Performance Prediction for Black-Box Components Using Reengineered Parametric Behaviour Models

Michael Kuperberg, Klaus Krogmann, and Ralf Reussner

Chair for Software Design and Quality, University of Karlsruhe, Germany
{mkuper,krogmann,reussner}@ipd.uka.de

Abstract. In component-based software engineering, the response time of an entire application is often predicted from the execution durations of individual component services. However, these execution durations are specific for an execution platform (i.e. its resources such as CPU) and for a usage profile. Reusing an existing component on different execution platforms up to now required repeated measurements of the concerned components for each relevant combination of execution platform and usage profile, leading to high effort. This paper presents a novel integrated approach that overcomes these limitations by reconstructing behaviour models with platform-independent resource demands of bytecode components. The reconstructed models are parameterised over input parameter values. Using platform-specific results of bytecode benchmarking, our approach is able to translate the platform-independent resource demands into predictions for execution durations on a certain platform. We validate our approach by predicting the performance of a file sharing application.

1 Introduction

To meet user requirements, software must be created with consideration of both functional and extra-functional properties. For extra-functional properties such as performance (i.e., response time and throughput), early analysis and prediction reduce the risks of late and expensive redesign or refactoring in case the extra-functional requirements are not met. Performance of component-based applications is predicted on the basis of performance of underlying components.

The performance of component-based applications depends on several factors [2]:

a) the *architecture* of the software system, i.e. the static "component assembly"
b) the *implementation* of the components that comprise the software system
c) the *runtime usage context* of the application (values of input parameters etc.) and
d) the *execution platform* (hardware, operating system, virtual machine, etc.)

Conventional performance prediction methodologies do not consider all four factors separately [4,24] or limit themselves to real-time/embedded scenarios [7]. To make the influence of these factors on performance explicit (or even quantifiable), these approaches would need to re-benchmark each component, or even the entire application each time one of the four factors changes. Instead, separating these factors is beneficial for efficient performance prediction in the following scenarios:

M.R.V. Chaudron, C. Szyperski, and R. Reussner (Eds.): CBSE 2008, LNCS 5282, pp. 48–63, 2008.

- **Redeployment** of an application to an execution platform with different characteristics, i.e. into a new *deployment context*.
- **Sizing** of suitable execution platform to fulfill changed performance targets for an existing software system, for example due to changes in the *usage context* (i.e., number of concurrent users, increased user activity, different input).
- **Reuse** of a single component in another architecture or **architectural changes** in an existing software system, i.e. changes in the *assembly context* of a component.

In this paper, we present a novel integrated approach that makes these factors explicit and quantifiable.

Our first contribution is a validated reverse engineering approach that uses machine learning (genetic programming) on runtime monitoring data for creating *platform-independent* behaviour models of black-box components. These models are parameterised over usage context and deployment context.

Our second contribution is the performance prediction for these behaviour models, which predicts *platform-specific* execution durations on the basis of bytecode benchmarking results, allowing performance prediction for components and also entire component-based applications. Re-benchmarking an application for all relevant combinations of usage and deployment contexts is thus not necessary anymore.

We validate our approach by reconstructing a performance prediction model for a file sharing application and subsequently predict the execution duration of the application, depending on usage context and deployment context. To the best of our knowledge, this is the first validated bytecode-based performance prediction approach. We describe how our approach maintains the black-box property of components by working without their source code and without needing the full inner details of their algorithms and implementations.

The paper is structured as follows: in Section 2, we describe related work. In Section 3, an overview of our approach is given and its implementation is described. Using a case study of a file-sharing application, we evaluate our approach in Section 4. The limitations and assumptions of the presented approach and its implementation are provided in Section 5, before the paper concludes in Section 6.

2 Related Work

This paper is related to reverse engineering of performance models, bytecode-based performance prediction, and search-based software engineering [12].

Reverse engineering of performance models using traces is performed by Hrischuk et al. [14] in the scope of "Trace-Based Load Characterisation (TLC)". In practice, such traces are difficult to obtain and require costly graph transformation before use. The target model of TLC is not component-based.

Using trace data to determine the "effective" architecture of a software system is done by Israr et al. in [16]. Using pattern matching, this approach can differentiate between asynchronous, blocking synchronous, and forwarding communication. Similar to our approach, Israr et al. support components and have no explicit control flow, yet they do not support inter-component data flow and do not support internal parallelism in component execution as opposed to the approach presented in this paper. As in TLC, Israr et al. use Layered Queueing Networks (LQNs) as the target performance model.

