Using computer simulation to predict performance of parallel programs

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

Abstract — **As computer programs are becoming more complex, the number of their configuration options increases as well. These options must be carefully configured to achieve high performance of the system. We propose using computer simulation to predict performance of the program and select its configuration resulting in maximum performance.**

This work presents our methodology and tools for building models of multithreaded programs and underlying hardware. We rely on a combination of static and dynamic analysis to build the model. Parameters of the model include incoming request flow, configuration options of the program, and hardware parameters. The model predicts performance of the program for each combination of parameters. As a result, the configuration resulting in high performance is discovered without expensive and time-consuming experiments with the live program.

I. INTRODUCTION

As size and complexity of modern software grows, programs are becoming increasingly hard not only to design and develop, but also to deploy. Their real-world performance depends not only on architecture of the program itself or on the underlying hardware, but also on values of numerous configuration options. Examples of these options can be size of the internal cache for input-output (I/O) operations or the number of working threads.

In order to achieve maximum performance configuration options must be carefully tuned by a system administrator. Currently system administrators rely on their professional experience and on basic monitoring tools for analyzing various program configurations. For example, they might decide that upgrading a CPU is necessary if the CPU load will exceed a given threshold, or that the number of working threads might be increased if the CPU has multiple cores. However, such approach is not accurate and requires a thorough understanding of both hardware and software of the system as well as details of its anticipated usage. Thus in practice system administrators perform multiple experiments (trial runs) to select an optimal configuration of the system.

During trial runs software is launched with different configuration options. The configuration resulting in the highest performance is selected for practical deployment. Unfortunately, such experiments are time-consuming and put a high workload on a system administrator. In addition to the actual running time of the experiments, he has to set up trial runs, collect data, and analyze results. Altogether, these activities result in a high cost of trial runs. Moreover, trial runs might require a special hardware or software not available at the present moment, which will effectively render this approach useless.

To reduce cost and time necessary for finding an optimal program configuration we propose conducting trial runs using the model of the program. The model will predict performance of the program for the given set of parameters, including incoming request flow, configuration options, and characteristics of the hardware. A number of different configurations will be tested; the configuration resulting in the highest performance will be selected as an optimal one for the actual program.

Performing trial runs using the model will have following advantages over experimenting with the live program:

- Once the model is created, running it does not require complicated and time-consuming operations for setting up the experimental system;
- Model runs faster than the actual program, which results in significant time savings;
- The model can find a good configuration automatically by searching through the configuration space of the program;
- The model can simulate execution of the program on the different hardware and/or operating system (OS).

In this paper we present our methodology and infrastructure for building models of computer programs. In particular, we concentrate on simulating multithreaded programs used for request processing, such as web servers.

We rely on a dynamic analysis to collect data for building the model of the program. We instrument both the program and the OS kernel, and then run the program in a certain configuration. As a result, data necessary for building the model are collected, including running time for individual components of the program, number and properties of I/O requests spawned by the program, and its probabilistic call graph. Benchmarking is used to collect data for the model of the underlying OS and hardware.

We use the OMNET simulation framework [2] to build the model of the program. We employ a modular approach: models are built from a number of small components; each component performing a simple operation such as computations, I/O activities, flow control etc. Components responsible for simulation of hardware and OS are

independent from the model of the program. These components are, in fact, separate models built using a combination of a discrete event simulation and statistical modeling. Modular architecture allows simulating execution of the program on different computer systems without changes in the structure of the model.

To verify our approach we have built the model of the tinyhttpd web-server [5] running on the OS Linux. In our experiments with different number of working threads and different incoming request rate the relative prediction error of the model varied in range of 3-85% with the average error 19.9%.

The rest of this paper is organized as following. Section 2 surveys the related work in the area. Section 3 outlines a general approach towards building the model. Section 4 focuses on data collection. Section 5 describes the model itself. Section 6 presents results obtained by simulating a web server. Section 7 concludes and outlines directions for future work.

II. RELATED WORK

Predicting performance of computer systems is a subject of active research, and simulation and modeling are actively used for this purpose. Existing performance models can be divided into three classes according to the method they use: analytical models, black-box models and simulation models.

Analytical models represent the system as a function $y=f(x)$, where x are metrics describing system's configuration and workload, and the output y is some measure of system's performance. Analytical models rely on knowledge of the system to explicitly specify function $f(x)$ using a set of formulas.

Analytical models were popular in modeling performance of both hardware [3] and software [14,15]. They are compact and expressive; however, their development requires considerable mathematical skills and deep understanding of the system. Moreover, complex behavior can be hardly described by analytical models.

Black-box models are called to alleviate these problems. They utilize no information about internal structure of the system and do not formulate the function $y=f(x)$ explicitly. Instead, a statistical [8] or table-based [17] method is used to approximate the function $f(x)$ and predict the performance of the system based on its input parameters.

The black-box approach does not require an extensive expert knowledge of the system, although it needs large amount of data for training the model. Its main disadvantage is lack of flexibility, as any change to the software or hardware of the system requires re-training the whole model. Nevertheless the black-box approach is frequently used to simulate a part of the system, be it a software component such as a DBMS query engine [9], or hardware component, such as a hard drive [11-13]. The black-box approach can also predict the performance of the whole program if a large amount of data is available either through multiple runs [23] or from a large user base [10].

Simulation models (also called **queuing models**) are another popular approach towards performance modeling. Structure of these models normally follows the structure of the system, where components of the model directly correspond to the components of the system. Usually these models are built using a **discrete-event** principle. In such models the simulation time is advanced by discrete steps; it is assumed that the state of the system does not change between time advances [16].

Developing simulation models require an expert knowledge about the system, which can be considered as a disadvantage. Nevertheless, simulation models are highly popular tool for simulating large complex systems, largely because their flexibility and extendibility. These models are widely used to predict performance of computer systems at different levels, ranging from communication networks [1,25,26] and large distributed computer systems [18-22] to individual components [24]. They can be also combined with other types of the models, e.g. analytic models [20].

Simulation and modeling has a plenty of applications, but in the context of our work we are especially interested in simulation of parallel and distributed systems and underlying hardware.

In particular, simulation has been successfully used to predict performance of distributed programs. The PACE framework [21,22] predicts running time of scientific application on a grid, while Layered Queue Networks [18,19] are used to predict performance of commercial multi-tier systems. All these works assume that the system uses a message passing framework for communication. This simplifies the model, but reduces the set of programs that can be modeled. Unfortunately, neither of these works provides any quantitative measure of model's accuracy.

Works [14, 15] presents a concept of a self-predicting program. The program is manually instrumented in the key points, and its analytical model is built. Authors use this approach to predict throughput of the DBMS depending on the size of the buffer pool. Study [15] reports relative prediction error within 10-68%. However, it is not clear if the presented methodology can be easily applied for other applications or if it can incorporate other parameters. E. Thereska and G. Ganger [20] address this issue by combining multiple types of the model, such as machine learning, analytical and queuing models to simulate a distributed storage system. The resulting model is used to detect performance problems in the system as they occur.

