

Resource Sharing in Performance Models

Vlastimil Babka, Martin Děcký, and Petr Tůma

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 118 00, Czech Republic
{vlastimil.babka|martin.decky|petr.tuma}@dsrg.mff.cuni.cz

Abstract. In software systems, individual components interact not only through explicit function invocations, but also through implicit resource sharing. The use of shared resources significantly influences the duration of the invoked functions. For resources that are heavily shared, capturing this influence can lead to performance models that have a large number of elements and a large number of dependencies. We introduce an approach that can model resource sharing separately from function invocations, keeping the performance model reasonably simple while still describing many of the effects of resource sharing on the duration of function invocations. The approach has been tested on the CoCoME component application modeling example.

Keywords. enterprise systems, performance modeling, resource sharing

1 Introduction

Our work focuses on performance models of software systems that describe the interaction of individual software components in terms of atomic actions, and that derive the duration of the atomic actions from the implementation of the components [15,16,17,18,26,27,29]. The duration of the atomic actions, needed to solve the performance model, is typically chosen to reflect the duration of function invocations on individual software components.

The duration of function invocations is typically determined by benchmarking. This, however, cannot be done by benchmarking the entire software system – if the entire software system were readily available and easily benchmarked, performance modeling would not be of much use. The individual components of the software system are thus benchmarked mostly in isolation and the average duration of function invocations determined by benchmarking applies to this isolated execution.

Unfortunately, the duration of a function invocation typically depends on the resources the function uses. When such resources are shared within a software system, the duration of the function invocation is likely to differ from the duration of the function invocation observed during the relatively isolated execution of benchmarks. Unless resource sharing is described in the performance model, this naturally impacts the performance model precision.

Resource sharing is very common and in some cases, such as sharing of processor caches, physical memory, or disk caches, concerns many components. Describing these resources in a performance model increases the complexity of the model both in the number of elements and in the number of dependencies between elements. This is probably why even recent work on performance modeling of software systems often tends to omit some heavily shared resources [16,17,18,26,27,29] or points out the high cost of solving the performance model when heavily shared resources are present [15]. Both intuition and evidence, however, suggest that these resources need to be modeled [13,14].

To remedy the difficulties associated with describing heavily shared resources in performance modeling of software systems, we present an approach where resource sharing can be modeled separately from function invocations. Our combined model consists of the *resource model*, which describes how resources are used, and the *performance model*, which describes how functions are invoked.

The resource model and the performance model complement each other. To solve the resource model, knowledge of the interactions between the individual components and of the degree of parallelism inside the individual components is needed – and is provided by the performance model. Similarly, to solve the performance model, knowledge of the duration of the atomic actions is needed – and is provided by the resource model. The two models are solved iteratively.

We test our approach by modeling the performance of the architecture provided by CoCoME [5]. This architecture describes an enterprise information system responsible for tracking the stocks and sales of products in multiple stores. Its prototype implementation relies on contemporary middleware technologies such as ActiveMQ [1], Apache Derby [2] and Hibernate [11], and can be considered a reasonably realistic example of a software system for the purposes of performance modeling.

To describe our approach in detail, we first illustrate on examples of two heavily shared resources why we believe such resource sharing to be difficult to describe in performance models. We proceed by detailing the concept of separating the resource model and the performance model and explaining the complementary character of the two models. The test of our approach follows, with due evaluation and conclusion.

2 Resource Sharing

In the context of performance modeling, our definition of a resource includes any shared entity that individual components rely on in their execution. Typical resources are physical memory, processor with its various caches, disk with its caches and queues, and even entire file systems and network connections. The duration of function invocations, represented by the duration of atomic actions in performance models, naturally depends on the use of resources. The exact nature of this dependency is affected by the resource kind and by the way it is used. We select two common examples for a more detailed look.

2.1 Sharing Processor Cache

Processor cache is a resource that is shared inherently by any code running on the same processor. To illustrate how sharing of processor cache can change the duration of method invocation, we use two simple experiments with a Fast Fourier Transform library, FFT for short.

