time, *execution times for code fragments depend on various factors, such as the degree of parallelism of the program and characteristics of the underlying hardware*. For example, consider multiple threads that perform equal amount of CPU-intense computations. If the number of threads is bigger than the number of CPUs, the amount of time required for each thread to finish computations will be higher than if that thread was running alone. The same logic applies to I/O operations: the amount of time required for an I/O operation to complete strongly depends on the number and properties of other I/O operations occurring at the same time.

As a result, instead of specifying the exact time τ required for each code fragment to finish, we rather define a set of parameters Π of that code fragment. In turn, Π will be used to calculate τ during the model's runtime.

This necessitates modeling of the underlying hardware and the Operating System (OS). This model will track $Q(t)$ – the state of the whole simulation at each moment of time t. Thread models use this OS/hardware model to compute τ as a function $\tau = f(\Pi, Q(t))$. We have developed models of the CPU/OS thread scheduler and the disk I/O subsystem that compute τ for computation fragments and disk I/O fragments correspondingly.

Different types of code fragments are described by different sets of parameters. A computation fragment is described by a parameter set $\Pi_{cpu} = \{\tau_{cpu}\}$, where τ_{cpu} is the *CPU time* for that fragment. The CPU time is the time necessary to execute the code fragment if it was running on the CPU uninterrupted. A disk I/O fragment is described by $\Pi_{disk} = \{dio_1, ..., dio_k\}$, where $\{dio_1, ..., dio_k\}$ are low-level disk I/O requests spawned by the OS for that I/O fragment.

4. MODEL BUILDING

Using the methodology described above, we have developed a framework to simulate multithreaded programs written in general-purpose programming languages such as C, C++, or Java. We do not restrict ourselves to modeling programs developed using a certain framework (e.g. MPI). We rather develop a generic approach that can be extended, if necessary, to support different frameworks for parallel programming. Nevertheless we pose several important assumptions on the systems we can simulate. First, we do not predict performance characteristics for each individual request. Instead, we predict average performance of the program for a given workload, which can be a mix of requests with different characteristics. Second, we assume that transition probabilities δ remain unchanged across the configuration space of the program. This, in turn, might imply that properties of incoming requests must also remain unchanged in time.

Furthermore, there are limitations caused by the simulation framework in its current state. First, we assume there is no other significant computation or I/O activity in the system, except the program being simulated. Second, currently we can simulate programs running only on the commodity hardware, such as a desktop PC or a general-purpose server. However, these limitations can be alleviated by improving our framework.

We rely on the OMNET [1] simulation framework to build our models. OMNET model consists of interconnected *blocks* communicating using messages. Internally, blocks and messages are implemented as C++ classes. Although OMNET provides general framework for developing those entities, it is a responsibility of the model developer to implement desired functionality in blocks and messages.

High-level and low-level models contain different types of blocks. High-level models contain blocks that represent request sources, sinks, queues, threads, and program-wide locks (barriers, critical sections etc). The high-level model also contains blocks simulating the I/O subsystem and the thread scheduler. Low-level (thread) models consist of computation blocks, I/O blocks, blocks that simulate calls to locks, blocks that read/write data from the high-level model, and dispatch blocks that implement transition probabilities δ.

Each block has a set of parameters that generally represents properties of the corresponding program structures. Values of the block parameters are obtained during the data collection stage. Since values of parameters for a block representing a code fragment can fluctuate across different executions of that code fragment, we treat them as a distributions $\mathbb{P}^{\Pi}$.

The high-level model creates requests, queues them, and sends them to low-level thread models for processing. The request itself is represented as a *request message* flowing from one block to another. The request normally corresponds to some data item in the real-life program, such as a file descriptor, socket ID, a class instance, or a handle. In the high-level model threads appear as "black boxes" without any notion of their internal structure. Each thread is represented as a separate block, such that, if the program has 8 working threads, it has 8 such blocks.

The Figure 2 shows the high-level model for the Galaxy – a simple scientific application. Here requests correspond to Java objects, and arrows depict how requests flow between the model's blocks. The model contains two working threads that interact through two queues, and one main thread.

Details of the thread are simulated by the lower-level thread models. Here the thread is represented as a group of blocks that corresponds to the vertices S of its probabilistic call graph. Execution flow in a thread is also simulated by the message passing. When the model of the program is started, the *computation flow message* (CFM) is created for every thread and is sent to its initial block s_0. Then the computation flow message starts traveling through the call graph of the thread, which simulates the execution of the actual thread. Flow of the message is controlled by the dispatch blocks that reroute the message to other blocks in the model according to the probabilities δ.