An alternate approach towards performance prediction is presented in [10]. The release copy of the program is instrumented manually, so performance measurements are collected directly from the user audience. Statistical techniques are used to calculate performance metrics for a given combination of workload, configuration options and hardware. Obviously, this approach works only for programs with a large user base (such as MS Office).

Modeling and simulation was also successfully used to predict performance of hardware components, in particular – hard drives.

DiskSim [24] was one of the first tools that rely on simulation models to predict response time for individual I/O requests. It simulates mechanical and electrical components of the hard drive in detail and achieves a good accuracy. However, DiskSim requires information that might not be readily available, such as physical layout of files on the disk or current position of the disk head.

Call for simplified storage system simulators resulted in development of black-box models. These models do not require any information on the internal structure of the storage system, which is a major advantage. They describe each individual request with metrics and then use some statistical technique, such as regression trees [4, 13] or tables [17] to predict service time for that request. These models report relative prediction error in a range of 10-70%. Such large errors are unavoidable because access times highly depend on location of requested data and position of the disk head. In absence of that information access times should be treated as a distribution rather than scalar numbers.

To some extent, this problem is addressed in the work [11] that uses Bayesian approach towards the disk performance modeling. This work clearly identifies factors affecting the request servicing time; it is used to classify requests into “slow” and “fast”. The reported accuracy of classification ranges from 54% to 96%. The model, however, assumes that the requests processing time distributed independently, which is not necessary true in the actual system.

Unfortunately, all referenced storage system models simulate only the hardware; they do not simulate behavior of the system’s cache and I/O scheduler – components that largely determine I/O performance of the application.

Our work extends 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 application or scenario and can be used to simulate a wide range of programs;
- Our model pays strong attention to the proper simulation of the thread-level parallelism in a multithreaded application;
- We propose an innovative technique to model I/O operations of the program, including simulation of both hardware and software components of I/O subsystem;
- Our paper presents an elaborate description of technical aspects of model building along with concrete metrics of simulation accuracy.

III. GENERAL APPROACH TOWARDS BUILDING THE MODEL

Our approach towards simulation of computer programs follows a conventional **discrete-event** principle. We rely on discrete-event models because of their expressive power, relative simplicity, ease of interpretation, and rich toolset for building such models.

For the purpose of simulation we represent computations performed by the program as request processing. We denote a **request** as an external entity the program has to react to, such as incoming user connection or data packet. The program **processes the request**, which involves performing certain operations. For example, the web-server can read a web-page from the disk and send it to the user, and scientific application can perform computations and send results to the next component of the system. Overall delay between request arrival and its completion constitutes a **request processing time** – an important metric we use to measure performance of the system. Predicting request processing time is the main goal of our model.

Although this approach naturally allows simulating request processing programs, it can be used to simulate other programs, including user applications. For example, in the context of a text editor a keystroke can be considered a request, to which the program responds by updating its UI and underlying data.

Our model explicitly simulates the flow of the request as it is being processed by the program, from its arrival to

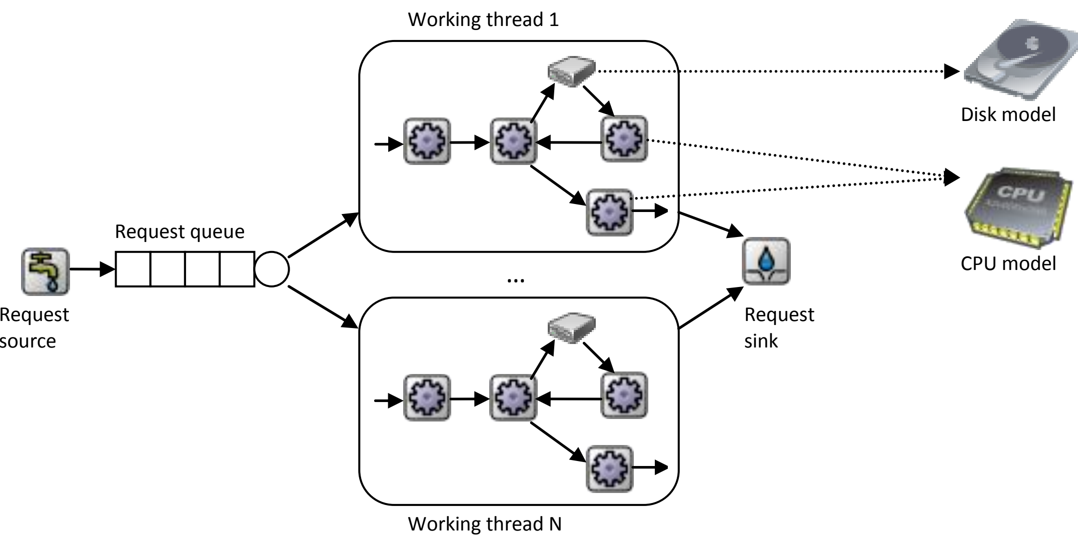

Figure 1. General structure of the model of the system

completion. Here we assume that every request in the program is represented by the corresponding data item such as a socket ID or an object representing the request. Correspondingly, processing the request implies manipulation with this data item, such as placing it into a queue, performing computations, or spawning subsequent requests, e.g. requests to the I/O subsystems (disk, network).

All request processing activities occur in the context of a thread. In our model we represent each thread as a separate entity (see Figure 1). Inside, the thread is represented as a probabilistic call graph whose nodes correspond to pieces of program's code – **code fragments,** while edges correspond to possible transitions between code fragments. In particular, we distinguish two major types of code fragments that perform CPU-intense computations and I/O operations correspondingly.

Execution of each code fragment results in the delay in the processing of the request by the program (we assume that code performing synchronization operations does not result in significant delays due to computational or I/O activities). These delays altogether with time spent by the request in various queues and buffers constitute the request processing time. Thus building the model requires following knowledge about code fragments:

- what code fragments constitute the program and in which order they are executed;
- exact time required by each code fragment to execute.

Whereas the structure of the program does not generally change, execution times 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 CPU-intensive 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 I/O operation to complete strongly depends on the number of other I/O operations occurring at the same time.

As a result, instead of specifying exact time required for each code fragment to finish, we rather define parameters of that code fragment. These parameters along with the actual workload of the system determine execution time of the code fragments. This time will be calculated using models of the CPU and the disk I/O subsystem. The former model simulates work of the CPU and the OS thread scheduler; it computes the amount of time required for CPU-intensive code fragments to execute. The latter one simulates the OS I/O scheduler and the hard disk itself; it computes the amount of time required for I/O-intense code fragments to execute. The network I/O model is a subject of the future work.

IV. DATA COLLECTION

Building models of the parallel program, CPU/scheduler, and I/O subsystem requires collecting comprehensive information on the program itself and also on the underlying OS and the hardware. This includes:

- information on thread interaction in the program, including synchronization mechanisms, request queues etc;
- probabilistic call graphs for threads;
- properties of program's code, such as amount of computational and I/O resources necessary for its execution;
- performance characteristics of the underlying OS and hardware.