When processing data in a memory buffer, FFT will naturally perform faster if the memory buffer is cached. In our first experiment, we model a situation where FFT is one in a pipe of functions competing for the data cache. In detail, we fill the memory buffer with input data, simulate the competing functions by reading a set number of random addresses outside the buffer, and finally perform the FFT.¹ Figure 1 shows the dependence of the FFT duration on the number of the random addresses. As expected, increasing the portion of the buffer evicted from the cache by competing functions decreases the FFT performance.

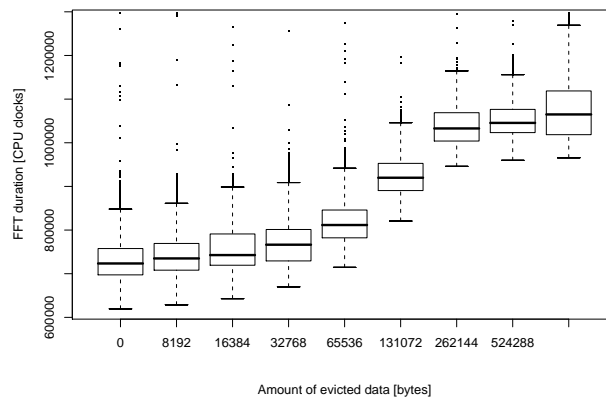


Fig. 1. Data cache sharing in FFT, one 128 KB buffer.

As our second experiment, we show that the results can also be surprisingly different from expectations. The experiment is similar to the first one except in that it uses separate input and output buffers and runs on a different hardware² Figure 2 shows that the negative impact of data cache sharing is limited and that even a positive impact can be observed.

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* [7].

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

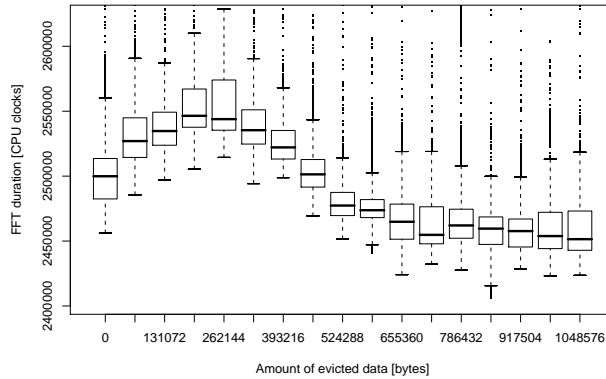


Fig. 2. Unusual effects of data cache sharing in FFT, two 320 KB buffers.

2.2 Sharing File System

File system is another frequently shared resource, or, rather, a complex of related resources whose sharing exhibits even residual effects, potentially affecting future operations. We illustrate two sharing effects via experiments.

The first experiment demonstrates how reading multiple files at a time prolongs the read duration compared to reading the files one at a time. First, we write four 256 MB files one at a time. Next, we measure the time to read the files one at a time and then multiple files at a time, interleaving reads of 32 KB, 1 MB and 16 MB blocks from each file. The results³ on Figure 3 show that the more frequent the interleaving, the bigger the performance impact.

In the second experiment, we measure the residual effect of writing multiple files at a time, which affects reading due to fragmentation. First, we write four 256 MB files, interleaving the writes as we did the reads in the first experiment. Next, we measure the time to read the files one at a time. Figure 4 shows a small but measurable performance impact.

Block size	Read slowdown
32 KB	2.60 ± 0.32
1 MB	1.33 ± 0.04
16 MB	1.03 ± 0.04

Fig. 3. Effect of interleaved reading.

Block size	Read slowdown
32 KB	1.06 ± 0.03
1 MB	1.05 ± 0.03
16 MB	1.02 ± 0.03

Fig. 4. Effect of interleaved writing.

³ Intel Pentium 4 2.2 GHz, 512 MB RAM, Hitachi 250 GB ATA, Fedora Core 5, *ext3*.

2.3 Describing Sharing

The common denominator of these two and other examples of resource sharing is that they are difficult to model precisely. When a performance model of a resource does exist, it typically focuses on a specific feature of the resource – for example, the cache models in [3,12] would not cover the processor cache example presented here simply because they did not focus on the interaction of multiple cache hierarchies, and the storage models in [6,19] would not cover the file system example presented here simply because they did not focus on interleaving and fragmentation. Furthermore, the performance models of resources can be based on a different formalism than the performance model of a software system, making their integration difficult.