The Figure 3 shows thread models for the Galaxy. Here arrows depict flow of the CFM between the threads' code blocks. The CFMs are created using the `sourceOnce` blocks. It must be noted that in addition to blocks that represent fragments of the program's code, thread models also contain service blocks used to control simulation and collect results. In particular, the `setTimer/readTimer` blocks measure the time necessary for the CFM to travel between those, which allows predicting execution time for fragments of the pro-

gram. The `stopper` block stops the simulation after receiving the predefined number of CFMs.

To pass the request message in and out of the thread we rely on a special group of blocks in the thread model – reader and writer blocks that implement s_{in}, s_{out} actions of our formal model. Once the CFM reaches the reader block, that block fetches the request message from the queue or from other source in the high-level model. If no request is available, the reader either delays the CFM until the request becomes available, thus effectively blocking the execution of the thread (which might correspond to a locking behavior in the producer-consumer pattern), or just reroutes the CFM. Correspondingly, when the CFM reaches the writer block, the block outputs the request message to the high-level model. The thread model shown at the Figure 3 (right) has a pair of reader/writer blocks that access a queue in the high-level model.

4.1 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.

To simulate these delays, we employ two groups of blocks: *caller blocks* and *central blocks*. Different types of caller and central blocks are used to simulate different types of delays. However, all of them interact according to the same principle.

Caller blocks are parts of the thread model. When the caller block receives a CFM, it delays that message and notifies the corresponding central block using a separate message. The exact type of that message depends on the type of the caller/central block pair. The parameters of the message are sampled from the $\mathbb{P}^{\Pi}$ – distribution of parameters for the caller block.

The central block is a part of a high-level model. Once it receives the 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 message back the caller block. The caller block in turn sends the original CFM to the next block in the thread model.

4.1.1 Simulating synchronization delays

To simulate high-level synchronization primitives in the program the modeling framework employs a wide range of different block types. 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 centra/caller block pair. For example, the barrier is represented by a SyncBarrier central block; calls to that barrier are represented by SyncBarrier_await caller blocks.

These 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.

The high-level model shown at the Figure 2 contains four central blocks that represent barriers and one central block that represents the critical section. Correspondingly, the thread models at the Figure 3 contain caller blocks that call these central blocks.

4.1.2 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 predicts 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 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. Once the CPU/Scheduler receives a message from the computation block, it puts that message into the queue of "ready" threads. When one of the computation cores of the simulated CPU frees, the CPU/Scheduler takes the first message out of the "ready" queue and simulates computations by introducing a delay whose length is equal to $min(\tau_{cpu}, OS\ time\ quantum)$. After the delay is expired, the CPU/Scheduler either sends the message back to the origin block (in case computations are complete) or places it back into the "ready" queue, where it awaits another time quantum. The length of the time quantum is sampled from the distribution that represents quantum length of the actual Linux thread scheduler.

In our example the high-level model contains the `cpuScheduler` block, which implements the model of the CPU/ Scheduler. Each of thread models contain one computation block that call `cpuScheduler`.

4.1.3 Simulating Disk I/O

To simulate delays that occur because of the disk I/O the model uses a combination of DiskIOOperation (caller block) and DiskIOModule (central block). DiskIOModule simulates the disk I/O subsystem of a computer and also predicts 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 disk I/O message is initially sent to the model of the I/O scheduler. If the hard drive

is idle at the moment, the I/O scheduler model sends the request directly to the hard drive model. Otherwise, the disk I/O message is placed in the queue that simulates the request queue of the actual I/O scheduler. When the hard drive model becomes available, it fetches the next request to be processed from that queue. The real I/O scheduler orders requests according to the index of the disk block they are accessing, but since this information is not known to the model, the hard drive model fetches requests from the random positions of the queue.

The model of the hard drive calculates the disk processing time τ_{disk} for the request and delays the request for that time. However, τ_{disk} depends on parameters of the request, such as the amount of data transferred, locality of the operation (how close are disk sectors accessed by different requests etc), and other factors. To account for those, the model treats τ_{disk} as conditional distribution $P(\tau_{disk}|x_{dio})$, where x_{dio} are parameters of the I/O request. As for now we use the type of the request (synchronous read, metadata read, read-ahead) as a parameter, since it implicitly represents the locality of the I/O operation. In particular, read-ahead requests usually do not require the disk seek operation and, thus, have significantly shorter τ_{disk}. Currently, we are working on incorporating other request parameters to increase the overall accuracy of simulation.

