Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs

Michael Kuperberg and Steffen Becker
{mkuper|sbecker}@ipd.uka.de
Institute for Program Structures and Data Organisation
Faculty of Informatics, University of Karlsruhe (TH), Germany

***Abstract*—Performance prediction of component-based software systems is needed for systematic evaluation of design decisions, but also when an application's execution system is changed. Often, the entire application cannot be benchmarked in advance on its new execution system due to high costs or because some required services cannot be provided there. In this case, performance of bytecode instructions or other atomic building blocks of components can be used for performance prediction. However, the performance of bytecode instructions depends not only on the execution system they use, but also on their parameters, which are not considered by most existing research. In this paper, we demonstrate that parameters cannot be ignored when considering Java bytecode. Consequently, we outline a suitable benchmarking approach and the accompanying challenges.**

I. Introduction

To meet requirements and expectations of users, modern software systems must be created with consideration of both functional and extra-functional properties. For extra-functional properties such as performance, early analysis and prediction reduce the risks of late and expensive redesign or refactoring if the required extra-functional properties are not satisfactory.

The performance (i.e., response time and throughput) of component-based software systems depends on several factors [1]:

a) the *architecture* of the software system, i.e. static structure of components and connections
b) the *implementation* of the components that comprise the software system
c) the *runtime usage* of the application (values of input parameters etc.) and
d) the *execution system* (hardware, operating system, virtual machine, middleware etc.) on which the application is run.

Making the influence of the execution system on performance explicit and quantifiable will help in different scenarios:

- **Redeployment of a component-based application in an execution system with different characteristics** (Fig. 1(a)): The execution system's characteristics will change when, for example, an operating system upgrade is conducted or when a more powerful server is bought. Assessing resulting performance changes *before* redeployment to compare benefit and costs beforehand is very reasonable.
- **Estimation of suitable execution system to fulfill changed performance targets for an existing software system ("Sizing")** (Fig. 1(b)): Changes in the usage profile (i.e., number of concurrent users, increased user activity, different input) of a business application may require adaptation of the application's performance that cannot be fulfilled with the original execution system. Performance prediction can be useful in choosing which execution system can fulfill the changed requirements.

The straightforward approach for predicting an application's performance on a new execution system would be to deploy the application there and then to test it, obtaining the resulting performance. While generally possible, testing requires installing the software on every concerned system and causes effort to provide the components implementing required services, an appropriate test workload and settings as well as further effort to take measurements etc.

To avoid the expensiveness and the inflexibility of the testing-based approach, the *building blocks* (such as bytecode instructions) which constitute component implementation can be used for performance prediction. Each building block's *performance* is obtained by benchmarking. The obtained performance is combined with the *frequency* of the building block, i.e. with the number of times the building block is executed during the execution of the component's services (cf. [2], [3]). In this paper, we target components that are compiled into Java bytecode and executed on a Java virtual machine, but the approach for .NET and other bytecode-oriented execution systems would be very similar.

The performance of such building blocks on an execution system can be captured in several ways. One possibility is to consider individual hardware and software resources that comprise the execution system and to evaluate the individual *resource consumptions* induced by the building block execution (for example, CPU time or memory usage). The results of these separate measurements must be integrated into a performance specification for the *entire* execution system. That is, the final result for the considered building block must be response time or another suitable metric.

Yet in general, such a bottom-up approach leads to a large

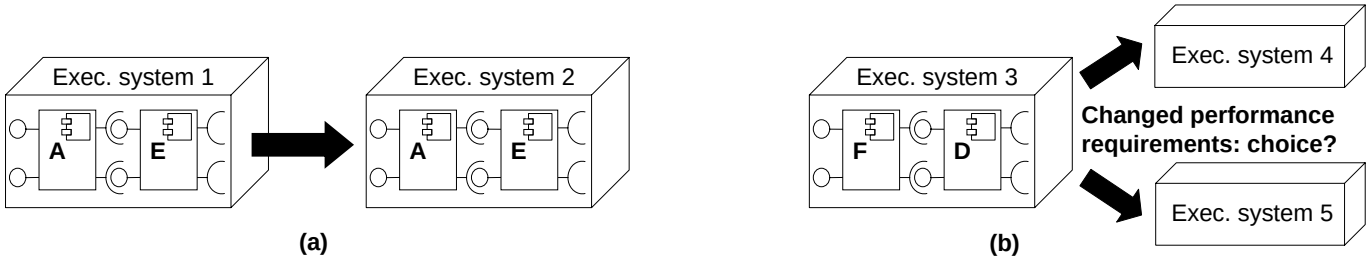

Fig. 1. Redeployment: scenarios for performance prediction

and complex model of the execution system. Additionally, the usage of the different resources of the execution system may lead to cross-influences. For example, memory-intensive computations may on some execution systems lead to unexpected hard disk usage due to swapping of virtual memory. Such cross-influences can impact several execution system resources and a detailed analysis of the resulting effects is required to evaluate the overall performance of a building block.