To collect this data we analyze the program, instrument, and run it in one specific configuration. The key assumption is that main parameters of the system remain the same for all other 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 rest of the data.

During 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 threads' code to detect code fragments and determine their types (CPU-intense or I/O-intense).

Once overall program structure is established, we instrument the program by inserting probes at the borders of individual code fragments. Currently instrumentation is done at the source level: the instrumentation library is statically linked to the program under the study and then **probes** (which are calls to the library's functions) are inserted into the source code of that program. Each probe is indentified by the unique ID. As a result, each code fragment can be uniquely identified by the pair of IDs of surrounding probes. The list of code fragments and corresponding probe IDs form the program's schema, which is used for the automated analysis of the program.

Program instrumentation and defining its schema complete manual analysis of the program. Rest of data collection is performed automatically by the tools we developed for this purpose.

To collect information on code fragments we run the instrumented program in some configuration. When the probe is hit during the program's execution, it creates the record that includes probe ID, the ID of currently executing thread, and current CPU and wallclock times for the thread. Our instrumentation is very lightweight: every probe slows the execution of the program in average by 1-2 microseconds, depending on the amount of information collected.

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

- the amount of CPU time required to execute each code fragment;
- the probabilistic call graph of the program;
- the total amount of time required to execute each code fragment (wallclock time);
- request processing time for the program.

Last two metrics are used solely for analyzing simulation results and model debugging.

Values of all these metrics can change over different executions of the same block, thus we treat them as distributions.

However, data obtained from the instrumented program do not include information on I/O operations initiated by the program. This information is essential for modeling the program itself and the I/O subsystem. Building the model of the program requires knowledge of the number and properties of I/O requests initiated by the program. Building the model of the I/O subsystem requires data on how quickly these requests can be processed and probabilities of serving requests from the cache.

Initially we tried to retrieve necessary data by monitoring calls to user-mode libc I/O functions, such as read() and write(). Unfortunately, this approach doesn't allow tracking I/O operations initiated by functions such as stat() and open(). These functions spawn multiple I/O operations of different types that access filesystem metadata (inode and directory entries). To reliably monitor these requests we intercept I/O operations in the OS kernel.

Below we discuss processing of I/O requests by the OS kernel in the context of monitoring I/O requests. In particular we concentrate on the Linux kernel v. 2.6.32 that was our testing platform.

When the user program calls a function such as read() or stat(), the control is transferred to the corresponding *system call* in the Linux kernel – sys_read() or sys_stat() correspondingly. The kernel determines if the requested data reside in the OS page cache. If they are, the kernel fetches data from the cache and returns them to the user program without accessing the hard drive.

However, if the cache does not contain the requested data, the kernel must read data from the hard drive (this situation is depicted at the Figure 2). To do this the system call relies on the *generic block layer* – a kernel component responsible for handling all the I/O operations on block devices. This results to a call to the generic_make_request() function that creates an I/O request of a given type. The generic_make_request() function sets up the completion routine that will be called once the data transfer is complete, passes the I/O request to the I/O scheduler, and suspends the calling thread.

The function of the *I/O scheduler* is to minimize the processing time for I/O request and, at the same time, increase total throughput of the disk. Although there are various I/O schedulers in Linux, all of them follow roughly the same principle.

The scheduler stores pending I/O requests in the queue ordered by the index of the requested disk block. This allows to minimize the disk seek time – the main parameter affecting performance of the hard drive. When the scheduler receives a new request, it first attempts to merge it with one of existing requests. If the request cannot be merged, scheduler inserts it into the appropriate position of the request queue.

Once the hard drive becomes available, its driver fetches the new request from the I/O request queue and sends it to the device. At this time the I/O scheduler calls a blk_start_request() function to notify the block I/O layer that the request was sent to the hard drive. When the hard drive finishes processing the request, the I/O scheduler calls the completion routine for that request. It wakes up the calling thread that completes the I/O operation.

To collect necessary data we use SystemTap framework [7] to insert probes into following places:

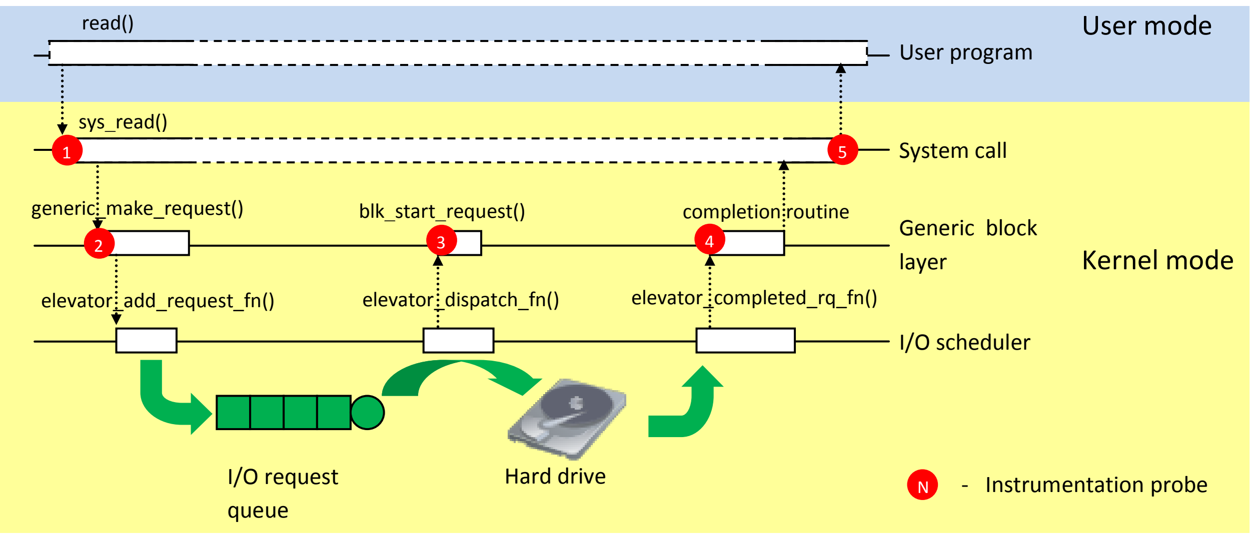

Figure 2. Instrumentation of Linux I/O subsystem

- start and end of the system call routine (probes 1 and 5 at the Figure 2). We instrument numerous system calls that can initiate I/O request, including sys_read(), sys_stat(), sys_stat64(), sys_statat() etc;
- generic_make_request() function (probe 2);
- blk_start_request() function (probe 3);
- end_bio_bh_io_sync() and mpage_end_io_read() completion routines (probe 4).