4.2 Simulating OS limits

OS can impose a variety of limits on the program such as the maximum number of open file descriptors. These limits can severely affect the program's behavior and can be viewed as additional parameters of the system.

To simulate OS limits we have also implemented a combination of a central block and caller blocks. A code fragment that attempts to acquire some resource (such as a call to `accept()` or `open()` functions that acquire a file descriptor) is represented by a 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. OSLimits updates the state of the system and notifies the caller if the resource was granted or not. The result of the call is logged by the caller block for the further analysis. Moreover, it can be used to reroute the CFM if the request was denied, thus simulating the behavior of the real program.

5. DATA COLLECTION

Building models of the multithreaded program requires collecting following information about the program itself as well as the underlying OS and the hardware:

- information on thread interaction in the program, including synchronization mechanisms and request queues;
- probabilistic call graphs $< S, s_0, \delta >$ for all the threads;
- parameters Π of individual code fragments;
- performance characteristics of the underlying OS and hardware.

To collect this data we analyze the system, instrument it, and run it in *one specific configuration*. This is a major advantage over the black-box methods, which require running the program in a large number of configurations.

We utilize a mixed approach towards program analysis. We manually analyze the program at the high-level to establish its structure and use automated solutions to obtain the rest of the data.

During the manual analysis we determine the general sequence of operations that happen during the request processing. First we identify synchronization mechanisms and working threads. Then we analyze the threads' code to detect code fragments and determine their types. Next, we instrument the program by inserting probes at the borders of individual code fragments. Each probe is identified by the unique ID, thus each code fragment can be uniquely identified by the pair of IDs of surrounding probes. Program instrumentation completes manual analysis of the program. The rest of the data collection is performed automatically.

To collect information on code fragments we run the instrumented program in one representative configuration. When the probe is hit during the program's execution, we record CPU time τ_{cpu} and wallclock time τ for the code fragment. Our instrumentation is very lightweight: every probe slows the execution of the program in average by 1-2 microseconds. To further decrease instrumentation overhead, we rely on statistical sampling [10].

Once execution of the program has finished, the instrumentation log is analyzed automatically and the following information is retrieved:

- τ_{cpu} for all code fragments, which forms $\mathbb{P}^{\Pi}_{cpu}$;
- transition probabilities δ for each thread;
- τ for all code fragments;
- performance metrics of the program, such as the response time R, throughput T etc.

τ and performance metrics (R, T) are used solely for analyzing simulation results and model debugging.

Unfortunately, the user-mode log does not include information on I/O operations and page cache usage. To capture this data we instrument following places of the Linux kernel using the SystemTap framework [2]:

- start and end of the system call routines that can initiate I/O requests, such as `sys_read()` or `sys_stat()`;
- the `generic_make_request()` function, which inserts the request for the low-level I/O operation into the queue of the I/O scheduler;
- the `blk_start_request()` function, which is called when the I/O scheduler passes a request to the physical device (a hard drive);
- I/O completion routines.

This instrumentation yields the following measurements:

- the number k and properties of I/O requests $\{dio_1, ..., dio_k\}$ issued by the system call. Properties of the request include type of the request (synchronous read, metadata read, read-ahead), and amount of data transferred. Altogether this data comprise the distribution of parameters $\mathbb{P}^{\Pi}_{disk}$ for the corresponding disk I/O fragment;

- τ_{disk}: the amount of time required to process the I/O request by the hard drive. τ_{disk} is calculated as the time difference between the call to the `blk_start_request()` and the completion routine. τ_{disk} is used to build the model of the I/O subsystem;
- τ_{io}: total time required for I/O request to complete, which is the sum of the time spent in the I/O scheduler queue and τ_{disk}. τ_{io} is calculated as the time difference between the call to the `generic_make_request()` and the completion routine. τ_{io} is used to verify the model of the I/O subsystem.

The number of I/O requests k is an important parameter of an I/O code fragment, as it implicitly represents the probability of hitting the OS page cache. As the value of k varies for different requests, it should be considered a random variable.

