

Resource Sharing in QPN-based Performance Models

V. Babka

Charles University Prague, Faculty of Mathematics and Physics, Prague, Czech Republic.

Abstract. Performance models of enterprise software systems allow predicting performance of the system in early development phases. The durations of atomic actions needed to solve the model can be significantly influenced by resource sharing, capturing this influence in the model is however difficult and often omitted. This paper bases upon our previous solution that uses separate resource and performance models and proposes a method of integrating these models at the tool level in the SimQPN Queuing Petri net simulator. The benefits include significantly shorter duration of the analysis and the possibility to create more accurate resource models.

Introduction

For enterprise software systems, performance of the final system is as important as fulfilling functional requirements. The system must be able to cope with certain amount of client request throughput and achieve sufficient response time in order to be of practical use. Performance engineering, which provides development techniques for meeting performance requirements in the final system, is therefore an important part of software development process.

The problem with performance is that it may be strongly influenced by early design decisions, and the cost of resolving a performance issue is higher when it is discovered later – in the worst case when the system is fully implemented and deployed. It is therefore desirable to be able to predict performance of a system in early development phases, such as design, and thus give the option to choose an architecture alternative that yields the most promising results before the actual software implementation process (i.e. coding) begins.

A common method for performance prediction is to create and analyze a performance model of the system. A performance model is ideally created from the software model (specified e.g. in UML) and describes, with certain degree of abstraction, interactions of atomic actions inside the system, such as method calls between components the system is composed of. Various formalisms for performance models exist, including Queuing Networks, Petri Nets or Stochastic Process Algebras with many variations and combinations [7]. To solve the performance model, the durations of the atomic actions (e.g. database queries) are determined, usually by benchmarking. Solving the model yields results such as estimated throughput and response time.

In our research, we are concerned with scalability analysis of distributed component systems. By repetitively solving the performance model with gradually increasing workload (i.e. number of clients), we can roughly estimate the scalability limits of the system being modeled. Our focus is to model resource sharing, whose effects depend on the workload intensity and which can significantly influence durations of atomic actions used to solve the performance model, yet is mostly neglected in related work.

The structure of this paper is as follows: First we present the problem of resource sharing in more detail, based on the results of our ongoing experiments. We proceed by outlining our group's previous work on performance modeling with resource sharing. Then we present the current work in progress that extends our approach by using richer QPN-based performance models and incorporates resource model in the SimQPN simulator, together with first results. Discussion of future work concludes the paper.

Resource sharing

In the following, a resource is any physical (hardware) or logical (software) entity that code needs for its execution. A typical example of a hardware resource is a processor or system memory, software resources are objects provided by the underlying operating system or middleware, such as mutexes, files or network sockets. When there is only one process running in the system, it can use all these resources exclusively and run with optimal performance. However, multiple concurrent processes compete for the resources, and have to e.g. wait for a mutex to be unlocked or take turns on a shared processor, which naturally affects their performance.

A more complex example of a shared resource are processor memory caches, which we previously studied in [6] and [5]. A memory subsystem in a contemporary processor is an important resource which improves performance of otherwise relatively slow memory accesses by employing several levels of buffers and caches and a prefetching mechanism that adapts to memory patterns of a code being executed. Thus, when multiple

unrelated memory intensive operations share a processor, scheduling of one operation can evict the data of other operations from the caches and reconfigure the prefetching mechanism.

We have conducted a number of benchmarking experiments [5,6] to quantify the influence of cache sharing. To demonstrate its significance, we present one of them here. The experiment measures the duration of a Fast Fourier transform (FFT for short) of a fixed size memory buffer (initialized with a fixed input data), which is an example of a memory intensive operation. To induce cache sharing, we execute an interfering operation between the buffer initialization and the actual transformation. This operation reads data at random addresses aligned at cache line size in a pre-allocated memory range different from the buffer used for FFT. This evicts parts of the FFT buffer from the caches. By varying the number of cache lines being accessed, we can observe the effect of cache eviction on the FFT transformation duration¹, as depicted in Figure 1.