Hence, a simpler alternative is to consider the interaction of the components (i.e. their building blocks) with the entire execution system as interaction with one large black box, i.e. at a high abstraction level.

Furthermore, benchmarking-based performance prediction as utilised in this paper should also be faster than testing, as there is no need to run the considered component service for its entire wall-clock time duration. If the considered component service performs *external calls* to services of other components, their performance must be incorporated into the prediction. This can be done through simulation or analytical methods, and both should also be faster than testing.

Using bytecode instructions (or other building blocks) for predicting performance of software systems is not limited to component-based software. However, it appears suitable for components (especially given the number of commercial component architectures that rely on Java), and it can be applied to systems where the source code is not available (such as legacy systems). Once component services are annotated with performance descriptions (e.g. mean response times), model-based prediction methods such as KLAPER [4] or Palladio [5] can use them to predict the performance of a component-based software system that is reusing such components.

First approaches for utilising bytecode for performance prediction are described in [2], [3] and [6], but they ignore the parameters of bytecode instructions, assuming that parametric dependencies at bytecode level are not important. Hence, while each instruction's frequency is counted by observing the application at runtime, identification of the instruction parameters is not attempted and not recorded.

The contribution of this paper is a demonstration of the importance of bytecode instruction parameters and, correspondingly, a comprehensive bytecode benchmarking approach design that takes care of parametric dependencies. We outline the needed steps and present the resulting challenges that we plan to address in future work.

To provide the background for our work, we explain bytecode-level parameters and present our case study in section II. Building on the study, we discuss the details on benchmarking and performance prediction in section III-A. Challenges that must be addressed are summarised in section III-B. General limitations and assumptions of the resulting approach are described in section IV. In section V, we consider related work and compare it with our approach. Section VI concludes by describing future work.

II. Java Bytecode Instructions and their Parameters

A. Java Virtual Machine, Java Bytecode and Java API

Applications targeted for execution on the Java platform are compiled into *Java bytecode*, an intermediate language that is less machine-specific than pure native machine code. Java bytecode is executed on the *Java Virtual Machine* (JVM), which abstracts the specific details of the underlying software/hardware platform. The usage of bytecode makes compiled Java components portable across different platforms.

Through the standard *Java API*, application programmers have access to a number of libraries that are bundled with the JVM. These libraries implement frequently needed functionality, such as collections, inter-program communication etc. The functionality is accessed from Java components by calling the API methods and using API interfaces/classes.

From the component point of view, when a class that is part of a component makes an API call, this call could be considered as a call to another component. However, following the properties of components as described in Szyperski [7, p. 30], the JVM and the attached Java libraries should not be considered as a component. This is because the JVM is neither a unit of independent deployment (it cannot be deployed where other Java components are deployed, namely inside a Java virtual machine), but also because the JVM cannot be composed by a third party into another Java component.

Furthermore, components are composed by connecting their provides and requires interfaces, but direct use of the Java libraries means direct access of API classes' fields, and therefore is beyond the interfaces provided by the API. Therefore, in this paper, we consider calls to the Java API (that are done from the compiled bytecode) as part of the internal implementation of the component, and not as calls to another component.

B. Java Methods and Their Parameters

In bytecode, four instructions (`invokeinterface`, `invokespecial`, `invokestatic`, `invokevirtual`) are used to invoke *any* methods from bytecode. This means that these instructions are used both for API methods and for methods provided by the calling component itself or provided by other components. The method signatures are passed to `invoke*` as bytecode instruction parameters stored directly inside bytecode. Therefore, the performance of these four method-invoking instructions will depend on the actual method invoked, as well as on the parameters of that method.

Consequently, each combination of `invoke*`, method signature and method parameters potentially has its own performance. However, previous approaches to bytecode benchmarking did not consider bytecode instruction parameters to account for this fact (see section V). As a method's parameters are already on top of the JVM stack when the method-invoking bytecode instruction is called, the method parameters should be detected and used for performance prediction, which has to be considered during benchmark construction and execution.