This instrumentation yields various measurements, the most important of those are:

- the number and properties of requests issued by the system call. Calculated as the number of calls to generic_make_request() made by the system call without taking into account merged requests;
- time required to process the request by the hard drive (**disk processing time**). Calculated as the time difference between the call to the blk_start_request() and completion routine.
- time required for I/O operation to complete **(total processing time**). Calculated as the time difference between the call to the generic_make_request() and the completion routine.

Similarly to parameters of the code fragments, parameters of I/O operations are treated as distributions.

Essentially, we employ benchmarking to collect information on the I/O subsystem. We collect data on I/O requests during the same program run when we collect data on program's code fragments. Moreover, since in the case of a cache hit no requests will be issued, the "number of requests" metric also implicitly represents caching behavior of the system. Our experiments with the web-server revealed that values of this parameter do not depend on the configuration of the program, but depends on the total number of requests processed so far (see Figure 3). After serving a large number of requests the system reaches some "steady state" in which the number of requests issued by the system call remains constant. We rely on this observation to simulate the system's page cache. In order to reach the "steady state" we issue a large number (typically – around 10^6) of "burn-in" requests prior to taking actual measurements of this metric.

V. MODEL BUILDING

We evaluated a number of tools for building discrete-event simulations, including JavaSim, MATLAB SimEvents, OMNET and other tools. Finally we have chosen OMNET [2] because of its high flexibility.

According to OMNET principles, the model consists of blocks connected to each other and 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.

In our model blocks represent elements of the program; most of the types of model blocks have direct analogs in the program itself (see Appendix 1 for a complete list of blocks). Each block has a set of parameters that generally correspond to properties of the corresponding program structures. Values of the parameters are obtained during the data collection stage. The vast majority of parameters are distributions, so when the block intends to get a value of the parameter, the value is sampled from a corresponding distribution.

Our model is built according to a hierarchical principle. At the high level the model depicts general flow of the request through the system (see Figure 4). High-level model creates requests, queues them, sends them to threads for processing and, when the processing is done, destroys requests.

The request itself is represented as a message flowing from one block to another. The request normally corresponds to some data item in the real-life program, such as file or socket ID, class instance, or handle.

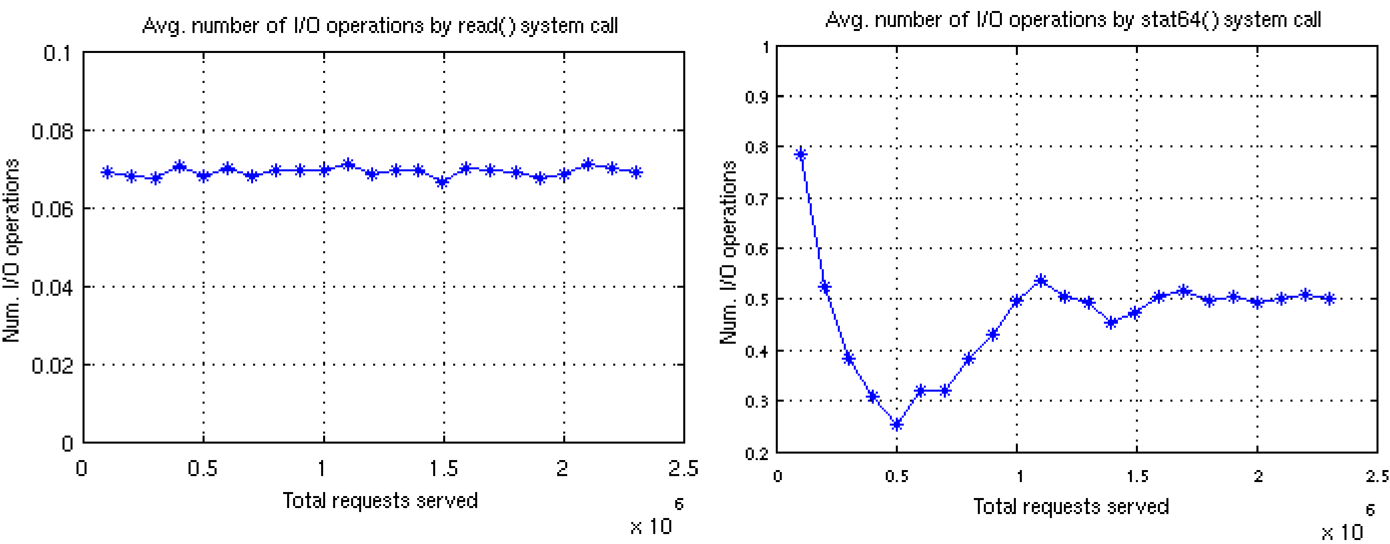

Figure 3. The number of I/O requests issued by the system call as an implicit characteristic of the page cache

Threads are central elements of the model. They simulate delays that occur during processing of the request. At the high level 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.

Details of the thread are simulated by the lower-level model (see Figure 5). On the lower level thread is represented as a group of blocks forming the probabilistic call graph of the thread. In this graph the caller block is connected (through a special dispatch block) to all the potential callees.

Execution flow in a thread is simulated by message passing. When the thread receives the request for processing, it creates a computation flow message and sends it to the first computation block in the thread. This message passes through the thread blocks until it reaches the last computational block. At this point processing of the request by the thread is considered complete and request is sent to the next block in the high-level model.

High-level and low-level models contain different types of blocks (see Appendix 1). High-level models contain request sources, sinks, queues and threads, while low-level (thread) models contain computation and I/O blocks and flow control blocks.

Blocks representing code fragments communicate with CPU/Scheduler and I/O models using messages. When the computation block is called, it sends the message to the CPU model. This message contains the amount of CPU time required to execute the corresponding code fragment as a parameter. Correspondingly, the I/O block sends one or more messages representing I/O requests to the I/O model. Parameters of the I/O request include the amount of data to be transferred and the type of the operation (synchronous read, metadata read or readahead). The corresponding model calculates the amount of time required to finish the operation and delays the request for that amount of time.

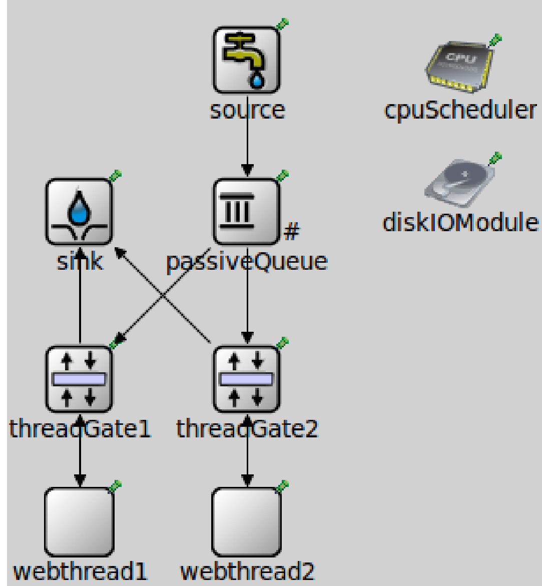

Figure 4. The high-level model of the web-server

As a result, the model of the system consists of two major independent components: the model of the program itself and models of the OS/hardware. This architecture allows simulating the same program running on different hardware and vice versa.

