Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 182 (2007) 91–106
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Parametric Performance Contracts for Software Components with Concurrent Behaviour

Jens Happe[1] Heiko Koziolek[2] Ralf Reussner[3]

Graduate School Trustsoft[4]
University of Oldenburg
26121 Oldenburg, Germany

Chair for Software Design and Quality
University of Karlsruhe
76131 Karlsruhe, Germany

Abstract

Performance prediction methods for component-based software systems aim at supporting design decisions of software architects during early development stages. With the increased availability of multicore processors, possible performance gains by distributing threads and processes across multiple cores should be predictable by these methods. Many existing prediction approaches model concurrent behaviour insufficiently and yield inaccurate results due to hard underlying assumptions. In this paper, we present a formal performance prediction approach for component-based systems, which is parameterisable for the number of CPUs or CPU cores. It is able to predict the response time of component services for generally distributed execution times. An initial, simple case study shows that this approach can accurately predict response times of multithreaded software components in specific cases. However, it is limited if threads change the CPU during their execution, if the effect of processor cache thrashing is present, and if the memory bus is heavily used.

Keywords: performance prediction, parametric performance contracts, service time distribution, software components, stochastic regular expressions, multicore processor, multithreaded behaviour

1 Introduction

An important extra-functional property of component-based software systems is *performance*, often expressed via metrics such as response time, throughput, or resource utilisation. Formal techniques for reasoning about the performance of software components during early development stages currently receive increasing attention [1].

[1] Email: jens.happe@informatik.uni-oldenburg.de
[2] Email: heiko.koziolek@informatik.uni-oldenburg.de
[3] Email: reussner@ipd.uka.de
[4] This work is supported by the German Research Foundation (DFG), grant GRK 1076/1

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2006.09.033

The goal of such *performance prediction methods* is to support early design decisions regarding component-based software architectures. These methods combine (a) performance specifications from *component developers*, (b) architectural descriptions from *system architects*, (c) hardware specifications from *system deployers*, and (d) usage scenarios from *domain experts* to make estimations about the expected performance of an application before actually implementing it.

With the recent shift of processor manufactures to build mainstream *multicore processors*, component-based performance prediction methods are required to make adequate predictions parameterised for the number of processor cores. Predictions could then support purchasing proper hardware for specific application contexts. Given a fixed hardware environment, predictions could help system architects to exploit the available hardware as good as possible.

Many existing performance prediction approaches are based on *analytical models* such as queuing networks, stochatic Petri nets, or stochastic process algebras [2]. Although multicore systems can be modelled with these formalisms, reality is often inaccurately reprensented by them because of hard underlying assumptions. For example, most of these formalism assume exponential distributions for execution times or only compute response times as mean values [3]. Because of the many influencing factors on performance in large enterprise systems, mean response times are often not useful. Predicting response times as distribution functions can be more useful to support design decisions.

In this paper, we extend our former formal approach for performance prediction of component-based software architectures [4] to explicitly include aspects of multithreaded behaviour. We require component developers to specify parametric performance contracts for their components in form of so-called service effect specifications. Component assemblers can then parameterise these specifications for their environment and the number of processor cores. Service effect specifications, which now allow the forking of threads, are transformed into an analytical model based on stochastic regular expressions [4]. Solving the analytical model yields response times for component services as arbitrary distribution functions.

The contribution of this paper is an initial approach to include multithreaded behaviour in component-based performance predictions. We report on a first case study to validate our approach in specific situations. In the case study, we have compared predictions of our method with measurements of an implemented example system. Although simple, the case study was suited to reveal more challenges for the prediction of the performance in multicore systems, such as CPU hopping and cache thrashing. We have summed up lessons learned from measuring a dual-core system and provide directions for future research.

This paper is structured as follows: Section 2 introduces our design model, which is based on annotated UML diagrams. Section 3 describes our analytical model, which is based on stochastic regular expressions, and explains the necessary computations to solve it, especially for the newly introduced parallel operator and summs up assumptions and limitations of our approach. In Section 4, we report on an initial case study and discuss our results. Section 5 analyses related work

in the area of performance prediction and component-based software architectures. Section 6 concludes the paper with an outlook on future work.

2 Design Model: UML Models

2.1 Overview

During early development stages of a component-based software system, reasoning about the quality attributes of the system has to be based on models, since an implementation is often not or only partially available. It is an established practice in the software performance engineering community to use a design-oriented model for specification and transform it into an analysis-oriented model to predict performance attributes [2]. The *design-oriented model* is often based on UML as the de-facto standard modelling language and uses extensions like the UML SPT profile [5] and self-defined semantics to include information related to performance.