Inside the JVM, a call to a single API method can result in many method calls and bytecode operations. Using a profiler, one can verify that for a single call to `System.out.println` method, many bytecode instructions but also several other API methods are invoked in a *call tree*. Some call trees can include native methods, so it is in general not possible to decompose a call tree into a set of bytecode instructions that does *not* include any method invocations. Hence, method invocations must be considered as building blocks in a bytecode-based performance prediction approach, and invocations of private and/or native methods that are not visible through the API must be considered as well.

One possibility to treat the resulting call tree of instructions *and* methods is to record each one of them and to use a benchmark to predict the performance of the call that is the root of the call tree. Obviously, this is not a favorable approach due to the resulting complexity. Additionally, propagation of measurement errors for the nodes of the call tree and other factors (e.g., need to circumvent Java security measures to benchmark private methods) favor a different approach, namely treating `System.out.println` and other API methods in an *atomic* way. Therefore, we prefer to remain at the highest abstraction level that is possible for components that are compiled to bytecode, and the Java API methods will not be decomposed any further in our approach.

Apart from method calls, parameters of other bytecode instruction can be expected to play an important role when working with collections. In the following section, we consider initialisation of collections and identify important parameter characterisations with respect to performance.

C. Influence of Bytecode Parameters on Performance of Instructions

For our case study, we have considered the initialisation of two data structure type: arrays and `ArrayLists`. Arrays in Java have a fixed size that must be specified at initialisation time while an `ArrayList` can grow dynamically and no initial size must be specified for it at initialisation time (although the Java API documentation specifies that the default initial capacity will be 10).

Initialisation of data structures includes allocating memory, and it is therefore logical to expect performance of the initialisation to depend on factors such as

a) collection's initial size
b) collection type
c) the type of the collection's elements (including the difference between value types and reference types) and
d) the size of each collection element (for example 8 bytes for a `double` vs. 4 bytes for an `int`)

To validate whether these dependencies really do exist for bytecode instructions, we have created arrays by using appropriate bytecode instructions such as `newarray` and `anewarray`. We then compared the results with initialisation of `ArrayLists`, which corresponds to a method call inside bytecode. The results in Fig. 2 show for each initial collection size (on the x-axis) the median of 100 measurements. In each measurement, 400 initialisations took place. The execution system was MS Windows Server 2003 SP1 on a single-core AMD Sempron™3100+ (1.80 GHz, 1 GB RAM) with Sun Java 1.5.0_11 and standard settings.

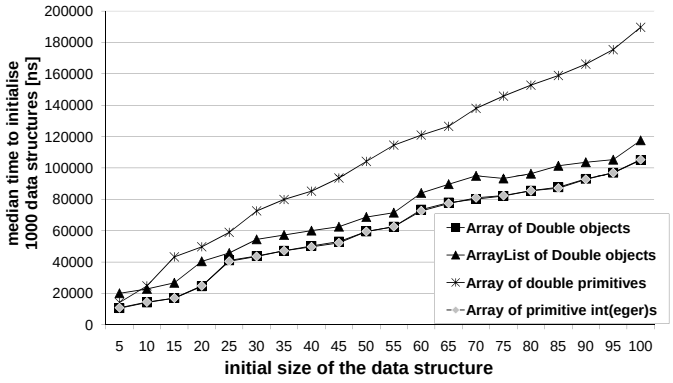

Fig. 2. Bytecode instructions' parametric dependencies in data structure initialisation

In fact, the measured values grow almost linearily with the initial collection size and all four expected parametric dependencies are exposed. This contradicts most existing approaches, which do not identify these impacts on bytecode instructions; at best, as in [3], only some few Java API methods where linear dependency is expected are measured and their performance is specified on a per-element basis.

III. Benchmarking of Java Bytecode and API

In this section, we present our approach for bytecode-based performance prediction of components and outline the consequences of considering the instruction parameters. The description of the methodology is followed by the discussion of the resulting challenges in section III-B.