In our work we employ various approaches to simulate different types of hardware. We use traditional discrete-event approach to simulate CPU and OS thread scheduler, while a combination of discrete event simulation and statistical modeling is used to simulate disk I/O. We put a number of assumptions about the underlying system which we believe are true for the most of server-side programs and scientific computing applications:

- Except for the program we simulate, all other computation and I/O activities in the system are negligibly small;
- all the threads in the program have the same priority;

These assumptions greatly simplify simulation of the hardware.

A. *CPU and thread scheduler modeling*

The CPU/Scheduler model simulates the round-robin thread scheduler with equal priority of all the threads.

Once the CPU/Scheduler receives a message from the computation block, it puts that message in the queue of "ready" threads. When one of the computation cores of the simulated CPU frees, CPU/Scheduler takes the first thread out of the "ready" queue and simulates computations by

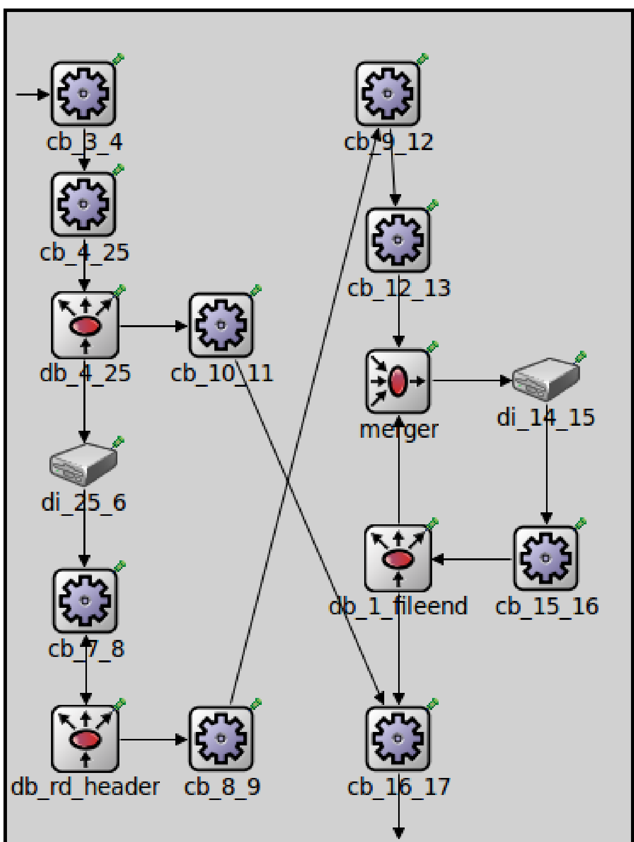

Figure 5. The low-level model of the web-server thread

introducing a delay. The length of delay is equal to the amount of CPU time required for the computation or to the OS time quantum, whatever is smaller. After the delay is expired, the CPU/Scheduler either sends the message back to the origin block (in case if computations are complete) or places it back into the "ready" queue, where it awaits for another time quantum. The length of the time quantum is sampled from the distribution that represents quantum length of the actual Linux thread scheduler.

As it can be seen, the model closely follows the functioning of the real thread scheduler. Our model of the CPU scheduler doesn't have any provisions for tracking threads suspended due to I/O or synchronization because these activities are simulated differently by the model.

B. *Disk modeling*

Our model of the I/O subsystem consists of two parts. The first part simulates the I/O scheduler using the discrete event model. The second part simulates delays that occur during the processing of the request by the hard drive (disk processing time) using the statistical model.

When the I/O block sends a request for disk I/O, this message is received by the I/O scheduler model. If the hard drive model does not process any request at the moment, the I/O scheduler model sends the request to the hard drive model directly. Otherwise the I/O request is placed in the queue that simulates the request queue of the actual I/O scheduler. When the hard drive model frees, it fetches the next request to be processed from that queue.

The model of the I/O scheduler employs FIFO queue, where requests are ordered by the time of their arrival. However, the real I/O scheduler orders requests according to the index of the disk block they are accessing. Since this information is not known to the model, the hard drive model fetches requests from the random positions of the request queue.

The model of the hard drive calculates the disk processing time t for the request and delays the request for that time. The model assumes that t follows the conditional distribution $P(t|x)$, where x are request parameters (metrics). In particular, we use two metrics to describe the request:

- the number of other requests sent to the hard drive by the I/O scheduler between enqueuing the given request and sending it to the hard drive;
- the type of the request (synchronous read, metadata read, readahead).

These parameters account for possible optimizations done by the I/O scheduler. The first parameter implicitly represents the queue size of the I/O scheduler. With the large number of I/O requests waiting in the queue, the scheduler can arrange them more efficiently, so the average disk processing time for each individual request will decrease. The second parameter accounts for the possibility that different types of requests require different time to process. In particular, we noticed that readahead requests are served significantly faster than the synchronous read and metadata requests.

Since both distribution parameters are integer numbers, we implement the distribution $P(t|x)$ as a table.

Our I/O model is very simple, but it represents behavior of the deadline I/O scheduler fairly well. We are currently working on simulation of other types of I/O schedulers, such as anticipatory and CFQ schedulers.

VI. MODEL VERIFICATION

In order to be useful, the model must accurately predict performance of the system. To estimate accuracy of the model we run the program in different configurations and record *actual* performance of the program for each configuration. Afterwards we simulate the program in the same configurations and record *predicted* performance. Then we calculate relative error ε between measured and predicted performance metrics as $\varepsilon = \frac{|actual - predicted|}{actual}$. 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 the relative error will be equal to 0.

In this work we have built the model of a *tinyhttpd* multithreaded web server [5]. When the web-server receives the incoming request, it puts it 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 requested file exists, the thread reads the file in 1024-bytes chunks and sends them to the client. Once data transfer is complete, the thread closes the connection and picks up the next incoming request from the queue. This web-server is simple and compact, which facilitates its analysis, but at the same time it is representative for a large class of server applications.

The web-server hosts 200000 static web pages from the Wikipedia archive. It runs on a server PC equipped with an Intel Q6600 quad-core 2.4 GHz CPU, 4 GB RAM and 160 GB hard drive. The server runs under Ubuntu Linux 10.04 OS. According to the common practice, *atime* functionality was disabled to improve performance of the server.

We use the *http_load* software [6] to simulate client connections to our web server. http_load reads a list of URLs from the file and then connects to the web-server to retrieve these pages. httpd_load is running on a client computer (Gateway laptop with Intel 2.4 GHz dual-core CPU, 4 GB RAM, 250 GB HDD) connected to the server with a 100 MBit Ethernet LAN. The client runs under Ubuntu Linux 9.10 OS.