We follow the approach outlined above and base our design model on UML keeping in mind that component-based development usually involves several developer roles as discussed in [6].

2.2 Component Specification

Component developers specify provided and required interfaces of their components (see Figure 1(a)). For performance prediction, additional information about the internal structure of the component is needed. Thus, in our approach, a parametric contract in form of a so-called service effect specification (SEFF) has to be specified for each provided service of a component. A SEFF is an abstraction of the control flow through the provided service [7]. It describes how the provided service calls services specified in the required interfaces. Here, SEFFs are modelled as special UML 2.0 activities, where each action represents a call to a required service. The exemplary SEFF in Figure 1(b) includes the control flow primitives supported by our analysis: Sequence, alternative, loop, and fork. Upon invocation of provided service `Z`, first required service `A` is called, and then either service `B` or `C`. Service `B` is called multiple times within a loop. After service `C` has been called, services `D`, `E`, and `F` are invoked concurrently. Finally, the control flow is joined again, and service `Z` ends its execution.

2.3 Stochastic Annotations

The structural information contained in a SEFF is not sufficient for performance analysis. We need additional stochastic information, namely transition probabilities on branches, number of loop iterations, and time consumptions of the service itself as well as its called services.

Transition probabilities and number of loop iterations cannot be specified by the component developer directly, because these figures often depend on how the component is used by third parties, which is unknown to the component developer. We have shown in [6] how component developers can specify the dependencies between

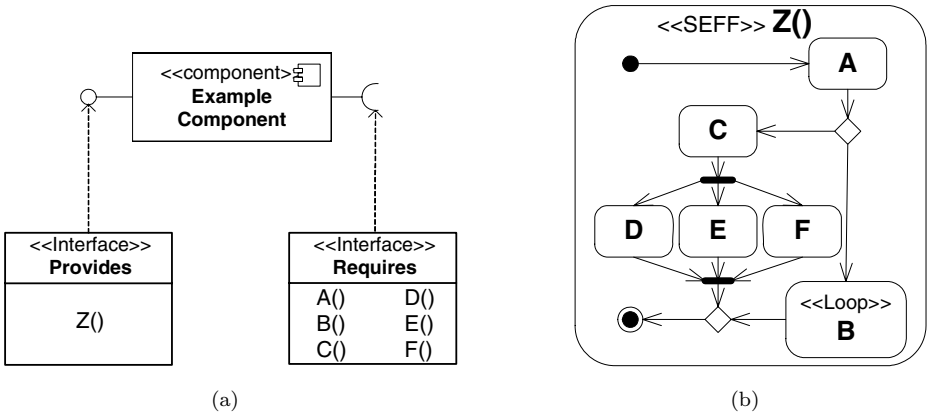


Fig. 1. Example Component and corresponding Service Effect Specification

input parameters and transition probabilities or the number of loop iterations, so that system architects can take these specifications and adjust them to their expected usage profiles. For the scope of this paper, we assume that the necessary information is available.

To model the time consumption of a service, we assign a random variable X to each action in the SEFF. The underlying probability mass function (PMF)

$$x : t \longmapsto p_t = P(X = t)$$

assigns a probability p_t to each execution time t, which can for example be given in milliseconds.

The time consumptions of external services can either be the result of another service or can be specified by the system architect. For the component's internal activities, the time consumption can be based on measurements or estimations using SPE techniques [8]. The stochastic information is included into the SEFF using the UML SPT profile [5].

3 Analytical Model: Stochastic Regular Expressions

To predict the response time of a service, the design model is transformed into an analytical model. We use stochastic regular expressions (SRE) [4] for this purpose and extend them with an operator for parallelism. The transformation maps the structural elements of activity charts to regular expressions. Performance relevant information present in the design model, like branching probabilities, loop iteration functions, and random variables for time consumption, are passed to the corresponding elements of the resulting stochastic regular expression.

The metamodel of SREs contains classes for symbols, alternatives, sequences, loops, and parallelism wich are specialisations of general expressions. Their interpretation is similar to the one of regular expressions, where symbols are known as terminal symbols and loops as Kleene stars. Each expression contains a probability mass function for its execution time. Furthermore, alternatives associate a proba-

bility to each option. For each loop, a probability mass functions characterises its number of iterations. The new operator parallel extends the usual regular expressions to model the parallel execution of two independent tasks. It can be interpreted as the forking of two threads or processes and their joining. For sake of simplicity, we omit synchronisation mechanisms like semaphores and monitors.

