Automatic Performance Modeling of Multithreaded Programs

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

ABSTRACT

Multithreaded programs express a complex non-linear dependency between their configuration and the performance. To better understand this dependency performance prediction models are used. However, building performance models manually is time-consuming and error-prone. We present a novel methodology for automatically building performance models of industrial multithreaded programs.

Keywords
Performance, model, program analysis

General Terms
Performance

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

1. PROBLEM AND MOTIVATION

Contention of hardware resources and synchronization operations cause parallel execution of some parts of the multithreaded program and sequential execution of others. This results into a non-linear dependency between a program’s configuration and performance.

Performance prediction models help understanding this dependency. They predict performance of a program for a given configuration. Configurations include properties of the hardware, characteristics of the workload, and options of the program itself e.g. the number of working threads.

Building practical performance models is hard. Modern programs are large and complex, so are their performance models. Thus performance models must be built automatically. However, building such model requires understanding the structure of the program, resource demands by its individual components, and the semantics of thread interactions.

We employ a combination of a static and dynamic analyses to extract information about structure and semantics of the program. The extracted information is translated into the discrete event model of the program, which is simulated using the OMNET [1] framework. We verified our methodology by building models of two industrial programs: the Apache Tomcat web server and the Sunflow 3D renderer.

2. BACKGROUND AND RELATED WORK

Performance prediction can be done using analytic [11][4] and statistical [8][10] models. Special-purpose approaches, such as LQN [13], Petri Nets [2], and Palladio models [3] are more capable methods for predicting performance of complex programs. Some of these models can be solved analytically. However, discrete event simulations offer higher degree of flexibility in simulating complex behavior [7][5].

Most of existing work concentrate on automatic generation of models for message passing programs [12] and distributed systems, which may not perform locking and I/O activities [6][9]. In this role their prediction error not exceed 10-15%. But applying these methods to modeling large multithreaded programs may result in decrease in accuracy [15].

Previously we demonstrated that a 3-tier model can accurately model performance of a multithreaded program [14]. The high-level tier simulates data flow in the program. It is a queuing model, where queues represent program’s queues and buffers, and service nodes represent the program’s threads and thread pools. The mid-level tier simulates delays that occur within the threads. It is a probabilistic call graph, whose vertices $s_i \in S$ represent code fragments (CFs) and edges represent possible transitions of control flow between these CFs. CF corresponds to a contiguous fragment of the thread’s code that performs either a CPU-intense computation, a disk I/O, or a synchronization operation. Additionally, $s_{in}$ and $s_{out}$ nodes read and write requests to the queues of a high-level model. The lower-level tier includes models of the locks presented in the program, the model disk I/O subsystem, and the model of CPU and thread scheduler.

Executing each CF $s_i$ results in delay $\tau_i$. To simulate effects of resource contention we do not specify $\tau_i$ explicitly. Instead, every time the CF needs to compute $\tau_i$, it calls a corresponding low-level model and passes it parameters $\Pi$ of that CF. Parameters vary for different CF types: for a computation CF $\Pi_{CPU}$ is the amount of CPU time required to execute that CF; for a disk I/O CF $\Pi_{disk} = (dio_1, \ldots, dio_k)$ are the number $k$ and properties of low-level I/O operations; for a synchronization CF $\Pi_{lock}$ are the reference to the lock, type of the operation (e.g. barrier.await), and timeout.
3. GENERAL APPROACH

We automatically build our 3-tier models using a combination of static analysis, and dynamic analysis. This allows generating a performance model automatically from running the program in a single representative configuration.

Building the model requires information about the program. Representing the program as a queuing system requires knowledge of queues, threads and thread pools (corresponding to service nodes in the model), and their interactions (correspond to $s_{in}/s_{out}$ CFs). Detecting thread pools is necessary because their sizes are major factors affecting performance of the program. Building the call graph for a thread requires discovery of CPU-bound computations, I/O and locking operations in a program (correspond to CFs), and determining sequence in which they are executed. Modeling delays $\tau_i$ requires knowledge of CFs parameters. Semantically correct simulation of synchronization requires knowledge of locks, their types (e.g. is the lock a barrier or a mutex), and parameters (e.g. capacity of the barrier).

We retrieve the necessary information in three stages: stack sampling, the static analysis, and the dynamic analysis. All stages are completely automated.