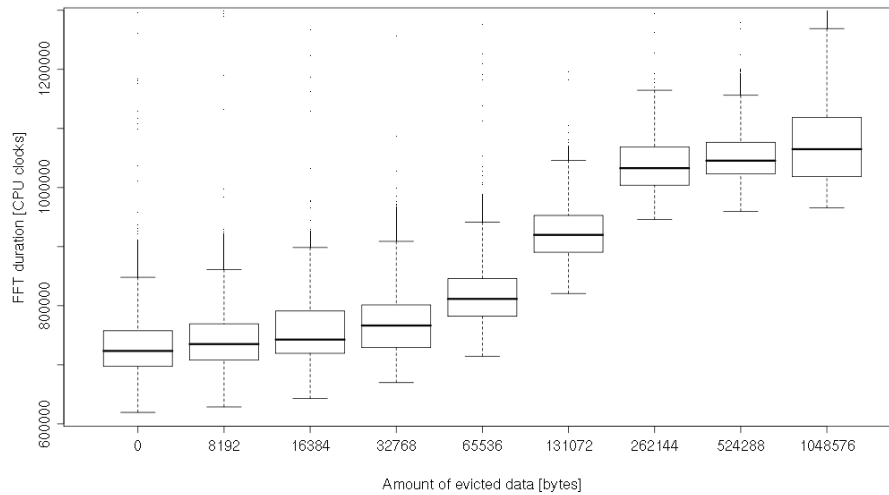


Figure 1. Data cache sharing effects on FFT duration, one 128 KB buffer

Results of this experiment show that slowdown of code execution due to cache sharing can be quite notable. Interestingly, in some scenarios we even observed a slight speedup instead of slowdown of the FFT itself. If we use a different hardware² and a FFT variant which uses separate buffers for input and output, with some buffer sizes we can see that while evicting a small part of the cache increases the transformation duration, more eviction surprisingly improves the apparent FFT performance, as Figure 2 shows.

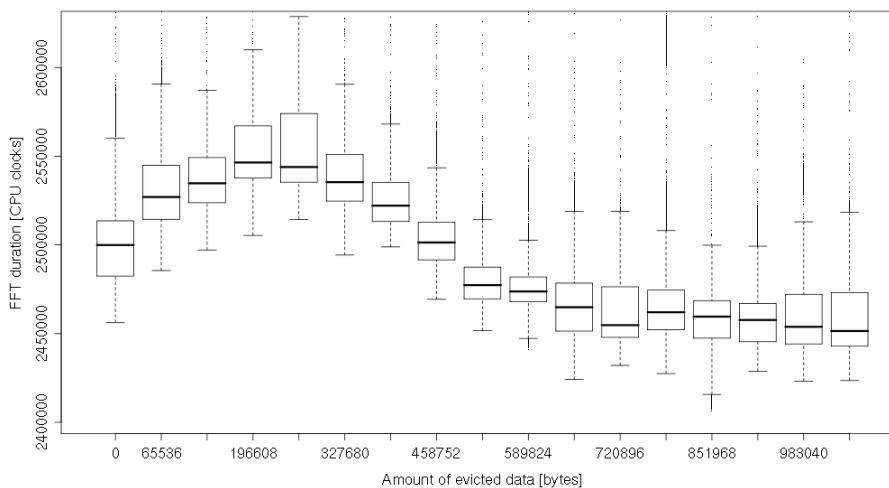


Figure 2. Unusual data cache sharing effects on FFT duration, two 320 KB buffers

¹ Intel Pentium 4 Northwood 2.2 GHz, 8 KB data L1, 12 KB code L1, 512 KB unified L2, Fedora Core 6. FFTW 3.1.1 *fftw_plan_dft_1d* [6]

² AMD Athlon 64 3000+ Venice DH7-CG 1.8 GHz, 64 KB data L1, 64 KB code L1, 512 KB unified L2.