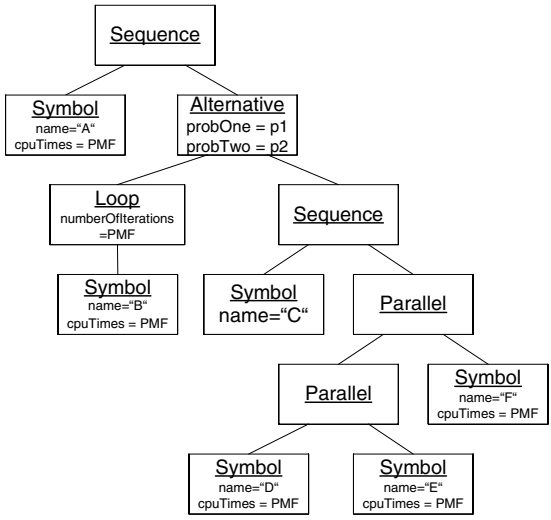


Fig. 2. Example of a Stochastic Regular Expression

Transforming the service effect specification shown in figure 1(b) to a SRE yields the abstract syntax tree shown in figure 2. The external service calls `A` to `F` can be found at the leafs of the tree. The concurrent execution of the operations `D`, `E`, and `F` is mapped to two `Parallel` nodes on the right hand side. The loop in the SEFF is mapped to a `Loop` node whose inner expression is `B`.

3.1 Model Analysis: Sequence, Alternative, Loop

To compute the execution time of the complete expression, the abstract syntax tree is traversed from its leaves to its root, computing the time consumption of each node. The execution time calculated for the root node is the response time of the component's provided service. In the following, we describe the computations for the operators sequence, alternative, and loop. Concurrency is introduced afterwards, extending the analysis to multiple CPUs and a new parallel operator.

The time consumption for a **sequence** is the sum of the time consumptions of each child node. The sum of two random variables is the convolution of their probability mass functions, if the random variables are independent [3, pp.106]. Hence under the assumption of independence, the time consumption of a sequence can be computed by:

$$x_{R_1 \cdot R_2}(t) = x_{R_1} \circledast x_{R_2}[t]$$

For an **alternative**, the time consumption is computed as the sum of the alternative paths weighted by the branch probabilities. The corresponding probability mass

functions are:

$$x_{R_1+R_2}(t) = p_1 x_{R_1}[t] + p_2 x_{R_2}[t]$$

where p_1 and p_2 are the probabilities of either choosing alternative one or two, respectively.

For **loops**, a probability mass function $l(i)$ is specified containing a probability for each number of loop iterations. Thus, the resulting probability mass function for the loop has the following form:

$$x_{R^l}(t) = \sum_{i=1}^{N} l(i) \overset{i}{\underset{j=1}{\circledast}} x_R[t]$$

with $N \in \mathbb{N}_0$ and $\forall i > N : l(i) = 0$. To compute the convolutions of the probability mass functions we use the discrete Fourier transform as described in [9], where also the computational complexity of the operations is discussed in detail.

3.2 Model Analysis: Parallel

To analyse concurrency, we extend stochastic regular expressions to multiple processing resources. We focus on symmetric multiprocessing (SMP) architectures where multiple CPUs of the same type are connected to a single shared memory.

Considering multiple processing resources and concurrency in service effect specifications leads to changes in stochastic regular expressions compared to [4]. In the single-threaded case, the execution time of a service is modeled by a single probability mass function. This is not sufficient for concurrent systems with multiple processing resources, since a service might utilise multiple processing resources and the usage depends on its degree of parallelism. To reflect this by the analytical model, we extend regular expressions with a computation time for each available processing resource. Here, *computation time* is the time that a service uses a processing resource. Consequently, the execution time of a service is the maximum of its computation times (assuming that there are no dependencies between the concurrent threads).

The **parallel** operator combines the computation times of its child nodes to optimally use the available processing resources. Each child expression might itself contain concurrent parts (e.g. it can be a parallel operator itself) and can thus contain multiple computation times. The mapping of all incoming computation times to the available processing resources should be optimal meaning that the maximum of all time consumptions is minimal for all possible mappings. In other words, the execution time of the parallel expression is minimised by using as much parallelism as possible.

However, in our case the optimisation is constrained. First, we do not allow a task to change its processing resource during execution. Second, we optimise only according to local parameters of the parallel operator and neglect the global view on the system. Thus, if multiple parallel operators are combined, a better scheduling might exist than found by our algorithm. The local optimisation also makes it questionable whether the parallel operator is associative or not. However,

as an approximation of actual system behaviour, the approach presented here is sufficient.