During the stack sampling stage we find thread pools and library usage. We execute the program and periodically take snapshots of its stack. Stack snapshots along with thread usage information are merged into a call trie. The trie is used to compute thread pools of the program.

During the static analysis stage we analyze all code of the program that can potentially execute and detect CFs in it. Synchronization CFs are detected from calls to synchronization routines. I/O CFs are obtained from I/O routines. $s_{in}$ and $s_{out}$ CFs are detected from reads and writes to the program queues. Remainder of the program’s code is considered as computation CFs.

During the dynamic analysis stage we detect locks and queues in the program, obtain parameters of CFs, and re-construct call graphs for the program’s threads. We instrument the program by surrounding every discovered CF with probes. Then we run the program in the same configuration and analyze its trace. The trace contains a sequence of probe hits for each thread, where every pair of adjacent probe hits represent an execution of a CF.

The sequence of CF executions is used to generate the probabilistic call graph for a thread. An additional post-processing is required to generate a useful model. First, the presence of CFs from the program’s libraries may result in incorrect behavior of the model. Thus we split library CFs into multiple nodes in the call graph, depending on their calling context. Second, simplistic generation of the call graph results in overly complex model, which has low performance and is hard to understand. To generate compact model we remove CFs with negligent impact on the performance.

CF parameters are also obtained from the trace. Parameters $\Pi_{lock}$ of $s_{in}$, $s_{out}$, and synchronization CFs are reported by instrumentation. Parameters of the locks and queues themselves (e.g. the capacity of the barrier, the maximum size of the queue) are obtained by instrumenting their constructors and initializers. Execution times $\Pi_{CPU}$ for computation CFs are obtained from timestamps. Parameters $\Pi_{lock}$ of I/O CFs are obtained from the BTrace log of low-level I/O operations performed by the program.

Finally, we generate the model by translating collected information into the format acceptable by OMNET.

4. RESULTS AND CONTRIBUTION

We used our approach to model two industrial applications: the Sunflow 3D renderer and the Apache Tomcat servlet container. For each of them we built the model from one configuration of the program and used it to predict performance in a set of other configurations. Then we measured actual performance in same configurations. The relative error $\varepsilon$ between measured and predicted performance was a metric of prediction accuracy. Our tests were conducted on a quad-core PC equipped with a single hard drive.

Sunflow is a medium-size (21987 LOC) program that uses ray tracing algorithm for image synthesis. Its performance measured as the time required to render the image. We experimented with two configuration parameters: the number of working threads used for computations (1 to 8) and the number of active CPU cores (1 to 4). The relative prediction error varies in $\varepsilon \in (0.003, 0.085)$ with the average error across all the configurations $\overline{\varepsilon} = 0.032$ (see Figure 1 top).

Tomcat is a large (182810 LOC) industrial web server and a servlet container. Its configuration includes the number of working threads to serve HTTP requests (1 to 10) and the rate of incoming HTTP requests. Its performance metrics are response time $R$ and throughput $T$.

We modeled Tomcat in two setups. In the first setup Tomcat was serving requests for static web pages. The error for response time $\varepsilon(R) \in (0.003, 0.932)$ with average $\overline{\varepsilon(R)} = 0.232$ (see Figure 1 bottom). The relatively high error is explained by variations in the page cache hit rate. The error for throughput $\varepsilon(T) \in (0.001, 0.807)$ and $\overline{\varepsilon(T)} = 0.016$.

In the second setup Tomcat was hosting a servlet, which used iText library to create PDF documents. The error for $R \varepsilon(R) \in (0.000, 0.375)$ with the average $\overline{\varepsilon(R)} = 0.143$. The error for $T \varepsilon(T) \in (0.000, 0.213)$ and $\overline{\varepsilon(T)} = 0.053$.

Our work shows it is possible to automatically build accurate models of complex multithreaded programs. We make important contributions to the state of the art. First, we present an end-to-end approach for accurate performance modeling of multithreaded programs. Second, we present a framework for construction of such models. Finally, we verify our approach by predicting performance of real-world industrial applications. The main limitation of our approach is an assumption that every lock in the program can be assigned to one of the pre-defined lock types (barrier, mutex, semaphore etc).

![Figure 1: Experimental results for Sunflow (top) and Tomcat (bottom)](image-url)
REFERENCES