Another observation is that often, the general knowledge of how a resource performs when shared is based on measurement rather than modeling. Instead of using modeling to predict performance issues, we use measurement to discover and understand performance issues and revert to modeling to confirm our understanding. The fact that even this method of arriving at understanding requires considerable amount of work [10,20] underscores our argument that heavily shared resources are inherently difficult to describe in performance models.

We believe that one approach to tackling the complexity of describing resource sharing is separating the resource model and the performance model for heavily shared resources as described next.

3 Combined Performance Model

To explain the concept of separating the resource model and the performance model, let us first consider a performance model of a software system that omits some heavily shared resources and that describes the interaction of the individual components in terms of atomic actions whose average durations are known. These average durations have been determined by measuring the durations of corresponding function invocations in a benchmark experiment.

Since function invocations use the omitted resources, we can only expect the performance model to provide a good approximation of the software system when the benchmark experiment that provided the average durations of atomic actions used a good approximation of the resource usage. In the examples from Section 2, if we were to model software that frequently evicts data from processor caches, we would have to frequently evict data from processor caches in the benchmark. Similarly, if we were to model software that uses heavily fragmented files, we would have to use heavily fragmented files in the benchmark.

The question that we need to ask is therefore how well we can approximate resource usage in the benchmark experiment. For further discussion, we describe resource usage as consisting of two related factors, *mode* of resource usage and *degree* of resource usage.

3.1 Mode of Resource Usage

The mode of resource usage describes in which way the resource is used – this concerns details such as access patterns, strategies, interleaving, etc. On our examples of the processor cache and file system resources, mode of resource usage boils down to issues including strong or weak locality of memory references, sequential or random character of file accesses, ratio of file reads to writes, etc.

These issues are not only specific to each resource, but also difficult to formalize and quantify. Fortunately, we can avoid the need for formalizing and quantifying the mode of resource usage by designing the benchmark experiment so that it resembles the scenario we are modeling. This is routinely done by many benchmarks, such as TPC [23], which resembles transaction processing applications, or RUBiS [21], which resembles an on-line bidding application.

We can conclude that as far as the mode of resource usage goes, benchmarks that provide the average durations to the performance model can be designed to resemble the scenarios we want to model and therefore approximate the mode of resource usage in these scenarios. Note, however, that this does not yet guarantee that the benchmarks will provide useful average durations, a simultaneous approximation of the degree of resource usage is needed for that as well.

3.2 Degree of Resource Usage

The degree of resource usage describes how much the resource is used – this concerns details such as capacity, size or rate of requests and replies, parallelism, etc. On our examples of the processor cache and file system resources, degree of resource usage boils down to issues including cache and working set sizes, number of threads and rate of context switches, number of concurrent file readers and writers, etc.

Compared to the mode of resource usage, these issues are relatively easy to quantify. Unlike the mode of resource usage, however, we cannot approximate the degree of resource usage by designing the benchmark experiment so that it resembles the scenario we are modeling. The mode of resource usage is typically an input to performance modeling, but the degree of resource usage is related to the output of performance modeling and therefore not available when designing the benchmark experiment.

To illustrate the argument, consider the examples of the TPC and RUBiS benchmarks. When designing a benchmark experiment that would approximate the database usage in these benchmarks, we can easily approximate the mode of usage, because the specification of the benchmarks includes the types and ratios of queries and updates to be executed. We cannot, however, approximate the degree of usage, because the specification of the benchmarks does not say how frequently the queries and updates are executed or how many queries or updates are executed in parallel. This information is what would be the output of the benchmarks – or the output of performance modeling, if we were to model rather than execute the benchmarks.

3.3 Iterating Between Models

As outlined, we are faced with a situation where we could expect the performance model to give us good results, if only we could feed it with good approximations of the average durations of atomic actions – and we could use benchmark experiments to provide us with good approximations of the average durations, if only we could design the experiments knowing what degree of resource usage to approximate – or, in other words, knowing the results of the performance model. In principle, this situation can be solved by starting with a sensible degree of resource usage and then iterating between using the benchmark experiments to obtain the average durations of atomic actions for the current degree of resource usage and solving the performance model to obtain an updated degree of resource usage. If and when the iteration stabilizes, the results will be based on good approximations of the average durations of atomic actions.