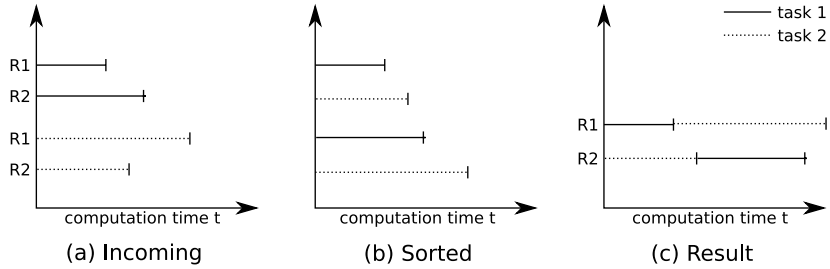
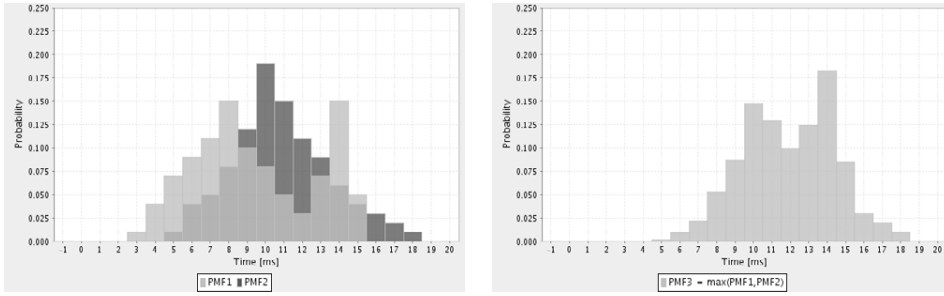


Fig. 3. Example compuatation for the parallel operator.

Figure 3 illustrates the algorithm of the parallel operator with an example. `task 1` and `task 2` are two expressions that shall be executed in parallel. Both are using the processing resources `R1` and `R2`. Figure 3(a) shows the computation time of both tasks on each resource. These four values need to be mapped on the two available resources `R1` and `R2`. To do so, we first sort the computation times (see Figure 3(b)). Then, the shortest and the longest computation times are added and assigned to `R1` (see Figure 3(c)). For the remaining computation times in figure 3(b), we proceed in the same way. The shortest and the longest computation times of the remaining ones are added and assigned to `R2`. The resulting computation times are shown in figure 3(c). The algorithm can be summarised as follows:

(i) Create a sorted list of computation times from the child nodes' computation times

(ii) Repeat until the list is empty:
  (a) Take the longest and shortest computation time from the list
  (b) Add them and store the result as one of the new computation times.



(a) (b)

Fig. 4. Example Max-PMF

For mean values, as used in the example, the computations are straight forward. Unfortunately, the sorting of random variables that are specified by probability mass functions (as used in our approach) is much more complicated. To illustrate this, consider figure 4(a), where the maximum of two random variables X_1 and X_2 shall be determined. The probability mass functions of both random variables are

strongly overlapping. Thus, neither X_1 nor X_2 can be said to be the maximum, since both cases are possible. However, a new random variable $X_{max} = max(X_1, X_2)$ can be determined by multiplying the cumulative distribution functions of X_1 and X_2 [3, p.109]. Figure 4(b) shows the result. For sorting, the problem is similar, as the actual order depends on the concrete values of the random variables. Thus, we have to find a way to compute a new random variable for the nth computation time from the incoming ones.

Sorting Random Variables

To sort a list of N computation times and determine the PMFs of the new random variables, we proceed as follows. For each incoming computation time X_n, the new PMF is set to $P_n(X = t)$, which contains the probability that up to time t at least n tasks are computing. In the unsorted set, each random variable is specified by a PMF $Q_n(X = t)$. From this PMF, the probability that the task is still computing at time t $Q(X_n > t)$ is determined as well as the probability that it is already finised $Q_n(X \leq t)$. Based on this, the probability that exactly n tasks are running at time t can be derived, as done in the following example with three concurrent tasks:

$$\begin{aligned} P_1'(X > t) \quad = \quad & Q_1(X > t)Q_2(X \leq t)Q_3(X \leq t) + \\ & Q_1(X \leq t)Q_2(X > t)Q_3(X \leq t) + \\ & Q_1(X \leq t)Q_2(X \leq t)Q_3(X > t) \end{aligned}$$

where $P_1'(X > t)$ is the probability that exactly one of three tasks is running. Each of the products yields the probability that either the first, second, or third task is active, while the others are already finished. Summing up the probabilities yields the probability that exactly one of the tasks is running. The general case can be formulated as follows:

$$P_n'(X > t) = \sum_{I \in \mathcal{P}(T),\ |I| = n} \left[\prod_{i \in I} Q_i'(X > t) \prod_{j \in T \setminus I} Q_j'(X \leq t) \right]$$

where $T = \{1, \ldots, N\}$ is the set of concurrent tasks, $\mathcal{P}(T)$ its power set, and I is a selection of n tasks from the set. For each possible combination of n tasks, the probability that exactly these tasks are running (first product) and all other tasks are finished (second product) is calculated for each possible combination of n tasks and then summed up. The probability that n or more tasks are running can be calculated by simply adding the probabilities from n to N running tasks.