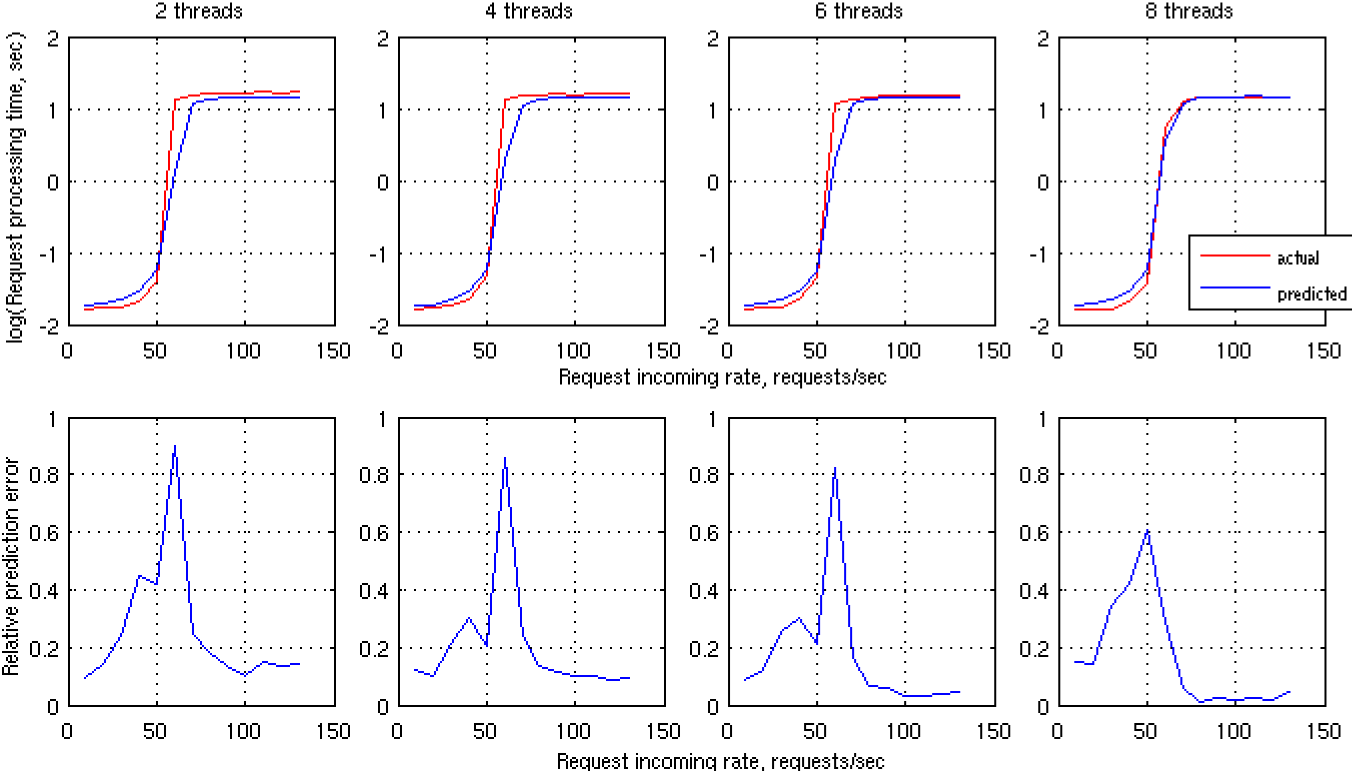

Figure 6. Results for predicting the request processing time

The request processing time (RPT) is the main metric we use to measure the web-server performance. We define RPT as a time difference between accepting the incoming connection and sending the response (more accurately – closing the communication socket). In addition to the response time, we also collect execution time for different code fragments of the program. A particular interest present execution times for I/O operations, as they often determine performance of the program.

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 run both the actual program and its model and record average values of performance metrics. During each run 10,000 requests are issued; this experiment is repeated three times to get averaged results for each configuration.

The behavior of the web-server varies greatly for different IRR values (see Figure 6). The web-server has two distinct states of operation. For low values of IRR (IRR<50 rps) the I/O subsystem is not fully utilized and the request processing time is minimal (RPT varies within 10-20 ms). High values of IRR (IRR≥60 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 system-wide limit of 1024 open connections and starts dropping incoming requests. At this point the RPT reaches 14-17 sec. and remains steady.

Our model predicts the request processing time for these stationary states reasonably well ($\varepsilon \leq 0.3$), but its performance decreases at the point where the web-server goes to the saturation state ($\varepsilon = 0.6$-0.85). However, the model accurately predicts values of configuration parameters where this transitional behavior occurs. This result is important, since the ultimate goal of our research is not just predicting performance of the program, but finding the point in the configuration space that yields high performance.

One noteworthy finding of our experiments is that the number of working threads has a relatively small influence on the request processing time. 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