Generally, benchmark experiments tend to be cumbersome and expensive, which is why, as a final step of our approach, we replace them in the iteration by performance models of resources. Together, the performance models of heavily shared resources form a *resource model*, which complements the *performance model* of the software system. In light of the arguments from Section 2, we note that the resource model remains separate from the performance model, giving us freedom in the choice of formalisms and even allowing us to arbitrarily combine the resource model with benchmark experiments when our knowledge of how a resource performs is not sufficient to construct a precise model – as would be the case with examples from Sections 2.1 and 2.2.

An obvious question is whether the iteration between the two models ever converges. As there are no constraints on either of the two models, convergence is generally impossible to guarantee – in fact, for all but trivial cases, a resource model or a performance model that prevents the iteration from converging can be constructed. We can, however, argue that the iteration between the resource model and the performance model that describes certain workload resembles initial behavior of the software system when put under that workload – as the iteration adjusts the resource usage and thus the average durations of atomic actions, so does the software system react to the workload by changing the average durations of function invocations. An oscillation or divergence in the iteration can therefore suggest a tendency towards similar behavior in the software system.

Finally, we should point out that the iteration does not accumulate error. For a given degree of resource usage, the resource model outputs the average durations of function invocations with an error inherent only to that model. The same is respectively true for the performance model. The errors therefore bear influence on the progress of the iteration, but when the iteration stabilizes, the results are precise up to the errors introduced in the last cycle only.

4 Proof of Concept Example

We have chosen the Common Component Modeling Example, or CoCoME [5] for short, as a proof of concept platform for our combined model. CoCoME

describes an enterprise information system that keeps track of *products* sold by a chain of *stores*. When stocked by a store, a product is represented by a *stock item* that has a bar code, a price and an amount as its important attributes. Sale of an item is done at one of the *cash desks* of a store, which locates the item by its bar code, shows its price and, when the sale completes, decrements its amount in the stock.

CoCoME has been created with emphasis on practical usability. The architecture is reasonably large and comes with a reference implementation in Java. This prototype implementation uses contemporary middleware technologies such as ActiveMQ [1], Apache Derby [2] and Hibernate [11], which makes it similar to the platforms modeled recently for example in [16,17,18,24,26,27,29].

When creating our combined model, we have further relied on the fact that the CoCoME architecture has been described within the SOFA component framework [22]. This description includes the deployment plan, which provides us with information on the placement of individual components of the application on the nodes that run it, and the behavior model, which provides us with information on the interactions between individual components. This information is used to construct the resource model.

4.1 Performance Model

The obvious performance related question in CoCoME is how many concurrent sales it can handle. To answer this question, we build our performance model by identifying the activities that make up the sale or that interfere with the sale and representing them explicitly in the performance model. Each sale consists of scanning the bar codes of the items being sold – a scan is followed by a query of the stock item in the database – and of booking the sale – a booking is done by an update of the amounts of stock items in the database. This activity is at the core of our performance model.

Other activities described by the CoCoME architecture, such as handling of customers that do not have enough money to pay for the sale, have been omitted from the performance model. These activities have no impact on how many concurrent sales can be handled, and their omission allows us to keep the model reasonably simple.

We have also decided to simplify activities of the components that are deployed on embedded devices and are unlikely to represent performance bottlenecks – for example, we do not model the bar code scanner or the cash desk display components separately as they are always serving only a single sale and there is little chance that the sale would progress at a speed that the bar code scanner or the cash desk display cannot handle. As a result, each cash desk is modeled by a single component that calls atomic actions at the rate that corresponds to the time elapsing between scans of individual items.

For the formalism of the performance model, we have decided to adopt LQN [28] – the feedback from LQN to the resource model will take the form of queue length and processor utilization values. Another choice would be adopting

SPN [9] – the feedback from SPN to the resource model would take the form of numbers of tokens in selected places. Both LQN and SPN were reported to achieve good results when modeling enterprise information systems [4].

The CoCoME specification defines the rates of requests, size of sales and other properties necessary to seed the performance model with proper constants. Where defined using tabulated distribution functions, we have used averages instead.