$$P_n(X > t) = \sum_{i=n}^{N} P_i'(X > t)$$

Form this, $P_n(X = t)$ can be determined easily. The result of this computation is a sorted list of N probability mass functions where $P_1(X = t)$ characterises the longest and $P_N(X = t)$ the shortest computation time. Now, we can apply the algorithm

as described in the beginning by adding (convolving for PMFs) the shortest and longest computation times.

The computational complexity of the parallel operator increases exponentially with the number of processing resources. However, since the number of available processors or CPU cores is limited, this should not constrain the applicability of the analysis.

3.3 Assumptions

Our approach makes some assumptions about the availability of data and the behaviour of the modelled system. In [6], we describe how the required data can be obtained in a component-based development process.

For the behaviour of the system, we assume that a task cannot switch the CPU. Additionally, scheduling is assumed to be optimal, as tasks are immediately scheduled to free CPUs. Furthermore, we neglect the overhead created by task switching. So far, we do not model locking or synchronisation mechanisms, since we are focussing on the influence of the concurrent execution of independent tasks on performance. We consider only CPUs of the same type and do not include other resources, such as memory, disks, or networks.

4 Case Study

The case study described here intends to analyse the validity of the new parallel operator, and thus does not model an industry size architecture. Instead, we use a rather simple architecture employing concurrency. We are planning larger case studies in the future.

Any validation of a performance prediction method must compare the prediction results with measurements performed on an implementation of the analysed architecture. Thus, we have created a design model for a simple program with multiple threads, implemented it in Java, and performed measurements on the implementation. The questions we asked ourselves before the case study were:

- How much do the predictions of our analytical model differ from measurements of the implementation?
- What kind of multithreaded behaviour can be analysed accurately?

4.1 Analysed Architecture

The design model of the analysed architecture is depicted in Figure 5. The system consists of a client component invoking a server component with concurrent requests. In the design model, we deployed both components on a server with a dual core CPU. For the provided service `performCalculation` of the client component, the corresponding SEFF is shown in Figure 5(b). It spawns three threads and calls the required service concurrently.

We implemented different required services to reduce the distortion of the results

by different algorithms. Two computationally complex, but less memory intensive algorithms were implemented. Of those algorithms, one calculated an array of prime numbers larger than a given integer (Primes). The other one calculated a fractal (Mandelbrot). Furthermore, two algorithms were using a large amount of memory. The first algorithm generated a large array of random numbers and sorted them (Sorting). The second algorithm performed a fast Fourier transform on a probability mass function (FFT).

Deriving the analytical model from the design model is straightforward. Thus, we omit an illustration of the analytical model for brevity.

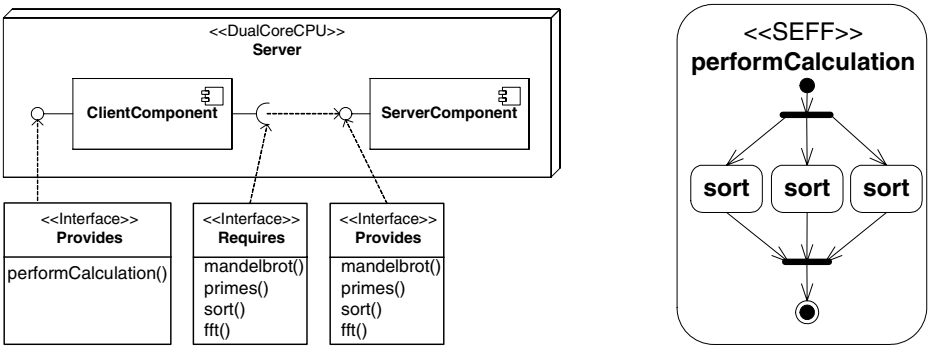


Fig. 5. Case Study Architecture and Example SEFF for the use case "Sorting"

4.2 Implementation and Case Study Setup

We implemented the architecture and the four algorithms described in the previous section in Java. The measurements discussed in the following were performed on a dual-core Pentium D with 3 GHz and Windows Server 2003 as operating system. During measurement the provided service of the client component was called repeatedly 500 times for all scenarios, and the response times were saved as probability mass functions.