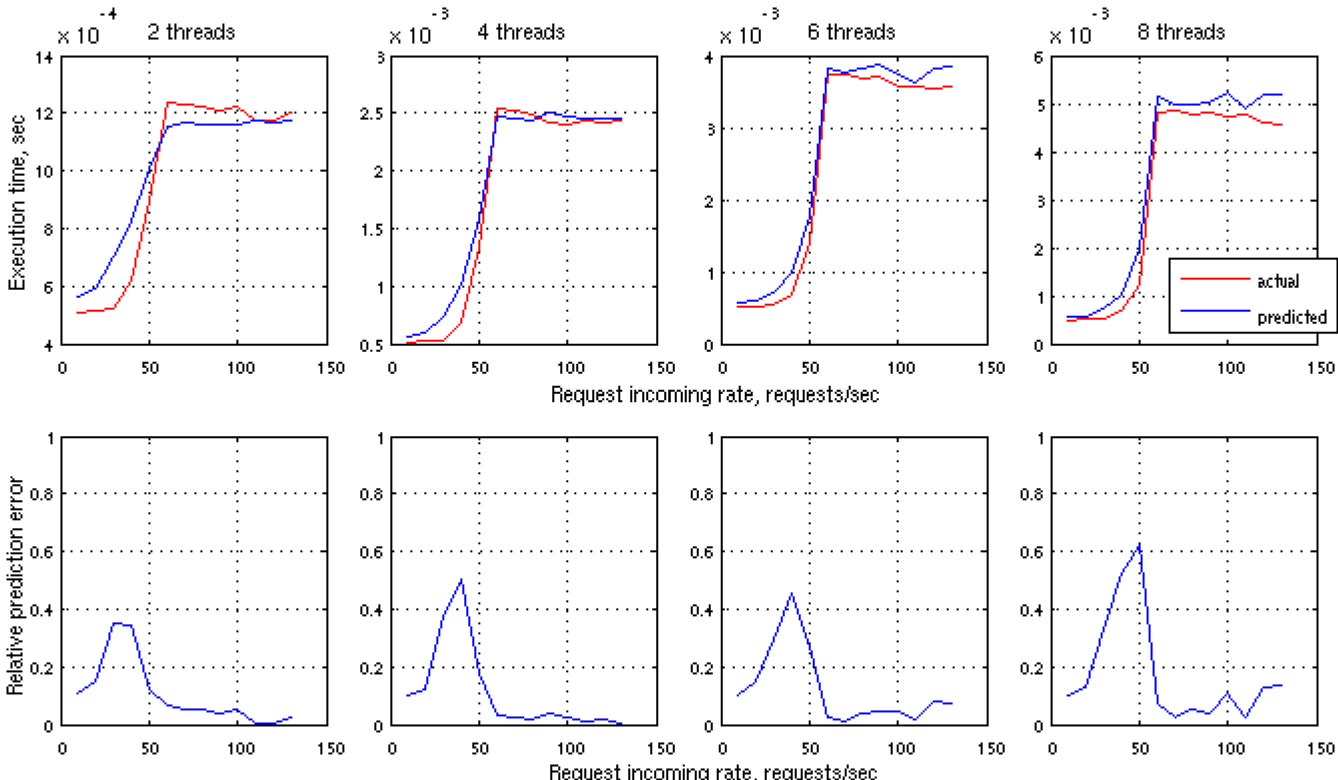

Figure 7. Results for predicting execution time for the read() I/O operation

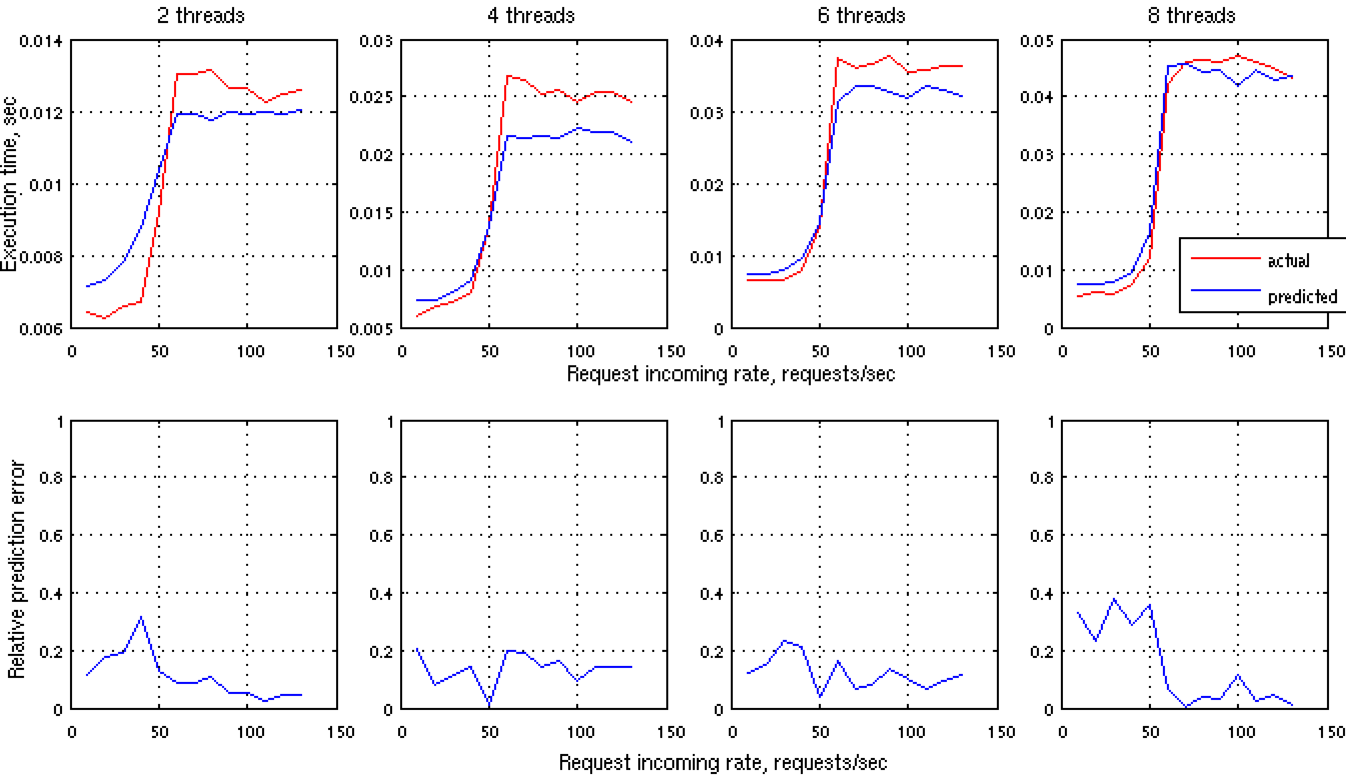

Figure 8. Results for predicting execution time for the stat() I/O operation

the average execution time for the I/O operation (see Figures 7,8). Our I/O model correctly predicts this behavior. The average error for predicting execution times ranges from 0.10 to 0.18 for read() and 0.11 to 0.15 for stat() 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.

VII. Conclusions and Future Work

In this paper we presented our methodology and the toolset for modeling performance of the multithreaded computer programs. We developed an extensive end-to-end simulation framework, which includes tools for data collection and model building. We also developed methodology and metrics for measuring the accuracy of our performance models. Finally, we verified our approach by building the model of the web-server.

Our model predicts the performance of the web-server with a reasonable degree of accuracy: average relative error varies from 0.164 to 0.256 for different configurations of the web-server. Even more important, **the model accurately predicts those configurations of the program where its behavior changes significantly.** This allows finding configurations resulting in high performance, which was the main goal of our work. Furthermore, the model predicts execution times for different parts of the program, which allows identifying performance bottlenecks. One particular interest is predicting execution times for I/O operations, since they often determine the program's performance.

Although our experiments have shown good results, an extensive experimentation is required to verify all the aspects of our work. In particular, in our recent experiments with the web-server I/O operations dominate over CPU computations. As a result, the accuracy of the simulation is largely determined by the accuracy of the I/O model. Thus to validate our model of the CPU/Scheduler we are working on the model of the computationally intensive program. In particular, we are building the model of the program that simulates gravitational interaction of celestial bodies.

Aside of conducting additional experiments we are actively working on improving our methodology and tools for model building. Main directions of our work are:

- improving framework for model building;
- ensuring model portability across different hardware;
- reducing the amount of human involvement;

A. Improving framework for model building

Improving framework for building simulations will increase accuracy, simplicity and usability of models.

In particular we are investigating different approaches towards I/O modeling. Our current model of the I/O subsystem still requires substantial knowledge of the I/O scheduler, which can be seen as disadvantage. Thus we plan to employ a purely statistical model that will simulate the total processing time of the request by the I/O subsystem. We expect this model to require less sophisticated data collection and allow simulating various types of the hardware, such as RAID arrays. Similarly, we plan to develop a model for network I/O since in certain scenarios network delays can become determinant of the program's performance.

B. Ensuring model portability across different hardware

We rely on benchmarking to retrieve characteristics of the hardware and integrate them into models. Currently we employ this approach to simulate I/O system; in the future we plan to use benchmarking for modeling computational activities.