4.2 Resource Model

The performance model needs the resource model to provide the average durations of two atomic actions, namely the stock item query and the sale booking update. Benchmarking experiments with the middleware used by the CoCoME reference implementation suggest that these durations are most sensitive to the use of system memory and to the use of database cache, which is why we describe these two resources in our model.

The model of the database cache assumes that the query and the update operations either use cached data or fetch data from disk and that the probability of the two alternatives depends on the relative size of the cache with respect to the size of the database. Similarly, the model of the system memory assumes that the query and the update operations access some resident pages and some swapped out pages and that the frequency of swapping depends on the relative usage of memory with respect to the total amount of available memory.

We therefore start the construction of the resource model by determining the average durations of the two alternatives of the query and update operations by benchmark experiments, see Figure 5. Similarly, we get the additive unit overhead of swapping, which is 162 ms.

Operation	Query time (ms)	Update time (ms)
cached	5.53	75
fetches	8.31	169

Fig. 5. Average durations of the atomic operations.

To determine what is the probability of each variant of the atomic operations, we need to calculate the degree of resource usage in the resource model, which depends on the particular implementation of the resources.

The implementation uses Hibernate for persistent representation of stock items in the database. Hibernate caches data separately for each transaction. The memory usage therefore grows linearly (i) with the number of transactions executing simultaneously and (ii) with the size of the data fetched in each transaction. The number of transactions executing simultaneously is not bounded, the size of the fetched data is bounded by the size of the database.

The database is Apache Derby, which keeps separate context for each connection and caches pages for all transactions together. Its memory usage therefore grows linearly (i) with the number of connections opened simultaneously and (ii) with the size of the data cached for all transactions. The number of simultaneous connections is bounded by the size of the Hibernate connection pool, the cache size is bounded by a configurable maximum cache size.

Considering the configurable maximum cache size with the default value of 10000 pages ($pages_{available}$), 4096 bytes each ($size_{page}$), and a single stock item occupying 460 bytes ($size_{stockitem}$), the probability that a query or an update is cached can be computed as the equation (4.1) suggests.

$$pages_{used} = \frac{size_{page}}{size_{stockitem}} \cdot products \cdot stores$$

$$P_{cached} = \min \left(1, \frac{pages_{available}}{pages_{used}} \right) \quad (4.1)$$

Considering a node with 512 MB of physical memory, of which 451 MB was available for applications ($memory_{available}$), the degree of memory usage is determined by two constituents listed in equation (4.4) – the memory occupied by the code and static data of the components from equation (4.3) (memory shared by the running virtual machines, memory private to each virtual machine, memory private to the database) – and the memory consumed by the concurrent activities from equation (4.2) (memory consumed per database connection, memory consumed per query).

$$usage_{concurrency} = \lceil concurrency \rceil \cdot usage_{connection} \quad (4.2)$$

$$usage_{components} = usage_{shared} + stores \cdot usage_{store} + usage_{database} \quad (4.3)$$

Given the degree of memory usage, the probability that a query or an update does not require swapping can be very roughly approximated by equation (4.5). Note that a rough approximation suffices since swapping should not occur during normal mode of operation, whose performance modeling is of interest.

$$memory_{used} = usage_{components} + usage_{concurrency} \quad (4.4)$$

$$P_{resident} = \min \left(1, \frac{memory_{available}}{memory_{used}} \right) \quad (4.5)$$

Concurrency represents the number of concurrent queries. During iteration, it is provided by the performance model, except for the initial value, which is taken to be zero. The solution of the performance model consists of the throughput and the observed service times of the atomic operations, the number of concurrent queries corresponds to the length of the queue of requests on the database component and is calculated using equation (4.6).

$$concurrency = queue_{length} = \sum_{m \in ops} throughput_m \cdot time_m \quad (4.6)$$

The value of *concurrency* is used by the resource model to recalculate the total memory usage $usage_{concurrency}$ and the probability $P_{resident}$ – the probability P_{cached} does not depend on *concurrency*. The iteration is repeated until the results converge, using a simple ϵ stability criterion.

4.3 Results

The results provided by our combined model in the form of an average throughput in stock items processed per second and an average time spent on each sale are displayed on Figures 6 and 8. The figures plot dependency of the throughput and round-trip values on the number of cash desks per store and the number of stores per enterprise as the scalability factors.