Regression splines are used by Courtois et al. in [9] to recognise input parameter dependencies in code. Their iterative approach requires no source code analysis and handles multiple dimensions of input, as does the approach described by us. However, the output of the approach in [9] are polynomial functions that approximate the behaviour of code, but which are not helpful in capturing discontinuities in component behaviour. The approach is fully automated, but assumes fixed external dependencies of software modules and fixed hardware.

Search based approaches such as simulated annealing, genetic algorithms, and genetic programming have been widely used in software engineering [12]. However, these approaches have not been applied to reverse engineering, but to problems like finding concept boundaries, software modularization, or testing.

Daikon by Ernst et al. [11] focusses on detection of invariants from running programs, while our approach aims at detecting parametric propagation and parametric dependencies of runtime behaviour w.r.t performance abstractions. Analysis is in both approaches supported by genetic algorithms.

Performance prediction on the basis of bytecode benchmarking has been proposed by several researchers [13,23,25], but no working approach has been presented and no libraries or tools are available. Validation has been attempted in [25], but it was restricted to very few Java API methods, and the actual bytecode instructions were neither analysed nor benchmarked. In [18], bytecode-based performance prediction that explicitly distinguishes between method invocations and other bytecode instructions has been proposed.

Obtaining execution counts of bytecode instructions is needed for bytecode-based performance prediction, and has been addressed by researchers (e.g. [5], [19]) as well as in commercial tools (e.g. in profilers, such as Intel VTune [10]). ByCounter [19] counts bytecode instructions and method invocations *individually* and it is portable, light-weight, and transparent to the application. ByCounter works for black-box components and its Java implementation will be used in this paper.

Execution durations of individual bytecode instructions have been studied independently from performance prediction by Lambert and Brown in [20], however, their approach to *instruction timing* was applied only to a subset of the Java instruction set, and has not been validated for predicting the performance of a real application. In the Java Resource Accounting Framework [6], performance of all bytecodes is assumed to be equal and parameters of individual instructions (incl. names of invoked methods) are ignored, which is not realistic. Hu et al. derive worst-case execution time of Java bytecode in [15], but their work is limited to real-time JVMs.

Cost analysis of bytecode-based programs is presented by Albert et al. in [1], but neither bytecode benchmarks not actual realistic performance values can be obtained, since the performance is assumed to be equal for all bytecode instructions.

3 Reverse Engineering and Performance Prediction

An overview of our approach is summarised in Fig. 1, and Sections 3.1-3.6 provide detailed descriptions of its steps. Our approach consists of two parts, separated in Fig. 1 by the dashed line. The first (upper, light) part **A** produces behavioural performance

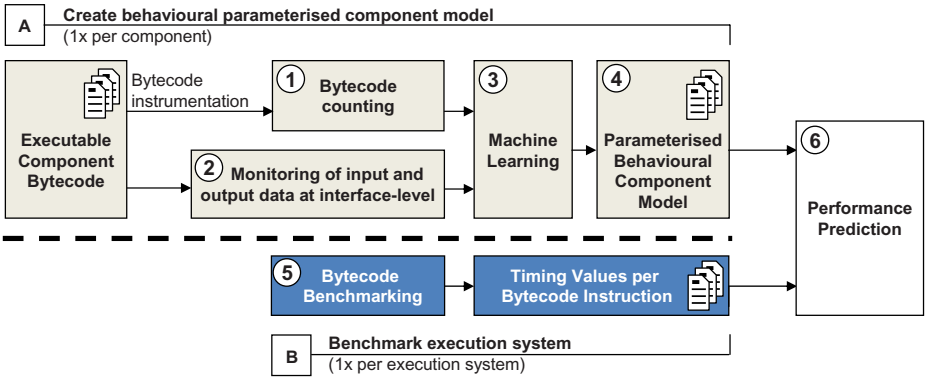

Fig. 1. Overview on the approach

models for black-box components. These models are platform-independent because they do not contain platform-specific timing values. Such timing values are produced by the second part of our approach (cf. (lower, darker) part **B** of Fig. 1), which uses bytecode benchmarking as described in Section 3.5.