However, benchmarking is a time consuming activity that requires access to the hardware we want to simulate. A simple solution would be establishing a repository of benchmarks. Model builders could use this repository to find data on the hardware which is the most similar to one they simulate. A more attractive alternative would be incorporating widely used characteristics of the hardware as parameters into the model. For example, disk model can include disk rotation speed and the average seek time, while CPU model can use publicly available results of industrial benchmarks as model parameters. This approach would eliminate necessity for time-consuming benchmarks, but certain types of hardware, such as RAID arrays, would require altering the structure of the model. Thus a combined approach might be employed, where certain components of the model, such as CPU, would be built using hardware parameters, while other components such as disk or networking will be built using benchmarking.

C. Reducing the amount of human involvement

Currently our models are built manually, which is a major inconvenience. Our long-term goal is developing an automated way of building models, or, at least, decreasing the amount of human involvement in this process.

As a first step, we plan to automate building of probabilistic graphs for working threads. We will intercept calls to functions that correspond to I/O or to synchronization routines. This would allow us to detect all potentially blocking operations and represent them as either I/O or synchronization code fragments in our model. Remaining code will be represented as computational fragments. This approach should automate building models of working threads; however, it will not detect objects representing requests. Thus there might be still a certain amount of human involvement in the process of building models.

Another totally different approach toward simulation would be representing the program as an automaton with the corresponding set of states. One particular type of state will be a starting state, which corresponds to the initial state of the program when the incoming request is received. While processing the request, the program will change its states according to some random distribution, until it reaches one of the final states, where request processing is completed or request is dropped. Every state change will require a certain amount of time, which will be also represented as a random variable. State information can be relatively easily extracted from the trace data generated by the instrumented program. However, there remains an open question on what

information should be included into the state (e.g. name and offset in the current function, variable values etc).

VIII. References

1. B. Aslam, M. Akhlaq, S. A. Khan, IEEE 802.11 wireless network simulator using Verilog, 11th WSEAS International Conference on Communications, Greece, 2007.
2. http://www.omnetpp.org/
3. E. Lee , R. Katz, An analytic performance model of disk arrays, Conference on Measurement and modeling of computer systems, p.98-109, Santa Clara, CA, 1993
4. M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, G. Ganger, Storage device performance prediction with CART models, 12th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2004), Volendam, The Netherlands, 2004
5. http://sourceforge.net/projects/tinyhttpd/
6. http://www.acme.com/software/http_load
7. http://sourceware.org/systemtap/
8. B. Lee et al, Methods of inference and learning for performance modeling of parallel applications, 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, San Jose, CA, 2007
9. C. Gupta , A. Mehta , U. Dayal, PQR: Predicting Query Execution Times for Autonomous Workload Management, International Conference on Autonomic Computing, p.13-22, 2008
10. Eno Thereska, Bjoern Doebel, Alice X. Zheng, Peter Nobel, Practical performance models for complex, popular applications, International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'10), New York, NY, 2010
11. T. Kelly, I. Cohen, M. Goldszmidt, and K. Keeton. Inducing models of black-box storage arrays. Technical Report HPL-2004-108, HP Labs, 2004.
12. S. Li, H. Huang, Black-Box Performance Modeling for Solid-State Drives, 18th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS) , pp.391-393, 2010
13. L. Yin, S. Uttamchandani, R. Katz. An empirical exploration of black-box performance models for storage systems, 14th IEEE International Symposium on Modeling, Analysis, and Simulation, pages 433-440, Washington, DC, USA, 2006.
14. E. Thereska, D. Narayanan, and G. R. Ganger. Towards self-predicting systems: What if you could ask 'what-if'?, 3rd International Workshop on Self-adaptive and Autonomic Computing Systems, August 2005.
15. D. Narayanan, E. Thereska, A. Ailamaki. Continuous resource monitoring for self-predicting DBMS, International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Atlanta, GA, September 2005
16. J.B. Sinclair. "Simulation of Computer Systems and Computer Networks: A Process-Oriented Approach". Feb.15, 2004
17. Eric Anderson. Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-4, HP Labs, 2001
18. C. Hrischuk, J. Rolia, C. Woodside, Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype, 3rd Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '95), Durham, NC, 1995
19. T. Israr, D. Lau, G. Franks, M. Woodside, Automatic Generation of Layered Queuing Software Performance Models from Commonly Available Traces, 4th Int. Workshop on software and Performance (WOSP 05), 2005.
20. E. Thereska, G.R. Ganger. IRONModel: robust performance models in the wild, ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Annapolis, MD, 2008
21. G. R. Nudd et al, "PACE- A Toolset for the performance Prediction of Parallel and Distributed Systems", International Journal of High Performance Computing Applications (JHPCA), Special Issues on Performance Modelling- Part I, 14(3): 228-251, SAGE Publications Inc., London, UK, 2000.
22. S. Jarvis, D. Spooner, H. Keung, G. Nudd, J. Cao, and S. Saini, Performance prediction and its use in parallel and distributed computing systems, International Parallel and Distributed Processing Symposium, 2003.
23. K. Singh, E. _Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Predicting parallel application performance via machine learning approaches. Concurrency And Computation: Practice and Experience, 19(17):2219-2235, 2007.
24. http://www.pdl.cmu.edu/DiskSim/
25. http://www.nsnam.org/
26. http://tetcos.com/software.html
27.

Appendix 1. Model blocks and their parameters

Model block	Description	Thread block?	Model icon	Analogy in the program	Important parameters
Request source	Generates requests	N		Socket for incoming connections	- request interarrival time
Queue	Queues requests	N		Buffer that queues requests	- maximum queue size; - queue type (FIFO/LIFO)
Sink	Discards requests and collects statistics about their processing, such as request processing time.	N		N/A	- N/A;
Computation block	Code fragment that performs CPU-intensive computations	Y		Code fragment	- CPU time required for computations
Disk I/O block	Code fragment that performs disk I/O	Y		Code fragment	- types of I/O requests (read, metadata read, readhaead); - number of I/O requests of each type; - amount of data for each type of requests;
Dispatch block	Routes the request to a different block based on transition probabilities	Y		IF statement, loop	- probability of sending a request to a particular block
Loop block	Sends the request to a given block(s) for a number of times	Y		for() loop	- Number of iterations
Delay	Delays a processing of request for a given time	Y		sleep() function	- delay time;
Thread gate	Separates blocks that form a thread from remaining blocks in the model. This block does not have a direct analog in the real program;	Y		N/A	- thread ID;
I/O subsystem	Calculates the amount of time necessary for the I/O operation and delays processing of the request for that time	N		I/O scheduler and the hard drive	- parameters of P(t|x) distribution
CPU/ Scheduler	Calculates the amount of time necessary for the CPU-intense computation and delays processing of the request for that time	N		Thread scheduler and the CPU	- the number of CPUs (cores)