The somewhat counterintuitive results of the experiment are related to the need to write back the modified cache lines that contain the results calculated by FFT. If the FFT transformation is performed repeatedly without interference, after each run the cache lines holding data from the output buffer are marked as dirty, because the results were written into them. Reading of the input buffer in the subsequent run replaces these cache lines, which have to be written back to the main memory so that the modified data is not lost. This puts more pressure on the memory subsystem, and slows down the execution. When we interleave the FFT transformation with the cache eviction code, the dirty cache lines are written back during the eviction, and replaced by cache lines that are not marked dirty (the eviction code only reads memory). The subsequent FFT transformation is therefore not slowed down by the writeback, data in the cache lines populated by the eviction code can be just forgotten. Note that this experiment is not at all artificial, interleaving FFT with a less memory intensive processing (which is likely to happen in practice during further processing of the FFT results) would result in the same apparent shortening of the FFT duration.

We have shown that sharing of resources such as processor caches by concurrent operations can have both significant and unexpected effects on performance. Thus, performance models may give inaccurate results when the durations of atomic actions are measured in isolation or with a fixed concurrency. This is a problem – the degree of concurrency often cannot be known in advance, but is rather one of the outputs of performance prediction. Resource sharing should therefore be part of the performance model itself.

Common performance models are naturally able to model some types of resource sharing – models based on queuing represent shared processors (and other resources with similar behavior) as queues, where concurrent operations wait to be served. Petri nets can easily model exclusive resources such as mutexes or thread pools. Sharing of resources such as the processor caches is, however, mostly omitted due to modeling complexity. Although several cache models exist [2,11], they do not cover all the features such as multiple levels of hierarchy, or are based on different formalisms than performance models, which makes them hard to integrate.

Past work

In order to incorporate the effects of resource sharing in performance modeling, in our previous work we have proposed a method that considers separate performance and resource models and is described in detail in [4]. This method can support virtually any performance model composed of interacting atomic actions with fixed average durations, provided that the output of the model's solver can be used in the resource model.

For each considered shared resource, we need a resource model to approximate the resource usage, which combines two related factors. Mode of resource usage describes how the resource is used (for example memory access patterns) and can be either formalized or determined by benchmark experiment resembling the modeled scenario. Degree of resource usage describes quantitative factors which are either known in advance (such as cache sizes) or depend on the performance of the modeled system (e.g. number of concurrently processed requests which can grow if the system cannot process them fast enough) and can be extracted from the output of the performance model.

Because of the latter, we have a situation where the output of the performance model (degree of resource usage) serves as an input of the resource model and the output of the resource model (durations of atomic actions) is an input of the performance model. We have solved this circular dependency by starting at minimal degree of resource usage and iterating the two models until the results stabilized, using a simple ε stability criterion.

To validate the method, we used the CoCoME [8] enterprise trading system as the case study. We selected the customer checkout use case for performance prediction and also included the workload of two other use cases (product orders and enterprise reports), all using a single enterprise server with database to store data such as product quantities and barcode numbers. Parameters for our scalability analysis are number of stores (which also determines number of product items according to the CoCoME specification) and number of cash desks per store.

Our performance model was created by hand from the CoCoME behavior and deployment description in the SOFA component framework [8], which provided us with information on both interaction and placement of the individual components. To simplify the model, we omitted some activities that cannot affect the throughput of the system significantly. The model was created in the LQN [16] formalism, with average queue length being the part of performance model output that serves as the resource model input.

To create the resource model, we analyzed the reference implementation of CoCoME, which is a distributed application written in Java that uses ActiveMQ [1] for messaging, Hibernate [10] object persistence layer and the Derby [3] database. The heavily shared resources we identified were (1) the cache in the Derby database and (2) system memory of the enterprise server. By benchmarking, we determined durations of database queries for all query types with two variants – cached by the Derby cache and fetched from disk. We also measured additive memory swapping overhead and memory consumed by all system components as well as each concurrent request.

We then designed a resource model which calculates probabilities of database query variants and probability of swapping. Input of this model is partially static (database cache size, system memory size, number of product items, memory consumption of components) and partially depends on the performance model output (memory occupied by concurrent requests). Output of this model is directly used as parameters of the LQN performance model.

