Towards Parameter-aware Benchmarking of Bytecode
and API for Predicting Component Performance

Michael Kuperberg and Steffen Becker
{mkuper|sbecker}@ipd.uka.de

Palladio Research Group at the Chair for Software Design and Quality*
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.

1 Introduction

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

The performance (i.e., response time and throughput) of component-based software systems depends on several factors [1]: (a) the implementation of its components (b) the architecture of the application, i.e. static structure of components and connections (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.

* This work is supported by the German Research Foundation (DFG), grant RE 1674/1-2
The straightforward approach for predicting the 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 required services, the right 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 component’s building blocks (such as bytecode instructions) can be used. Each building block’s performance is considered separately from its frequency during the execution of the component’s services \cite{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.

Existing approaches ignore the parameters of bytecode instructions, assuming that parametric dependencies 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 bytecode instruction parameters’ importance 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 2. Building on the study, we discuss the details on benchmarking and performance prediction in section 3.1. Challenges that must be addressed are summarised in section 3.2. General limitations and assumptions of the resulting approach are described in section 4. In section 5, we consider related work and compare it with our approach. Section 6 concludes by describing future work.

2 Java Bytecode Instructions and their Parameters

2.1 Foundations

Java bytecode instructions are executed on the stack-based Java Virtual Machine (JVM). Some instructions expect their parameters directly on the stack: for example, \texttt{iadd} adds two integers that it expects on top of the stack at runtime. For other instructions, parameters are stored inside compiled component bytecode and are not passed through the stack, while other instructions make use of both ways. This must be considered when bytecode instruction parameters have to be recorded for use during prediction.

Four instructions (\texttt{invokeinterface}, \texttt{-special}, \texttt{-static} and \texttt{-virtual}) are used to invoke methods whose signatures are passed as parameters in bytecode. These instructions can be used to invoke the Java API methods and we consider the
calls to the Java API as part of bytecode and not as calls to other components. Therefore, each \texttt{invoke*method name} combination has its own performance and must be benchmarked separately. The performance of each API method itself depends on the API method’s parameters. Usually, a method’s parameters are already on top of the JVM stack when the method-invoking bytecode instruction is called. This has to be considered during benchmark construction and execution.

### 2.2 Influence of Bytecode Parameters on Performance of Instructions

For our case study, we consider the creation of data structures such as arrays and \texttt{ArrayList}s. It is intuitive to expect performance of such operations to depend on (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 (i.e., 8 bytes for \texttt{doubles} vs. 4 bytes for \texttt{ints}).

To validate whether these dependencies exist for bytecode instructions, we have created arrays by using appropriate bytecode instructions. We then compared the results with creation of \texttt{ArrayList}s, 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\textsuperscript{TM} 3100+ (1.80 GHz, 1 GB RAM) with Sun Java 1.5.0_11 and standard settings.

As expected, the measured values grow almost linearly 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.

![Fig. 2. Bytecode instructions’ parametric dependencies in data structure initialisation](image-url)
3 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 3.2.

3.1 Methodology

We follow the notation in [2], where frequency of individual bytecode instructions is expressed by the application vector \( P = (p_1p_2p_3 \ldots p_n)^T \). In \( P \), \( n \) 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 \( n \)-dimensional system vector \( S = (s_1s_2s_3 \ldots s_n)^T \). The performance prediction is obtained by \( P \cdot S \) under the assumption that with the same usage (i.e., input parameters etc.), \( P \) remains the same across 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 \( S \).

A compiler should not be used for generating the microbenchmarks, as full control over the generated bytecodes is needed and some needed bytecode instructions arrangements that are correct cannot be generated by compiling Java source code. 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 Dimensionality of the System Vector  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 $S$ 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 $P$. 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 $P$ will be valid there as well. For obtaining $P$, 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 $P \cdot S$, i.e. by $\sum_{i=1}^{\mid P \mid} p_i \cdot s_i$. For this, we assume that elements of the system vector $S$ 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, [4] describes the computation of $P \cdot S$ using convolution.

3.2 Challenges  The implementation (i.e., mapping of bytecode API to machine code) of the Java Virtual Machine (JVM) is not specified and varies across vendors. This makes the performance of bytecode even harder to predict, since an instruction’s performance will be different for distinct JVMs that executed on the same underlying hardware/operating system.

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, and it may be simpler to identify the computationally expensive bytecode instructions and to benchmark only them and API methods, while approximating the performance of the “cheap” instructions.

An API method may be implemented in bytecode, but platform dependent-features such as file system access will require usage of native (non-Java) methods at some point of the implementation of many API methods. Therefore, it is not possible to break up every API method into elementary Java bytecode instructions. For benchmarking the
Java API, method parameter generation/instantiation, parameter range coverage (representability), runtime exception handling, inheritance/polymorphism, invocation target instantiation (for non-static methods) and many other challenges must also be addressed in addition to parameter generation for bytecode instructions.

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 [5]) 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 [6] 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 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 [7] has shown that some pairs are very frequent while others do not occur at all; [8] uses sequential bytecode blocks instead of individual bytecodes. For our research, we may evaluate 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.

4 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 \( P \). 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, [9]).

5 Related Work

Sitaraman et al. [10] 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 [11]), 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 H Bench: 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”.

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.

6 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 integrated into component-oriented modeling languages, such as the Palladio Component Model [12]. 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 [12].

References