It has been shown [25] that the cache of a constant size can be represented as a birth-death process. If there are no changes in parameters of the incoming requests, such a process converges to the equilibrium state, where the probability of a cache hit remains constant. Unfortunately, in the general case the size of the OS page cache can vary. But if our initial assumptions (absence of other significant activity in the system and no rapid changes in the workload) hold, the size of the OS page cache should not change. As a result, the probability of cache hit should also remain the same.

To validate this claim we conducted a series of experiments with the web server that uses `stat()` and `read()` functions to access data on the hard drive. These experiments confirmed that, after serving a large number of requests, the system reaches an equilibrium state and the distribution of k does not change. This is an important observation as it allows us to easily simulate the OS page cache. To bring a system into an equilibrium state we issue a large number (around 10^5) of "warm-up" requests prior to taking actual measurements of k.

6. MODEL VERIFICATION

To test our approach for performance prediction we built models of two multithreaded programs. These programs are very 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. The second program is tinyhttpd, a disk I/O-bound web server. Instrumented source code and models for these programs can be downloaded from [3]. We have created other models which are not presented here due to space constraints.

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 simulate the program in same configurations and record predicted performance. Then we calculate relative error ε between measured and predicted performance metrics as

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

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 the Ubuntu Linux 10.04 OS.

6.1 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 [5] 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 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 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.

Configuration options of the Galaxy include two parameters that directly influence performance of the program: the number of force threads and the number of collision threads. To cover all the configurations that are adequate for our hardware we experimented with 1,2,4,6,8,12 and 16 force threads and collision threads. Overall configuration space included 49 configurations, which includes all possible combinations of the number of force and collision threads.

We ran Galaxy in all 49 configurations. In every configuration Galaxy was used to simulate 10000 bodies for 1000 iterations (in all the experiments we assume that our test workloads are representative). To get reliable measurements of the program's performance three runs were performed in each configuration. Iteration lengths R were recorded during each run; their mean value $\overline{R}$ was used as an *actual* value of $R_{measured}$ for each configuration of the Galaxy.

One noteworthy finding is that the Galaxy iteration length depends mostly on the number of force threads, and only slightly on the number of collision threads. This can be explained by the fact that force calculations are significantly more expensive than the collision checks. However, the overall performance of the Galaxy highly depends on the number of collisions. As simulation time advances, many objects are getting merged during the collisions, so the Galaxy must perform less computations. This observation is important since the model must accurately simulate such a complex behavior.

We manually built both high-level and thread models of the Galaxy. The high-level model is shown at the Figure 2. Upon model initialization requests 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_forces-`

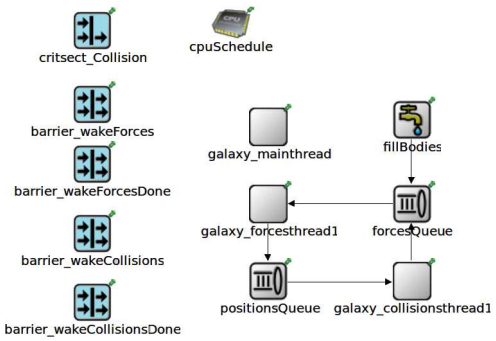

Figure 2: OMNET model of the Galaxy (high-level)

`thread` and `galaxy_collisionthread` blocks represent the force thread and the collision thread, while the `galaxy_mainthread` block represents the main thread of the program.

Low-level thread models for the Galaxy are shown at the Figure 3 (the model of the collision thread is not shown due to the lack of space). 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.

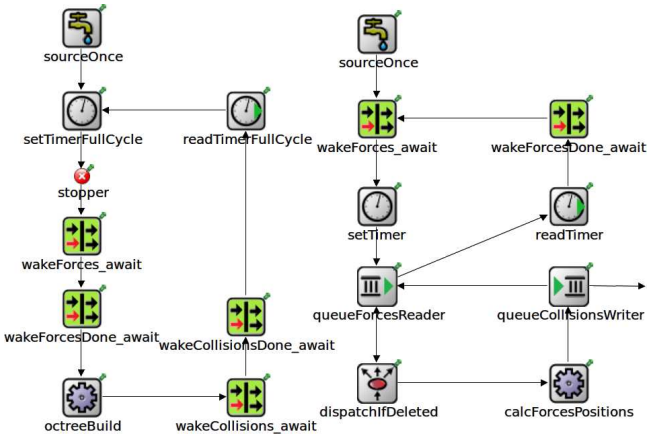

Figure 3: OMNET models of Galaxy: main thread and force thread

The model of the force thread uses the `queueForcesReader` reader block to fetch requests from the `forcesQueue`. Then the `dispatchIfDeleted` dispatch block simulates a check if the body has been marked as collided with another body. If the collision has occured, the corresponding request 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 request to the `positionsQueue` using the `queueCollisionsWriter` writer block and attempts to fetch a new request 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.