On average, the combined model needed only three iterations to converge. This is because the memory consumption per query is relatively low compared to the total memory consumption. The value of the *concurrency* variable at the end of the iteration was approximately 1.24 for a single store with 8 cash desks, 5.16 for two stores, 11.57 for three stores and growing rapidly. We can therefore conclude that the effect of memory consumption per query is not negligible.

For comparison, the results of a real benchmark of the CoCoME reference implementation are on Figures 7 and 9. The benchmark, as well as the benchmark experiments used to obtain the average durations of the atomic actions, have used an Intel Pentium 4 Xeon 2.2 GHz machine with 512 MB RAM running Fedora Core 6 and the database cache of 10000 pages for the server machine, and a dual Intel Core 2 Quad Xeon 1.8 GHz machine with 8 GB RAM running Fedora Core 6 for the client machine. The relatively low amount of memory on the server machine was used deliberately to allow the manifestation of swapping with a reasonably small number of cash desks and stores.

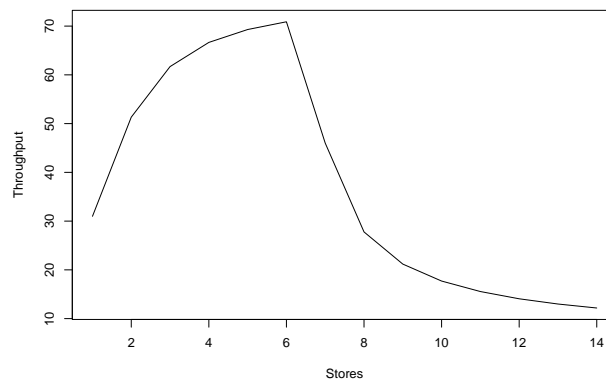


Fig. 6. Throughput calculated from the combined model (8 cash desks per store).

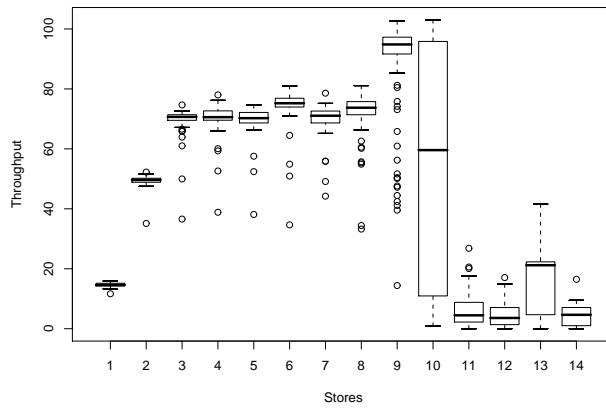


Fig. 7. Throughput benchmarked on the prototype (8 cash desks per store).

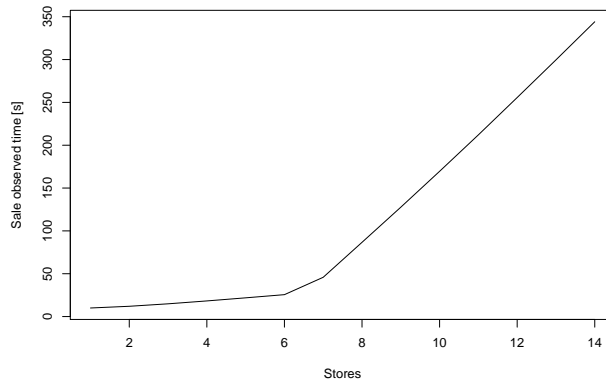


Fig. 8. Round-trip calculated from the combined model (8 cash desks per store).

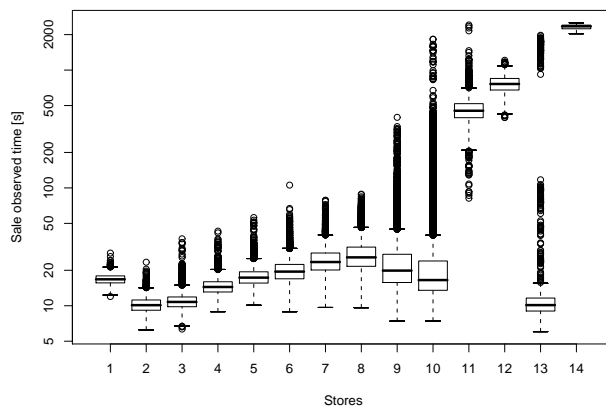


Fig. 9. Round-trip benchmarked on the prototype (8 cash desks per store).