We adjusted the parameters of the algorithms (e.g. number of generated random or prime numbers) so that their response time for a single execution with one active core was about 50ms (short) or 500ms (long).We chose the following independent variables for the experiments: (a) memory intensive vs. CPU intensive algorithms, (b) short (50ms) vs. long (500ms) execution times, (c) one vs. two active processor cores and (d) sequential vs. parallel execution of an algorithm.

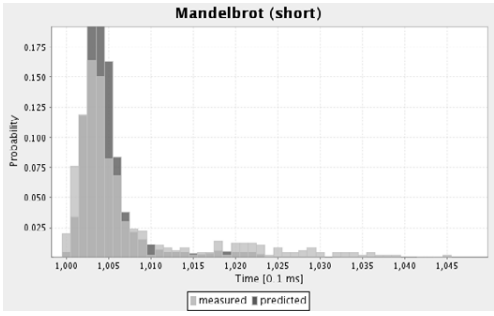
The only dependent variable is the response time of the client component's provided service. As input data for our computations, we used the measured response times of the algorithms for the single threaded execution with one active processor core. So, the parallel operator is the only influencing factor for the predictions. From the independent variables, we created the following scenarios:

(i) CPU intensive, two active cores, parallel execution
- short execution time

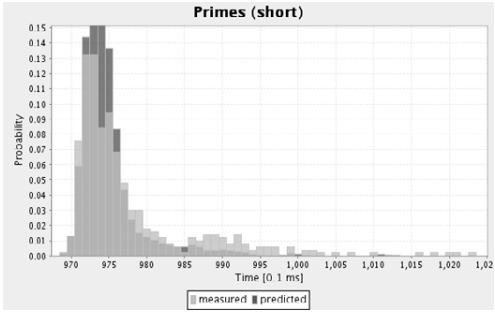
- long execution time

(ii) Memory intensive, short execution times
- sequential and parallel execution for one core
- sequential and parallel execution for two cores

4.3 Results: 1. Scenario (Predictions vs. Measurements)



(a)

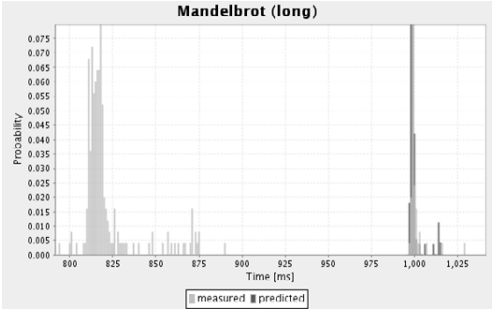


(b)

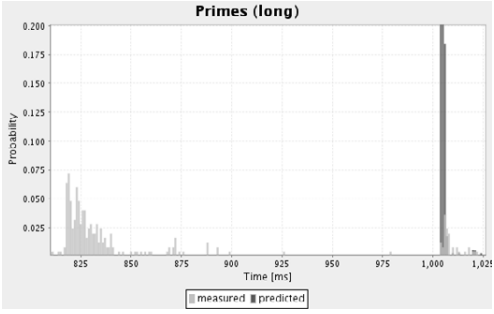
Fig. 6. Predictions vs. measurements for short task execution times.

Figure 6 shows predicted and measured response times for the first scenario with *short* execution times (50ms per task). Predicted and measured probability mass functions are strongly overlapping. Also the mean values of the measured functions are with 100.9ms (Mandelbrot) and 98.1ms (Primes) similar to the predicted values of 100.6ms (Mandelbrot) and 97.6ms (Primes). As an effect of the convolution, the computed PMF is smoother than the measured one. The measurements have a higher variance than the predictions (10.6/10.9 opposed to 0.9). During the measurements, the utilisation of both CPUs was about 70%, because the tasks are joined in the end and have to wait for each other before continuing.

As evidenced in figure 7, the measured *long* execution times (500 ms per task) for three concurrent threads are about 180ms faster than the predictions in most cases. This phenomen might be caused by the scheduling algorithm of the operating system, which moves tasks among the CPUs to get a balanced utilisation (so-called *CPU-Hopping*). In our prediction model, we assume that tasks cannot be moved



(a)



(b)

Fig. 7. Predictions vs. measurements for long task execution times.

from one CPU to another during their execution. Our assumption obviously does not hold for long time executing tasks.

Interestingly, some of the measured execution times exactly match the predicted ones. These values are outliers and form a second peak in the distribution function of the measurements. This behaviour could be observed for completly different algorithms (Mandelbrot and Primes), and is thus not related to characteristics of the code. The scheduling algorithm of the operating system is a possible explanation for this behaviour. The outliers are measurements from tasks that have not been moved among the processors. In these cases, the actual execution time matches our prediction. In general, our prediction match the worst-case execution times in this scenario, and thus should nevertheless be suited to make rough estimations and support early design decisions.

