References
1. Team Gwawr Web Site (2012), http://solarcarwales.co.uk/gwawrhomeeng.htm, (Accessed 2 February 2012)
2. Friis H.T.,(1946) Proc. IRE, vol. 34, p.254. 1946
3. Okamura, Y., et al., (1968), "Field Strength and its Variability in VHF and UHF Land-Mobile Radio Service", Rev. Elec. Comm. Lab. No.9-10pp. 825 - 873, 1968.
4. Hata, M. (1980), "Empirical Formula for Propagation Loss in Land Mobile Radio Services", IEEE Trans. Vehicular Technology, VT-29, pp. 317 - 325, 1980.
5. Mathwork Web Site(2012), http://www.mathworks.com/matlabcentral/fileexchange/25941-okumura-hata-mode, (Accessed 21 February 2012)

UDC 004.94

SIMULATING PERFORMANCE OF MULTITHREADED PROGRAMS

A. Tarvo
Brown University, Providence, RI, USA

Introduction

Performance is an important characteristic of any software system. It depends on various factors, such as parameters of underlying hardware, system's workload, and configuration options of the program. Proper understanding of how these factors affect performance of the system is essential for many tasks, including configuration management, building autonomous data centers, and answering "what-if" questions. Usually this requires building a model of the program that will predict performance of the program in different configurations.

Building performance models of computer programs is hard, but building models of multithreaded programs is even more challenging. Such models require simulating the complex locking behavior of the application and concurrent usage of various computational resources such as the CPU and the disk I/O subsystem. As a result, existing performance models either impose restrictions on the types of programs that can be modeled [1] or require collecting vast amounts of data about the performance of the system in different configurations [2]. Such limitations often make these models impractical.

Our work attempts to overcome these limitations. We have developed a PERSIK (PERformance SImulation Kit) - a simulation framework for modeling performance of multithreaded programs running in various configurations under the established workload [3].

Model definition

For the purpose of simulation we represent computations performed by the program as request processing. We denote a request as some external entity; the

program responds to the request by performing certain operations. The overall time required to process the request (response time R) and the number of requests served in the unit of time (throughput T) are the most important performance metrics of such request-processing system. This representation allows simulating a variety of programs, such as server applications, reactive systems, and scientific applications.

We employ a two-tier approach towards building the model. At the high level we simulate the program using a variation of a queuing network model. At the low level we simulate threads as probabilistic automata. Both types of model are built according to a discrete-event principle.

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 where queues correspond to program's buffers, and server nodes represent program's threads. However, it is more flexible than classical queuing networks as it does not restrict structure of the model, the number of service nodes n, or parameters of the arrival process λ.

Lower-level thread models simulate delays that occur in threads when they process requests. Thread models are implemented as probabilistic call graphs $<S, \delta>$, whose vertices S correspond to the pieces of the program's code – *code fragments*. In particular, we distinguish computation, I/O, and synchronization code fragments (CF) in the program. The edges of the graph represent a possible transition of control flow between the code fragments; probabilities of these transitions are defined with a mapping $\delta: S \rightarrow P(S)$.

Execution of each code fragment results in the delay τ. However, whereas the structure of the call graph $<S, \delta>$ remains invariant, the amount of delay τ depends on the degree of the parallelism in the program. For example, consider multiple threads that perform equal amount of 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. This phenomenon is known as a "resource contention". Resource contention can happen during parallel I/O operations as well.

To correctly simulate time delays that occur due to resource contention, we describe each code fragment with parameters Π and then use Π to compute τ. In particular, parameters Π_{io} of an I/O CF are the number and properties of low-level I/O operations, while parameters Π_{cpu} of a computation CF is the amount of CPU time τ_{cpu} for it (the time necessary to execute the CF if it was running on the CPU uninterrupted).

To compute τ from Π we use dedicate models that simulate underlying OS and hardware. These models track the state of the simulation Q(t) at each moment of time t. They use Q(t) along with parameters Π of the code fragment to compute the actual delay τ for that CF. In our work we have developed two such models: one is the model of CPU/thread scheduler, while another is the model of the disk subsystem.

Model building

Building the model of the multithreaded program involves three stages: data collection, model building, and model verification.

During the data collection we first manually inspect the program's source, and identify threads and code fragments in it. Then we instrument the program by inserting probes at the borders of individual CFs. Rest of the data is collected automatically.

We run a program in a one representative configuration. When the probe is hit during the program's execution, instrumentation records τ_{cpu} and delay time τ for each CF in the log file. Once the run is finished, we analyze that log and extract parameters Π_{cpu} of a computation CFs, transition probabilities δ for each thread, and performance metrics R and T of the program.

To extract performance metrics for disk I/O CFs, we instrument the OS kernel. In particular, we instrument system call routines that can initiate I/O requests (sys_read(), sys_stat() and others), the generic_make_request() function that insert I/O requests into the I/O scheduler queue, the blk_start_request() function that passes the request to a hard drive, and I/O completion routines. As a result of this kernel instrumentation we retrieve parameters Π_{io} of I/O CFs.

Once the data is collected, we build a performance model of the program using the PERSIK modeling toolset. PERSIK is an extension to the OMNET– a discrete event simulation framework [4]. PERSIK model consists of interconnected *blocks* communicating using *messages*. Internally, blocks and messages are implemented as C++ classes.

PERSIK models are built according to a formal definition stated earlier. A high-level PERSIK model contains blocks that represent program's queues, request sources, and threads. Message flow between those blocks simulates request flow through the program. Thread blocks are the models on their own: they consist of computation blocks, I/O blocks, blocks that simulate calls to locks, and blocks that read/write data from the high-level model. Messages in the thread model simulate computation flow in the thread.