5 Evaluation

One approach to evaluating our method is comparing the results predicted by the combined model on Figures 6 and 8 with the results measured by the real benchmark on Figures 7 and 9. This comparison suggests that the approach is reasonably precise – the maximum throughput predicted by the model is 71 stock items per second, the maximum throughput measured by the benchmark is 74 stock items per second. Similarly, the duration of a sale predicted by the model is 10 seconds for one store (with 8 cash desks) and grows to around 26 seconds before the server starts swapping, the duration of a sale measured by the benchmark is 17 seconds for one store and grows to around 25 seconds before the server starts swapping. If a better precision were desired, the usual calibration of the model could be employed as in [24,26].

As stated in the introduction, however, our goal is not predicting the exact values of round-trip and throughput, but predicting how the values of round-trip and throughput change when the scale of the application changes. Comparing the results, we see that our approach predicted getting within 10% of the maximum throughput around 2 stores, when the measurement shows this happening around 3 stores. Our approach also predicts the onset of swapping and the associated degradation in throughput around 8 stores, when the measurement shows this happening around 10 stores. The difference can be explained by our inability to determine the precise memory requirements of the individual components, something that is difficult to do with current tools.

When comparing our results to the results of other researchers, we did not achieve that good a precision in predicting the values of round-trip and throughput. We believe that this is not inherent to our approach but specific to the proof of concept performance model, which has been kept intentionally simple – our approach allows us to easily reuse performance models such as those in [24,26].

We note, however, that of the models of similar platforms in [16,17,18,24,26,27,29], none is able to predict the onset of swapping and the associated degradation of performance due to resource sharing of computer memory. We believe that this is a result of practical importance in performance modeling.

Finally, we should point out that early work recognizing the importance of resource models in addition to performance models exists, such as [25]. There, complexity functions in workload models are introduced to provide the contention model, which corresponds to the performance model in our terminology, with information on resource usage. The work, however, still expects that resource sharing will be described together with component interaction inside the contention model, which is something that we argue is too complex.

The idea of iterating between the resource model and the performance model can also be seen as an extension of earlier iterative approaches to solving the performance models. For illustration, [15] describes solving the performance model iteratively in initialization and simulation phases, with the initialization phase calculating service times and routing probabilities based on an assumed value that the simulation phase adjusts. The work, however, gives no details on this

value and views the entire iteration as an unfortunate consequence of the performance model complexity.

6 Conclusion

We have described an approach to performance modeling of software systems based on creating a combined model, in which a resource model and a performance model complement each other in an iterative calculation. The advantage of the approach is that it allows modeling heavily shared resources separately from other concerns, keeping the combined model reasonably simple.

We use an example of an enterprise information system to show that even with very simple resource and performance models, the effects of resource sharing can be predicted. We use the average durations of only four variants of two atomic actions determined by benchmarking experiments for the resource model, and only six tasks in the performance model. With that, we predict the effects of both database cache sharing and system memory sharing, including the conditions of resource exhaustion, which are rarely modeled elsewhere.

Our approach is based on the assumption that resource usage can be described as consisting of two factors – mode of usage and degree of usage, where the mode is determined by the scenario we are modeling and captured by the resource model, and the degree is determined by the performance of the system we are modeling and provided by the performance model. We believe that our approach has a potential to work for heavily shared resources such as physical memory or file systems, where long periods of stable execution are considered. In these situations, the effects of resource sharing can often be modeled relatively simply in the resource model, without cluttering the performance model.

An obvious requirement of our approach is being able to model the behavior of individual resources. In this aspect, our approach retains the options of using benchmark experiments to measure the resource, using a separate performance model to model the resource, or modeling the resource as a part of the integrated performance model of the entire software system. We should therefore always be better off than approaches that only have one integrated performance model.