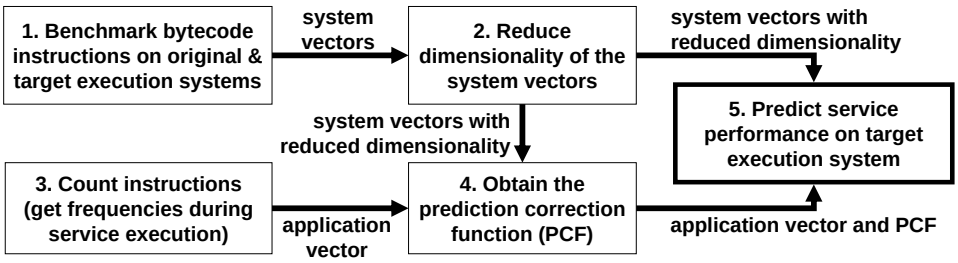

Fig. 3. Steps to predict performance using bytecode benchmarking

A. Methodology

We follow the notation in [2], where frequency of individual bytecode instructions is expressed by the *application vector* $\vec{P} = (p_1 p_2 p_3 \dots p_n)^T$. In $\vec{P}$, n denotes the number of available instructions and $p_i \geq 0$ is the execution frequency of instruction i. Correspondingly, performance of instructions is collected in the the n-dimensional *system vector* $\vec{S} = (s_1 s_2 s_3 \dots s_n)^T$. The performance prediction is obtained by $\vec{P} \cdot \vec{S}$ under the assumption that with the same usage (i.e., input parameters etc.), $\vec{P}$ remains the same accoss execution systems and component service executions. In Fig. 3, the steps of the proposed methodology are outlined.

Step 1. Benchmarking Bytecode
As we need to benchmark the bytecode instructions individually, a suite of microbenchmarks must be constructed. The microbenchmarks will fall into two groups: (a) the API methods that are called when instructions such as `invokevirtual` are executed and (b) the instructions that do not call the Java API.

When parameters have to be generated, attention must be paid to their representativeness and several microbenchmark runs must be performed if necessary (e.g. to discover linear dependencies). Parametric dependencies must be reflected by elements of $\vec{S}$. Considering the fine-granular nature of bytecode instructions, the microbenchmarks should measure a sufficient number of instructions with regard to timer resolution.

This will be problematic when it is not possible to separate the preparation of parameters passed over the JVM stack from the execution of the instruction itself. The performance of parameter preparation must then be benchmarked and subtracted from such results.

The benchmarking step has to be executed once for each execution system, and changed settings of the execution system require a repetition of this step. Also, the microbenchmarks should be executed several times to be able to eliminate outliers and to obtain statistically significant results.

Step 2. Reducing the System Vector's Dimensionality
The benchmarking step produces a very large system vector due to instruction parameters, the API methods and their parameters. We have envisioned two potential ways to prevent the size of the system vector $\vec{S}$ from becoming prohibitively expensive to work with: (a) usage of parametric descriptions in system and application vectors, for example duration per character in string conversion and (b) clustering of system vector's elements to group measurements with similar values. These methods could be combined as well.

Step 3. Obtaining the application vector
From the execution of the component's service on the original execution system, we obtain the frequencies p_i for the application vector $\vec{P}$. Since we assume that the component will be run on the target execution system with the same service parameters (i.e., runtime usage profile will remain stable), we assume that $\vec{P}$ will be valid there as well. For obtaining $\vec{P}$, we aim at bytecode instrumentation which has to be carried out on the original execution system only.

Step 4. Obtaining the prediction correction function
Hotspot compilation, JVM optimizations and other techniques might distort the prediction. To cope with this, we plan to introduce a *prediction correction function* (PCF). PCF could be obtained, for example, by an algorithm that uses machine learning to quantify the difference between prediction and reality (i.e., measurements). To learn, the algorithm takes a small fixed set of generic component services and compares prediction and measurement. From this comparison, a correcting function is derived that could be applied to the performance prediction of other component services. Of course, the application of PCF requires a representative learning set and a learnable correction function.

Step 5. Predicting the performance
The prediction is computed by $\vec{P} \cdot \vec{S}$, i.e. by $\sum_{i=1}^{|\vec{P}|} p_i \cdot s_i$. For this, we assume that elements of the system vector $\vec{S}$ contain only simple numerical values. However, API methods that sort a collection perform differently for *same* size of the collection because the sorting effort depends on the orderliness of the collection. If probability distributions are used to express the resulting variation of the performance behaviour, [8] describes the computation of $\vec{P} \cdot \vec{S}$ using convolution.

B. Challenges

Of course, not all instructions are as expensive as collection initialisation or API calls. Hence, some simple instructions may be orders of magnitude faster and/or parameter-independent. We aim at identifying the computationally expensive bytecode instructions and at fully benchmarking *only* them and API methods, while approximating the performance of the computationally "cheap" instructions.