We evaluated the models by comparing the predicted results with results from benchmarks of the reference implementation of the whole CoCoME system. Our goal to predict the scalability limits was met, although the prediction was a bit too pessimistic, which can be explained by the difficulty to measure precisely memory requirements of the individual components in the garbage collected Java environment.

QPN-based Performance Models

While the LQN formalism proved quite sufficient for the CoCoME performance model and the results were satisfactory, we have considered also different formalisms. For better accuracy, the performance model should provide means to express usage of exclusive software resources such as thread pools or locks that are common in software systems but modeling them with LQN usually leads to less accurate and detailed models [12]. We should also be able to integrate the performance and resource models at the tool level. Both the ability to accurately model commonly used elements of software systems and sufficient tool support is crucial for any practical application of the approach.

Queuing Petri nets (QPN) are a good candidate to fulfill our demand for richer performance models, since they combine the modeling power of both queuing networks and (colored) Petri nets [13]. The fact that the SimQPN Petri net solver is available to us (details in next section) makes also the tool integration viable. In short, QPNs consist of places and transitions connected to form a bipartite directed graph – places connected to a transition are called input places of the transition, places that the transition is connected to are called output places. The places contain non-negative number of tokens of colors from a defined set of colors, with defined initial arrangement (called marking). Each transition has a set of modes, in which it may fire – consume tokens in input places and create tokens in output places. The modes define how many tokens (non-negative integer) of each color in each input place are needed for the mode to become enabled (and that are destroyed when the mode fires) and analogically what tokens are created after the mode fires. When more modes of a transition or multiple transitions are enabled, a transition to fire first is chosen randomly according to weights assigned to the modes.

There are two types of places in QPNs – ordinary and queuing. In an ordinary place, incoming tokens become available immediately to all transitions for which this place serves as an input place. A queuing place is divided to a service station with queue, where tokens wait for available server and then are served depending on their color (typically using an exponential service time distribution parameterized by mean), and a depository which collects served tokens and makes them available to transitions.

The methodology for creating performance models in QPN is described for example in [13]. Very basically, a QPN network models a distributed system, where tokens model arriving client requests as well as the calls between components of the system, and queuing places represent hardware resources such as processors or disks. A QPN solver calculates mean token throughputs, populations and residence time in each place in the system's steady-state. Because token populations represent degree of concurrency similarly to the queue lengths in LQNs, QPNs should be suitable formalism for our resource sharing modeling approach.

To validate the applicability of QPNs for our approach, we have (manually) converted the performance model of CoCoME from LQN to QPN. We also adapted our scripts that parse the model solver output to feed the resource model, and that modify the input of performance model with values from the resource model. We then performed a scalability analysis both with LQN and QPN and compared the results. As Figure 3 shows, there is some difference in absolute numbers that could be attributed to the absence of exact 1:1 mapping between the model variants. More importantly, the prediction of the system scalability limit is preserved.

Integrating Resource Models

Although the results of the QPN-based performance model are comparable with the LQN-based one, there is a great difference in duration of our scalability analysis, which takes less than a minute with LQN but several hours with QPN. This is due to the nature of the solvers we used for the two formalisms – the LQNS solver [15] is analytical and therefore fast, QPNs however due to their greater expressiveness suffer much more from the state explosion problem and therefore cannot be solved analytically except for simple models [13]. The SimQPN solver [14] is therefore based on discrete-event simulation and statistical collection of results, which is much more computationally expensive, and is a price for the greater modeling power.

The analysis is further prolonged due to our iterative approach – the model instance has to be solved several times with different parameters instead of once. We will now present a work in progress that mitigates this impact of resource modeling by exploiting the use of a simulation-based solver. This approach is possible thanks

to an ongoing collaboration with Samuel Kounev, one of the SimQPN authors.