4.4 Results: 2. Scenario (Further Measurements)

For scenario 2, we only provide measurements and no predictions, since the measurements already violated our assumptions and thus we could not make useful predictions. The measurements in figure 8 show the execution times of the memory intensive algorithms (Sorting and FFT) for one and two active processor cores. We measured the response times of the sequential execution and the concurrent execution.

Figure 8(a) and 8(c) show how the response time of the client's component

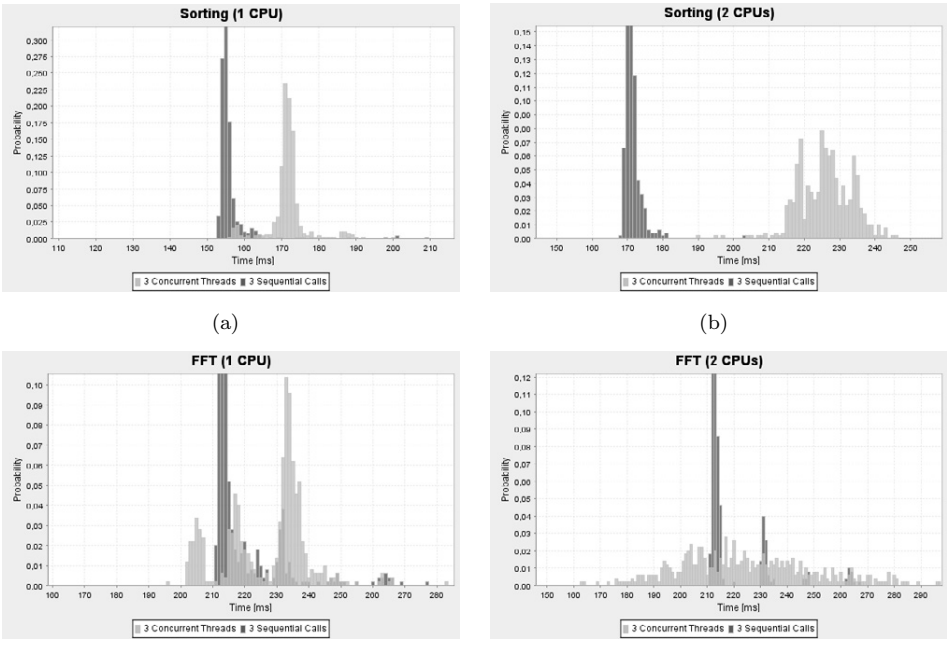


Fig. 8. Measurements of memory intensive algorithms.

provided service is affected by multi-threading on a single CPU. In figure 8(a) (Sorting), the response time of the concurrent execution is about 20ms slower than the sequential execution. This is a performance loss of about 15%, because multi-threading creates additional overhead due to task switching.

In figure 8(c), the concurrent execution of the FFT algorithm with three threads yields the probability mass function with three different peaks. As an effect of the disturbing non-deterministic influences of the scheduler, the variance of the concurrent algorithms is much higher than the variance of the sequential execution.

Figure 8(b) shows an unexpected effect: The concurrent execution of of three sorting algorithms on a dual-core processor is about 55ms slower than its sequential execution on the same CPU. Moreover, the sequential execution is already about 15ms slower than on a processor with one core. The parallel execution on a dual-core processor yields a performance loss of 70ms (i.e., about 50%).

A dual-core Pentium D has only one memory bus that can be used by one core at a time. For concurrently executing memory intensive algorithms, the memory bus becomes a bottleneck. However, this does not explain the large difference to the sequential execution. A further delay might be caused by a higher number of cache invalidations as a result of continuous writing by both cores. This is known as cache thrashing.

4.5 Lessons Learned

Coming back to the initial questions of the case study, we can state for the first question, that for algorithms with a low memory footprint, we can make accurate predictions (even if they only reflect the worst case behaviour). However, for algorithms with a high memory footprint our approach would yield inaccurate results.

First, we did not include memory access in our model. Thus, we would have predicted similar execution times for memory intensive algorithms as in the first scenario. As we have seen in the second scenario, the memory bus can become a bottleneck and, thus, has to be modelled for accurate predictions. Second, we did not include internal information about the CPU, like cache misses or pipeline invalidations. However, we can predict the response times for the parallel execution of CPU intensive tasks with reasonable accuracy.

Furthermore, the assumption that tasks cannot be moved among different CPUs or processor cores is not valid. If the task execution time exceeds certain limits, the scheduler starts to move tasks among the CPUs to get a more balanced utilisation.