For benchmarking the Java API, additionally to method parameter generation and parameter range coverage (representability), we must deal with runtime exception handling, inheritance/polymorphism, invocation target instantiation (for non-static methods) and some other challenges.

An implementation of the JVM (i.e., mapping of Java bytecode instructions and Java APIs to machine code) is not specified and varies across vendors. It is possible that an instruction's performance will be different for distinct JVMs that executed on the *same* underlying hardware/operating system. We can count for this by considering the JVM as a part of the execution system and then specify the bytecode instruction performance for each execution system.

Modern virtual machines start the execution of bytecode in interpretation mode while trying to detect performance hotspots. These hotspots are further compiled into native machine code on the fly if the result is an overall performance increase. The standard Sun Microsystems JVM features the so-called *just-in-time compilation* (JIT [9]) capability, and for long-running business applications that our research targets, such compilation will be relevant. Thus, identifying and quantifying hotspot compilation is a challenge for our research.

Pipelining [10] is a technique used by CPUs to increase execution speed. For our methodology, it means that a sequence of bytecode instructions may execute in less time than the sum of individual execution times, thus distorting the prediction. A study must be performed to evaluate whether pipelining is important at bytecode (JVM) level.

Background memory (de)allocation, i.e. the so-called Garbage Collection (GC), is another important factor that depends on the implementation of a JVM is which is visible through irregular or periodical events halting or slowing the program execution. Such interruptions and delays can distort the prediction, and we will try to understand the effects of GC on long-running business applications by inspecting its duration in different modes, as for example exposed by usage of `-verbose:gc` parameter of Sun Microsystems' JVM.

Granularity of the benchmark atoms is important both for complexity, accuracy and precision. Analysis of instruction tuples in [11] has shown that some pairs are very frequent while others do not occur at all; [12] uses sequential bytecode blocks instead of individual bytecodes. For our research, we may evalute the use of individual non-API bytecode instructions to separate parameter preparation for API calls from the actual API calls' performance.

Error and error propagation are general issues in benchmarking and prediction; additionally, we can expect simple bytecode instructions to be very fast. Consequently, special attention must be paid to benchmark such instructions with respect to timer resolution, confidence intervals, outliers and similar aspects.

IV. Assumptions and Limitations

For an initial implementation of the proposed approach, some helpful assumptions must be made which limit the complexity of the undertaking. For instance, a virtual machine can include different optimisations of bytecode execution, for example by replacing some instructions with semantically equivalent but faster ones in vendor-specific ways. Such behaviour would distort the performance prediction by altering the application vector $\vec{P}$. Therefore, we assume that no such optimisations take place at runtime.

A component service implementation can include sections that may be executed parallely (for example, by spawning a new thread). Counting the bytecode instructions and combining them with the results of the bytecode benchmark as our approach proposes will then result in wrong prediction results if the service is executed on a system that offers hardware-supported parallelism. For now, our approach assumes that the execution of the service is not parallelisable.

The state of the components may impact their performance. As this is hard to measure and hard to predict, separate provisions would have to be developed for this. For now, we aim to abstract from the state of the individual component and to investigate it in future work.

The performance of the component's required services must be included into prediction. We assume that such external calls can be detected and integrated into prediction, for example by using Sevice Effect Specifications (SEFFs, [13]).

V. Related Work

Sitaraman et al. [14] discuss parametrics of both space and time behaviour of collection initialisation from the implementation viewpoint, while our work focuses on capturing the resulting dependencies during measurement/benchmarking and during prediction.

In the Java Resource Accounting Framework (JRAF [15]), Binder and Hulaas use bytecode instructions counting for the estimation of CPU consumption. However, all bytecodes are treated equally, parameters of individual instructions (incl. API method names) are ignored, which contradicts our case study findings. In JRAF, it is assumed that invocations of API methods break down to invocations of elementary bytecode instructions in a platform-independent way, while we consider API calls as atomic.

In HBench:Java [3], Zhang and Seltzer build the system vector by separating high-level JVM "components" such as system classes (i.e., API implementation), memory management, JIT and control flow/primitive bytecode execution. However, the evaluation was performed by selecting and benchmarking only 30 particularly expensive API methods (some of them were found to show linear dependency on one parameter), and no absolute comparison between measured and predicted performance is provided. Therefore, it remains questionable

whether the such "components" can be combined into a suitable and application-independent prediction. Furthermore, it is not clear how an application vector can be obtained with respect to the JVM "components". For our approach, we aim to consider the JVM as a black box and to study execution of bytecode instructions.