The high-level model also contains OS/hardware models: the model of the CPU/thread scheduler and the model of the I/O subsystem. The CPU/thread scheduler model is a simple queuing model that simulates a round-robin thread scheduler and CPU with a given number of cores. The model of a disk I/O subsystem is more complex: it includes a queuing model of a disk I/O scheduler and a statistical model of a hard drive. The hard drive model simulates the processing time of the I/O request by the hard drive τ_{disk} as a distribution $P(\tau_{disk}|x_{dio})$, where x_{dio} is the type of the I/O request (synchronous read, metadata read, read-ahead), which implicitly represents the locality of the I/O operation.

To verify our approach we have built a model of the MolDyn program. MolDyn is a multithreaded scientific application that iteratively computes interaction of 8192 argon atoms in a cubic volume. It is a part of the Grande multithreaded benchmark [5].

The length of iteration is the most important performance metric of the MolDyn. In terms of our formal model it corresponds to the response time R. The length of iteration highly depends on the number of MolDyn working threads. Too few working threads will result in inferior performance; too many threads will result in resource contention. To predict how the performance of the MolDyn depends on the number of threads we have built the model of that program using PERSIK framework.

The figure 1 compares the predicted performance of the Moldyn program versus the observed one, measured on a quad-core machine. As one can see, the PERSIK predicts the iteration length with the adequate accuracy. The relative error ranges within (0.003; 0.220) across different configurations; the average error across all the configurations is 0.105. The spikes in the actual performance of the MolDyn are caused by the "false sharing" – a conflict that occurs when multiple CPU cores attempt to simultaneously update adjacent records in the cache [6]. More accurate simulation of CPU cache remains a subject of the future work.

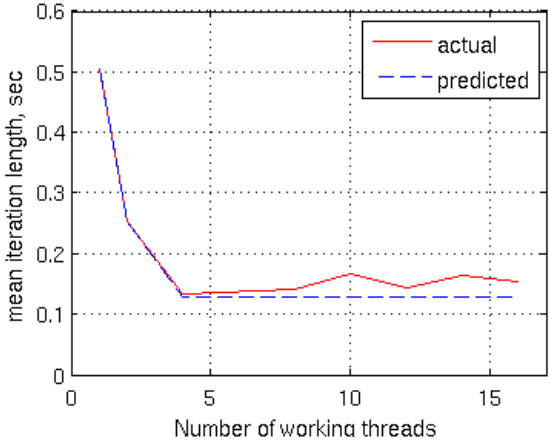

Figure 1: Experimental results for the MolDyn multithreaded program

Conclusion

In this paper we presented a brief outline of our methodology and the toolset for modeling performance of the multithreaded computer programs. Please see [7] for a more complete description of our work.

We have verified our approach by building the model of the multithreaded scientific computing program. The relative error of our model is 0.105, which is comparable to the accuracy of statistical models. However, our models require less data and allow for modeling a wider range of computer programs.

The stable version of our framework along with examples of models is available for download [3]. We plan to further develop our approach by improving the accuracy and automating building performance models.

References
1. A. Ganapathi et. al., Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning//ICDE'09, P. 592–603
2. G. R. Nudd et. al., Pace – a Toolset for the Performance Prediction of Parallel and Distributed Systems//Int. Journal of High Performance Computations Applications, vol. 14, 2000. – P. 228-251.
3. http://cs.brown.edu/~alexta/PERSIK.html
4. http://www.omnetpp.org/
5. J.M. Bull et al., A Benchmark Suite for High Performance Java//Java Grande Conference'99. – P. 81-88
6. W. Bolosky, M. Scott, False Sharing and its Effect on Shared Memory Performance//SEDMS'93. – P. 57-71
7. A. Tarvo, S.P. Reiss, Using Computer Simulation to Predict the Performance of Multithreaded Programs//ICPE'12. – P. 217-228

УДК 004.942

МЕТОДИКА ОЦЕНКИ ФУНКЦИОНИРОВАНИЯ БЕСПРОВОДНЫХ ОДНОРАНГОВЫХ СЕТЕЙ С ЯЧЕИСТОЙ ТОПОЛОГИЕЙ

И.А. Адуцкевич
Белорусский государственный университет, Беларусь

Беспроводные одноранговые сети с ячеистой топологией используются в ситуации, когда необходимо организовать сеть между вычислительными устройствами в условиях отсутствия либо нежелательности использования инфраструктуры, обеспечивающей сетевое взаимодействие. В этом случае мобильные устройства могут создать временную сеть для обеспечения связи в данный момент времени, другими словами – организовать сеть «на лету». Каждый узел такой сети способен генерировать данные адресованные любому другому узлу в сети. Все узлы сети при необходимости обеспечивают возможность ретрансляции данных до конечного получателя. В общем случае эта сеть может быть подключена к другим сетям передачи данных через один или несколько узлов, выполняющих функцию шлюза. Поддержка многоскачковой передачи данных в мобильных беспроводных одноранговых сетях является ключевым отличием данного типа сетей от других беспроводных телекоммуникационных систем.

Вне зависимости от способов радиопередачи и модели передвижения узлов топология беспроводной одноранговой сети в любой фиксированный момент времени может быть представлена в виде графа [1]. Для описания беспроводных одноранговых сетей с помощью графов узлы сети сопоставляются с вершинами графа, а соединения между узлами соответствуют ребрам графа. В качестве упрощения модели будем полагать, что