5 Related Work

The transformation of UML design documents with performance annotations like the UML profile for schedulability, performance and time (UML SPT profile) [5] is a common approach in the performance prediction community. For example, the CB-SPE tool of Bertolino et al. [10] uses UML diagrams that are annotated with the UML SPT profile and transforms them into a queueing network model which is then solved to get the required performance metrics. Balsamo et. al. [11] generate event-driven simulations from annotated UML models to make performance predictions. Petriu et. al. [12] use graph-grammar based transformations to derive layered queueing networks from annotated UML specifications. Due to the large number of design and analytical models, Grassi et al. [13] introduced an intermediate language called KLAPER, which is supposed to ease the transformation between design oriented models and analytical models.

On the analytical side, queueing network models, stochastic Petri nets , and stochastic process algebras like PEPA [14] are among the most established analytical performance prediction approaches [11]. Even though these models provide completely different formalisms for system specification, they are mostly transformed to continuous time Markov chains for analysis. This is followed by a lot of well known mathematical assumptions that limit the applicability of these models. Exponential distributions for time consumptions and the Markov property (transition probabilities/rates depend on the current state only and are independent of the history) are the most important ones.

6 Conclusions and Future Work

In this paper, we present a formal performance prediction approach for component-based systems that is parameterisable for the number for CPUs or processor cores.

It utilises design models in UML provided by component developers and system architects and transforms them into an analytical model based on stochastic regular expressions. We introduce a new parallel operator to analyse multithreaded behaviour. A case study shows that the approach is able to predict accurate response times if threads do not change CPUs, the scheduling is optimal, no synchronisation is necessary, and the effect of cache thrashing can be neglected.

In the future, this approach could be useful for system architects because they will be able to make rough estimations of the expected performance of their architectures without actually implementing them. This reduces problems due the well-known fix-it-later approach to performance issues in software development, and possibly saves costs for subsequent refactorings and redesigns of implementations.

Our case study shows that the approach is still very limited in predicting multithreaded behaviour. The effects of CPU hopping and cache thrashing vastly distort the results and need to be included into our analytical model in the future. Moreover, synchronisation mechanisms such as semaphors or monitors do have an important influence on the performance of a component service and have been neglected so far. Larger case studies with industrial size component-based systems on multicore processors are needed to validate the approach further.

References

[1] Becker, S., Grunske, L., Mirandola, R., Overhage, S.: Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective. In: Architecting Systems with Trustworthy Components. To Appear in LNCS. Springer (2006)

[2] Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. IEEE Transactions on Software Engineering **30** (2004) 295–310

[3] Trivedi, K.S.: Probability and Statistics with Reliability, Queuing and Computer Science Applications. 2nd edn. Wiley & Sons, New York, NY, USA (2001)

[4] Koziolek, H., Firus, V.: Parametric Performance Contracts: Non-Markovian Loop Modelling and an Experimental Evaluation. In: Proceedings of FESCA2006. Electronical Notes in Computer Science (ENTCS) (2006)

[5] Object Management Group (OMG): UML Profile for Schedulability, Performance and Time. http://www.omg.org/cgi-bin/doc?formal/2005-01-02 (2005)

[6] Koziolek, H., Happe, J.: A Quality of Service Driven Development Process Model for Component-based Software Systems. In: Component-Based Software Engineering. Volume 4063 of Lecture Notes in Computer Science., Springer-Verlag GmbH (2006)

[7] Reussner, R.H., Schmidt, H.W., Poernomo, I.: Reliability prediction for component-based software architectures. Journal of Systems and Software – Special Issue of Software Architecture – Engineering Quality Attributes **66** (2003) 241–252

[8] Smith, C.U., Williams, L.G.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley (2002)

[9] Firus, V., Becker, S., Happe, J.: Parametric Performance Contracts for QML-specified Software Components. In: Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA). Volume 141 of Electronic Notes in Theoretical Computer Science., ETAPS 2005 (2005) 73–90

[10] Bertolino, A., Mirandola, R.: CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. In: Proc. CBSE. Volume 3054 of LNCS., Springer (2004) 233–248

[11] Balsamo, S., Marzolla, M.: A Simulation-Based Approach to Software Performance Modeling. In: Proc. 9th ESEC, ACM Press (2003) 363–366

[12] Petriu, D.C., Shen, H.: Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In: Computer Performance Evaluation – Modelling Techniques and Tools. Volume 2324 of LNCS., Springer (2002) 159 – 177

[13] Grassi, V., Mirandola, R., Sabetta, A.: From Design to Analysis Models: a Kernel Language for Performance and Reliability Analysis of Component-based Systems. In: Proc. WOSP '05, New York, NY, USA, ACM Press (2005) 25–36

[14] Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press (1996)