In [2], Meyerhöfer and Lauterwald want to measure the application vector by using interceptors in an application server. Interceptors then instrument the Java classes when they are loaded and the executed bytecode instructions are counted. Although API methods are mentioned in [2] as a potential extension for that approach, method parameters are not considered and it is not discussed how the substantially larger number of the building blocks can be handled. To deal with JIT effects, the authors suggest to maintain two performance measurements per instruction (for interpreted and JIT-compiled modes). However, it is not described how such measurements can be obtained.

VI. Conclusions

This paper presents a detailed description of a substantially refined performance prediction methodology. This methodology relies on bytecode instruction benchmarking and learns from the execution of existing component services to enhance the performance prediction on new execution systems. A case study is presented that shows the importance of parametric dependencies for API methods and other bytecode instructions. We discuss the inclusion of calls to the API into bytecode benchmarking, since API calls are a very important part of bytecode execution.

Other important factors influencing both benchmarking and prediction, such as respect of hotspot compilation and garbage collection are also discussed. Several suitable scenarios for the usage of the prediction algorithm are described and the challenges as well as assumptions and limitations are presented. We also mention ways to handle the large size of benchmark results that is caused by analysis of parametric dependencies and by API benchmarking.

The proposed approach can be beneficial for developers and architects who wish to evaluate the performance of components. For this, the bytecode benchmark can be intergrated into component-oriented modeling languages, such as the Palladio Component Model [5]. This can be done by constructing the behavioural specifications of the component services and annotating them with performance predictions obtained through the methodology proposed in this paper.

Future work will start with constructing the API benchmark and by investigating whether benchmarking of a subset of remaining bytecode instructions is sufficient for acceptable precision of prediction. After the validation of the proposed prediction methodology, we plan to extend it to allow probabilistic descriptions of benchmark results. Ultimately, we plan to integrate our approach into the modeling and prediction tooling of the Palladio Component Model [5].

References

[1] S. Becker and R. Reussner, "The Impact of Software Component Adaptation on Quality of Service Properties," *L'Objet* -, vol. 12, no. 1, pp. 105–125, 2006.

[2] M. Meyerhöfer and F. Lauterwald, "Towards Platform-Independent Component Measurement," in *Tenth International Workshop on Component-Oriented Programming*, W. Weck, J. Bosch, R. Reussner, and C. Szyperski, Eds., 2005.

[3] X. Zhang and M. Seltzer, "HBench:Java: an application-specific benchmarking framework for Java virtual machines," in *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*. New York, NY, USA: ACM Press, 2000, pp. 62–70.

[4] V. Grassi, R. Mirandola, and A. Sabetta, "From Design to Analysis Models: a Kernel Language for Performance and Reliability Analysis of Component-based Systems," in *WOSP '05: Proceedings of the 5th international workshop on Software and performance*. New York, NY, USA: ACM Press, 2005, pp. 25–36.

[5] S. Becker, H. Koziolek, and R. Reussner, "Model-based Performance Prediction with the Palladio Component Model," in *Workshop on Software and Performance (WOSP2007)*. ACM SIGSOFT, 2007.

[6] A. Camesi, J. Hulaas, and W. Binder, "Continuous Bytecode Instruction Counting for CPU Consumption Estimation," in *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006)*, pp. 19–30.

[7] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. ACM Press, Addison-Wesley, Reading, MA, USA, 1998.

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

[9] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley, 1999.

[10] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, 2003.

[11] J. Lambert and J. F. Power, "An Analysis of Basic Blocks within SPECjvm98 Applications," Department of Computer Science, National University of Ireland, Maynooth, Co. Kidare, Ireland, Tech. Rep. NUIM-CS-TR-2005-15, 2005.

[12] P. Wong, "Bytecode Monitoring of Java Programs," *BSc Project Report, University of Warwick*, 2003.

[13] R. H. Reussner, H. W. Schmidt, and I. Poernomo, "Reliability Prediction for Component-Based Software Architectures," *Journal of Systems and Software – Special Issue of Software Architecture – Engineering Quality Attributes*, vol. 66, no. 3, pp. 241–252, 2003.

[14] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy, "Performance Specification of Software Components," in *SSR*, 2001, pp. 3–10.

[15] W. Binder and J. Hulaas, "Using Bytecode Instruction Counting as Portable CPU Consumption Metric," *Electr. Notes Theor. Comput. Sci.*, vol. 153, no. 2, pp. 57–77, 2006.