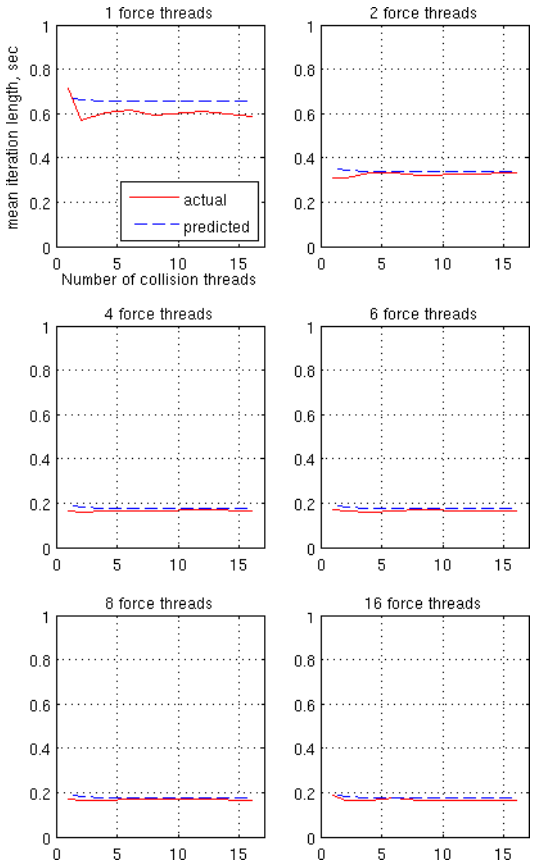

Figure 4: Experimental results for the Galaxy

Finally, `wakeForces_await` block suspends the thread until it is waken up by the the central thread during the next iteration.

To define parameters for the low-level thread models we instrumented 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`) $= 5.40 \times 10^{-3}$ sec, and $\overline{\tau}_{cpu}$ (`calcForcesPositions`) $= 4.33 \times 10^{-5}$ sec.

We used the model to predict the iteration length of the Galaxy in each configuration. Similarly, the model was ran three times in each configuration and the average iteration lengths was used as a predicted value $R_{predicted}$. The comparison of actual and predicted iteration lengths is shown at the Figure 4.

Table 1: 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

Table 2: 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

The relative error 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 [14], [6]. Relative errors for all the configurations are listed in the Table 1.

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 of the program. In particular, it correctly points that the number of force processing threads must be >= 4, while the number of collision threads has no significant impact on Galaxy performance.

Table 2 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.

6.2 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. In our work we built the model of a tinyhttpd multithreaded web server [4]. 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 large class of server applications.

When the tinyhttpd receives an incoming request, it puts the request into the queue until one of its working threads becomes available. The working thread then picks the request 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 it sends the "Internal Server Error" response. Once data transfer is complete, the working thread closes the connection and picks up the next incoming request 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, `atime` functionality was disabled to improve performance of the server. We used the http_load software [6] to simulate client connections to our web server. 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 we used to measure the performance of a web server was the response time R. We define R as a time difference between accepting the incoming connection and sending the response (more accurately – closing the communication socket). In addition to R we also measured 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. By varying the IRR we simulate 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; every run was repeated six times to get averaged results for each configuration.

Depending on the values of IRR the web server has two distinct states of operation (see Figure 5). 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 negated by a proportional increase in the average execution time for each individual I/O operation. We believe this example illustrates necessity for the proper simulation of I/O operations, as they often becoming a determining factor in the program's performance.