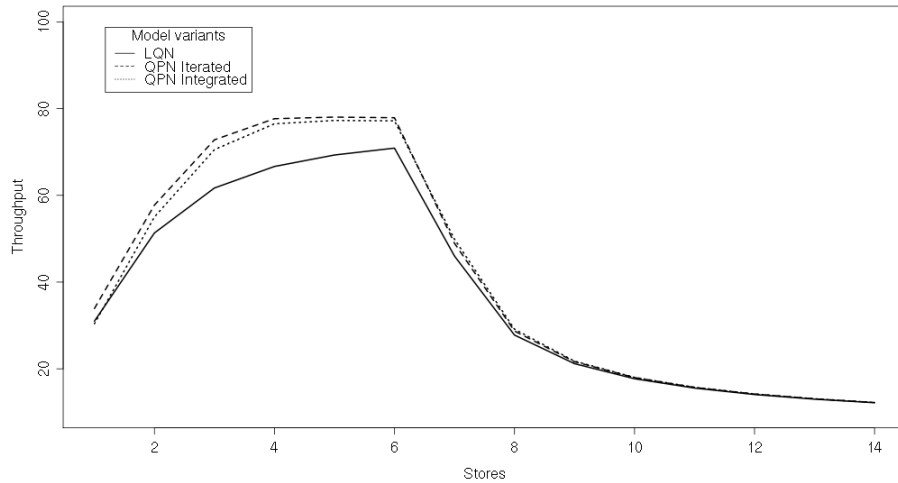


Figure 3. Throughput prediction with different model variants (8 cash desks per store)

The main idea of this approach is to integrate resource model calculations into the model simulation by the SimQPN tool, instead of iterating complete simulation runs. For the model of our CoCoME case study, this means that the memory and cache resource model would continuously observe current token populations and adjust the parameters of the QPN on-the-fly. This is feasible to implement in SimQPN, which is a Java-based simulator tailored specifically to QPN simulation (instead of a general purpose simulator) and all parts of a QPN network are available as Java objects with readable and changeable attributes. A resource model can be therefore implemented as a Java class and integrated into the simulator, after taking care of several technical details.

One of the decisions to make is how often the resource model should be invoked to recalculate the model parameters based on current token populations. The most accurate variant would perform this operation on each population change – in SimQPN this corresponds with each event processing, which would however impose a significant performance overhead. A feasible alternative is to split the simulation time into intervals, during which average token populations are collected and used for resource model recalculation at the end of each interval. The obvious question is how to find an optimal interval length – shorter intervals mean better accuracy but greater overhead and vice versa. On the other hand, too long intervals can cause the updates to reach a steady state unnecessarily slowly – our previous iterative approach can actually be seen as an extreme variant of this, with intervals as long as the whole simulation. Currently we use a fixed interval with length set by a user, but this is an obvious opportunity for further optimizations.

Our iterated approach assumes that the input (and thus also the output) of the resource model will eventually converge to a steady state. In the integrated approach we therefore also assume that resource model recalculations will eventually stabilize and thus we need to determine convergence of their input values (i.e. token populations). For now we use a simple method that after each interval compares the current values with values from previous recalculation. If the difference does not exceed a configurable relative threshold, the resource model recalculation is not performed. After a number of successive intervals pass with no recalculation, the values are considered stable. The resource model is not called anymore, standard statistics collection of SimQPN is started and the duration of the rest of the simulation is controlled by the usual termination criteria of SimQPN [14].

We have applied the integrated resource model approach in the CoCoME case study and compared it with the iterated approach. In terms of performance prediction results, the differences between the iterated and integrated resource models are negligible, as Figure 3 depicts. There is however a significant improvement in the duration³ of the analysis. Table 1 presents the durations for different variants of the performance model and different termination criteria of SimQPN.

³ Intel Xeon E5345 2.33 GHz Quad-Core (note that the analysis is not optimized for multi-core execution), 8 GB RAM, Gentoo Linux, Sun JDK 1.6.0_05. The analysis covered 1-10 stores and 1-8 cash desks per store, with resource model recalculation interval of 100000s, 5% relative threshold and 5 passes for determining convergence. SimQPN stopping criteria were either fixed length (1000000s) or 5% relative precision. The simple model is a subset of the full model, which models customer checkout only, omitting workload by other use cases.