Acknowledgments. The authors would like to thank the team working on the SOFA model of the CoCoME architecture for providing a supporting framework, especially the deployment plan and the behavior model. This work was partially supported by the Grant Agency of the Czech Republic project GD201/05/H014 and by the Czech Academy of Sciences project 1ET400300504.

References

1. ActiveMQ, <http://activemq.apache.org>
2. Apache Derby, <http://db.apache.org/derby>
3. Agarwal A., Hennessy J., Horowitz M.: *An Analytical Cache Model*, TOCS 7(2), ACM 1989

4. Balsamo S., DiMarco A., Inverardi P., Simeoni M.: *Model-Based Performance Prediction in Software Development*, TSE 30(5), IEEE 2004
5. CoCoME, <http://agrausch.informatik.uni-kl.de/CoCoME>
6. Drakopoulos E., Merges M. J.: *Performance Analysis of Client-Server Storage Systems*, TC 41(11), IEEE 1992
7. Frigo M., Johnson S.G.: *FFTW*, <http://www.fftw.org>
8. Ghosh A., Givargis T.: *Cache Optimization for Embedded Processor Cores: An Analytical Approach*, TODAES 9(4), ACM 2004
9. Haas P. J.: *Stochastic Petri Nets: Modelling, stability, simulation*, Springer 2002
10. Hauswirth M., Diwan A., Sweeney P. F., Mozer M. C.: *Automating Vertical Profiling*, OOPSLA'05, ACM 2005
11. Hibernate, <http://www.hibernate.org>
12. Hossain A., Pease D. J.: *An Analytical Model for Trace Cache Instruction Fetch Performance*, ICCD'01, IEEE 2001
13. Kalibera T., Bulej L., Tuma P.: *Benchmark Precision and Random Initial State*, SPECTS'05, SCS 2005
14. Kannan H., Guo F., Zhao L., Illikkal R., Iyer R., Newell D., Solihin Y., Kozyrakis C.: *From Chaos to QoS: Case Studies in CMP Resource Management*, SIGARCH CAN 35(1), ACM 2007
15. Kant K., Sundaram C. R. M.: *A Server Performance Model for Static Web Workloads*, ISPASS'00, IEEE 2000
16. Kounev S., Buchmann A.: *Performance Modeling of Distributed E-Business Applications using Queuing Petri Nets*, ISPASS'03, IEEE 2003
17. Liu Y., Gorton I.: *Performance Prediction of J2EE Applications Using Messaging Protocols*, CBSE'05, Springer 2005
18. Liu Y., Fekete A., Gorton I.: *Predicting the Performance of Middleware-Based Applications at the Design Level*, WOSP'04, ACM 2004
19. Pentakalos O. I., Menasce D. A., Halem M., Yesha Y.: *An Approximate Performance Model of a Unitree Mass Storage System*, MSS'95, IEEE 1995
20. Pimentel A. D., Thompson M., Polstra S., Erbas C.: *On the Calibration of Abstract Performance Models for System-Level Design Space Exploration*, SAMOS'06, IEEE 2006
21. RUBiS, <http://rubis.objectweb.org>
22. SOFA Component Model, <http://dsrg.mff.cuni.cz/sofa>
23. TPC Benchmarks, <http://www.tpc.org/information/benchmarks.asp>
24. Ufimtsev A., Murphy L.: *Performance Modeling of a JavaEE Component Application using Layered Queuing Networks: Revised Approach and a Case Study*, SAVCBS'06, ACM 2006
25. Vetland, V.: *Measurement-Based Composite Computational Work Modelling of Software*, Doctoral thesis, University of Trondheim 1993
26. Xu J., Oufimtsev A., Woodside C. M., Murphy L.: *Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queuing Network Templates*, SIGSOFT SEN 31(2), ACM 2006
27. Xu J., Woodside C. M.: *Template-Driven Performance Modeling of Enterprise Java Beans*, MWS'05, IEEE 2005
28. Woodside C. M., Neron E., Ho E. D. S., Mondoux B.: *An Active-Server Model for the Performance of Parallel Programs Written Using Rendezvous*, JSS 6(1-2), Elsevier 1986
29. Wu X. P., Woodside C. M.: *Performance Modeling from Software Components*, WOSP'04, ACM 2004