To build the model of the tinyhttpd, we instrumented its code with 21 probes and ran it in the configuration with 4 threads and IRR=70 rps. Our model predicts the R for stationary states reasonably well ($\varepsilon \leq 0.30$), but its accuracy decreases at the point of transition (see Table 3). 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. Unfortunately, the model predicts this metric with somewhat lower accuracy: at the

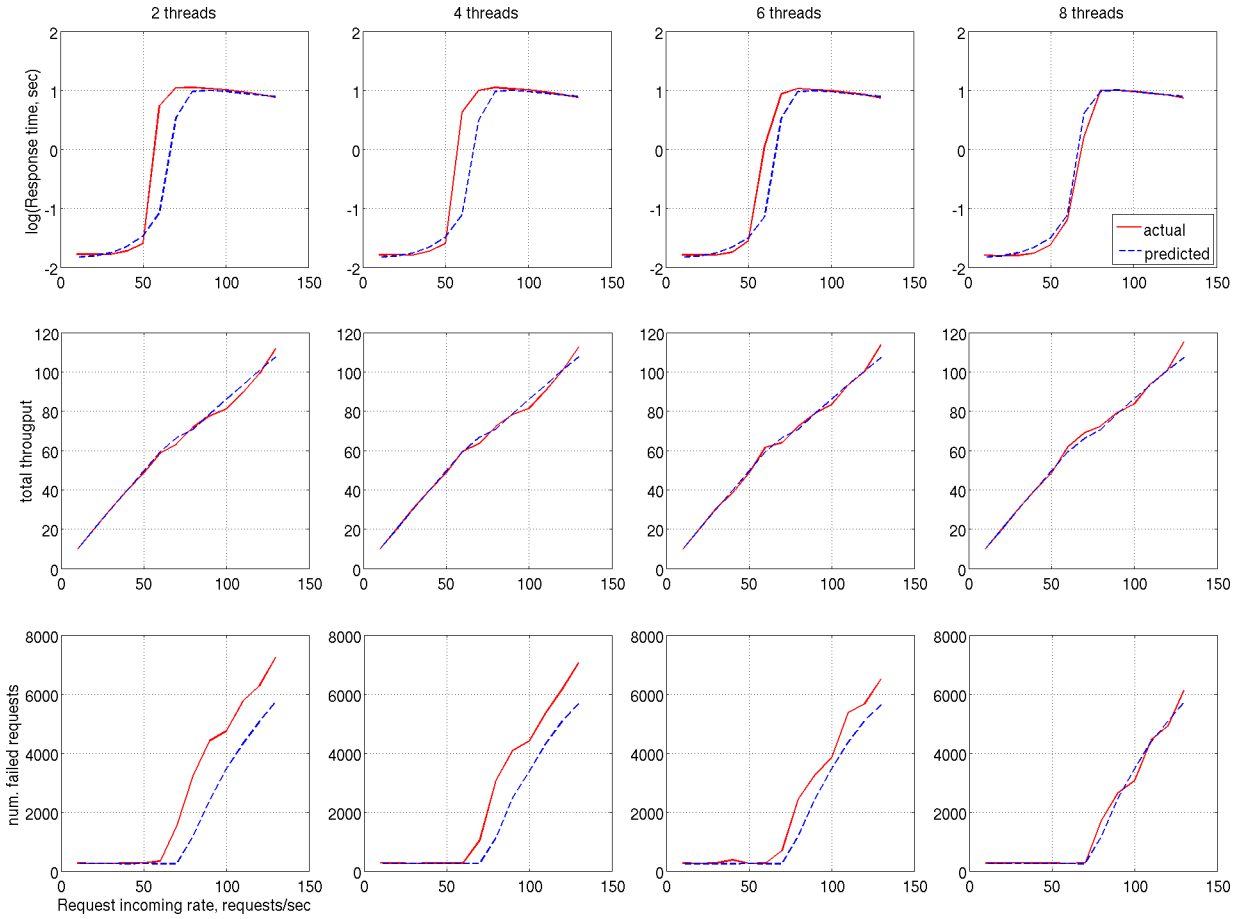

Figure 5: Experimental results for the tinyhttpd. top row: the response time R (logarithmic scale); middle row: the throughput T; bottom row: the number of error responses

transition point $\varepsilon = 1$ and average error $\overline{\varepsilon} = 0.214$. 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. We expect to further improve accuracy by developing a more sophisticated I/O model. Nevertheless, even with the current simple model our results are comparable to those obtained from a more refined model of the Linux I/O subsystem [17].

Furthermore, the *model 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.