Table 1. Analysis durations of iterated and integrated resource model approaches.

Model, stop criterion	Iterated duration (h)	Integrated duration (h)
Simple, fixed	6.5	2.5
Simple, relPrec	11	4.5
Full, fixed	14.5	4
Full, relPrec	27	10

Conclusion and Future Work

We have proposed an approach for integrating QPN-based performance models with resource models in the SimQPN tool. Durations of analyses are significantly shorter compared to our previous approach that iterates the two models. Since this is a work in progress, several aspects of the approach could potentially be optimized to further improve its performance. Using QPNs for the performance model also allows creating richer and more accurate performance models with respect to software contention, although our current case study is quite simple and thus does not take advantage of these benefits. We plan to either extend the case study or switch to a more complex one for our future research.

Our future work should focus on creating more models of commonly shared resources and integrating them into the SimQPN tool. The models should be general with several parameters in order to be reusable – values for the parameters would be obtained by benchmarking. The integrated approach gives us opportunity to create resource models with more complex input than just number of concurrent requests – we can observe interactions of atomic actions in detail which could be useful e.g. in a processor cache model. For practical usability of the approach, we plan to create tools for automatic or semi-automatic performance model construction from the system description in the SOFA component model.

For a discussion of related work we refer the kind reader to our paper [4] due to space constraints.

Acknowledgments. I would like to thank Samuel Kounev, whose help was essential for resource model integration in SimQPN, and my advisor Petr Tůma for his valuable advice. This work was partially supported by the Czech Science Foundation under the contract no. 201/05/H014.

References

1. ActiveMQ, <http://activemq.apache.org>
2. Agarwal A., Hennessy J., Horowitz M.: An Analytical Cache Model, TOCS 7(2), ACM, 1989.
3. Apache Derby, <http://db.apache.org/derby>
4. Babka, V., Decky, M., Tuma, P.: Resource Sharing in Performance Models. In: EPEW'07, Springer, 2007.
5. Babka, V., Tuma, P.: Effects of Memory Sharing on Contemporary Processor Architectures. In: MEMICS'07, Znojmo, Czech Republic, 2007.
6. Babka, V.: Influence of Resource Sharing on Performance, Master Thesis, Charles University, 2007.
7. Balsamo, S., DiMarco, A., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development. In: TSE, IEEE Computer Society Press, Los Alamitos, 2004.
8. Bures, T., Decky, M., Hnetyňka, P., Kofron, J., Parizek, P., Plasil, F., Poch, T., Sery, O., Tuma, P.: CoCoME in SOFA, Chapter in The Common Component Modeling Example: Comparing Software Component Models, Springer, 2008.
9. Frigo M., Johnson S.G.: FFTW, <http://www.fftw.org>
10. Hibernate, <http://www.hibernate.org>
11. Hossain A., Pease D. J.: An Analytical Model for Trace Cache Instruction Fetch Performance, ICCD'01, IEEE, 2001.
12. Kounev, S.: Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction, Ph.D. Thesis, Technische Universität Darmstadt, Germany, May 2005.
13. Kounev, S., Buchmann, A.: On the Use of Queueing Petri Nets for Modeling and Performance Analysis of Distributed Systems, Chapter in Vedran Kordic (ed.) Petri Net, Theory and Application. Advanced Robotic Systems International, Vienna, Austria, 2007.
14. Kounev, S., Buchmann, A.: SimQPN: A Tool and Methodology for Analyzing Queueing Petri net Models by Means of Simulation. In: Performance Evaluation, Vol. 63, Issues 4-5, Elsevier, 2006.
15. LQNS - Layered Queueing Network Solver, <http://www.sce.carleton.ca/roads/lqns/>
16. Xu J., Oufimtsev A., Woodside C. M., Murphy L.: Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queueing Network Templates, SIGSOFT SEN 31(2), ACM, 2006.