The output of the model is not limited to the high-level performance metrics such as R and T. It can predict execution time for individual code fragments or groups of code fragments. In particular, for tinyhttpd it also predicts the overall time required for the working thread to process the request ($\varepsilon \in (0.001, 0.340)$, $\overline{\varepsilon} = 0.074$), time necessary to complete for `stat()` ($\varepsilon \in (0.044, 0.456)$, $\overline{\varepsilon} = 0.212$) and `read()` ($\varepsilon \in (0.001, 0.539)$, $\overline{\varepsilon} = 0.151$) calls. This enables us to use the model for finding unobvious bottlenecks in the program. For example in our case it was discovered that a single `stat()` call can take as much time as reading the whole file in 1kb blocks. Furthermore, the model produces readings of hardware resource usage, such as average CPU or hard drive load. Unfortunately, space limitations do not allow us providing elaborated results for these metrics.

7. CONCLUSIONS AND FUTURE WORK

In this paper we presented our approach for modeling performance of the multithreaded computer programs. We pay special attention to simulating concurrent usage of the underlying OS and hardware by multiple threads. As a result, our models do not require extensive amounts of training data and do not pose significant restrictions on the types of programs that can be modeled.

We implemented our methodology in practice by developing an extensive end-to-end framework for simulating multithreaded programs, which includes tools for data collection and model building. Finally, we verified our approach by building models of both CPU- and I/O-bound multithreaded programs.

Our model accurately predicts the performance of multithreaded programs with a high degree of accuracy. The model also allows predicting performance of individual program components and usage of computation resources. More importantly, *our models accurately predict those configurations of the program that result in its high performance.*

Table 3: 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

These results encourage us to continue working on simulation of computer programs. Most importantly, we plan automate building the models of programs. This would allow us building models of complex applications such as industrial web and DBMS servers, search servers and crowlers. As a first step, we plan developing tools for automatic discovery of I/O and synchronization routines in the program. This would allow instrumenting the program's code and building probabilistic graphs of the working threads without human intervention. As a more distant prospective, we plan to automate building high-level queuing models of the program.

Moreover, we investigate different approaches towards I/O modeling. We plan developing a purely statistical model of the disk I/O. This model should allow simulating various types of the hardware, such as RAID arrays, and provide higher accuracy. Similarly, we plan developing a model for network I/O since, in certain scenarios, network delays can become determinant of the program's performance.

8. REFERENCES

[1] `http://www.omnetpp.org/`.

[2] `http://sourceware.org/systemtap/`.

[3] `http://cs.brown.edu/~alexta/PERSIK.html`.

[4] `http://sourceforge.net/projects/tinyhttpd/`.

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

[6] 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.

[7] 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 (ICDEW)*, pages 87 –92, march 2010.

[8] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proc. of International Conference on Data Engineering*, pages 592–603, Washington, DC, USA, 2009. IEEE Computer Society.

[9] S. Gilmore, J. Hillston, L. Kloul, and M. 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.

[10] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proc. of International Conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 156–164, New York, NY, USA, 2004. ACM.

[11] 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.

[12] 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.

[13] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance, Computer System Analysis Using Queuing Network Models*. Prentice Hall, 1984.

[14] 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.

[15] M. N. Bennani and D. A. Menasce. Resource allocation for autonomic data centers using analytic performance models. pages 229–240, 2005.

[16] 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 Computer Society.

[17] 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.

[18] 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.

[19] K. Singh, E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Predicting parallel application performance via machine learning approaches: Research articles. volume 19, pages 2219–2235, Chichester, UK, December 2007. John Wiley and Sons Ltd.

[20] D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora. A framework for measurement based performance modeling. In *Proc. of International Workshop on Software and Performance*, WOSP '08, pages 55–66, New York, NY, USA, 2008. ACM.

[21] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical performance models for complex, popular applications. In *Proc. of SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '10, pages 1–12, New York, NY, USA, 2010. ACM.

[22] E. Thereska and G. R. Ganger. Ironmodel: robust performance models in the wild. In *Pro. of SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, pages 253–264, New York, NY, USA, 2008. ACM.

[23] E. Thereska, D. Narayanan, and G. R. Ganger. Towards self-predicting systems: What if you could ask "what-if"? *Knowl. Eng. Rev.*, 21:261–267, September 2006.

[24] R. van der Mei, R. Hariharan, and P. Reeser. Web server performance modeling. *Telecommunication Systems*, 16:361–378, 2001. 10.1023/A:1016667027983.

[25] Y. Z. Wenying Feng. A birthâĂŞdeath model for web cache systems: Numerical solutions and simulation. *Nonlinear Analysis: Hybrid Systems*, 2:272–284, 2008.

[26] X. Wu and M. 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.

[27] 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.