Part **A** of our approach works on components and for component-based applications, for which only binary code and no source code may be available. In Step 1, the considered component is executed in a testbed (e.g. derived from running representative applications) and executed bytecode instructions and method invocations are counted. In Step 2 (which can be executed concurrently with Step 1), inputs and outputs of the considered component are monitored at the interface level.

Machine learning in Step 3 then (i) estimates the parametric dependencies between input data and the number of executions for each bytecode instruction and (ii) finds data and control flow dependencies between provided and required service, which is important for components since output of one service will be the input of another one.

Step 4 uses results from machine learning in Step 3 and constructs a behavioural component model which is parameterised over usage context (input data), external services and also over the execution platform. Such a model includes how input data is populated through an architecture, a specification how often external services are called with which parameters, and how an execution platform is utilised.

Part **A** of our approach is executed once per component-based application.

In Step 5 (part **B**), bytecode instructions are benchmarked on the target execution platform to gain timing values for individual bytecode instructions (e.g. "`IADD` takes 2.3 ns"). These timing values are specific for the used target platform, and the benchmarking step is totally independent of the previous steps of our approach. Hence, benchmarking must be executed only once per each execution platform for which the performance prediction is to be made.

The results of part **A** and part **B** are inputs for the performance prediction (Step 6), which combines performance model and execution platform performance measures to predict the actual (platform-specific) execution duration of an application executed on that platform.

The separation of application performance model and execution platform performance model allows to estimate the performance of an application on an execution platform without actually deploying the application on that platform, which means that in practice, one can avoid buying expensive hardware (given a hardware vendor providing benchmarking results) or also avoid costly setup and configuration of a complex software application.

3.1 Counting of Bytecode Instructions and Method Invocations

To obtain runtime counts of executed bytecode instructions (cf. Fig. 1, Step 1), we use the ByCounter tool, which is described in detail in [19] and works by instrumenting the bytecode of an executable component. We count all bytecode instructions individually, and also count method invocations in bytecode, such as calls to API methods.

The instrumentation with the required counting infrastructure is *transparent*, i.e. the functional properties of the application are not affected by counting and all method signatures and class structures are preserved. Also, instrumentation runs fully automated, and the source code of the application is not needed.

At runtime, the inserted counting infrastructure runs in parallel with the execution of the actual component business logic, and does not interfere with it. The instrumentation-caused counting overhead of ByCounter is acceptable and is lower than the overhead of conventional Java profilers. As said before, the instruction timings (i.e. execution durations of individual instruction types) are not provided by the counting step, but by bytecode benchmarking in Step 5.

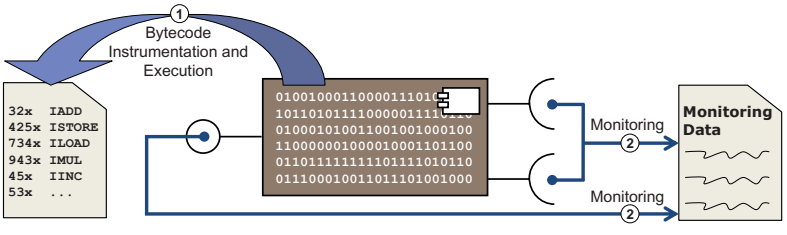

Fig. 2. Data extraction from executed black-box components

The counting results (cf. Fig. 2, Step 1) are counts for each bytecode instruction and found method signature, and they are specific for the given usage context. The counting is repeated with different usage contexts of the considered component service, but on the same execution platform. Counting results are saved individually for each usage context and later serve as data on which machine learning is run (cf. Fig. 1, Step 3).

3.2 Data Gathering from Running Code

To capture the parametric dependencies between the application input and output, our approach monitors at the level of component interfaces (cf. Fig. 1, Step 2). We gather runtime information about component behaviour by executing the component in a testbed or executing the entire application (cf. Fig. 2, Step 2).

To obtain representative data, the execution of the monitored component services must be repeated for a set of representative inputs to the application (recent overview on test data generation can be found in [21]). The datasets obtained from monitoring serve as the input for the machine learning (Fig. 1, Step 3) to learn the parametric dependencies between input and output.

For each component service call (provided or required), our tool monitors the input parameter values of each component service call and the properties of the data that is returned by that service call. Monitored data properties are:

- for primitive types (i.e. `int`, `float` etc.): their actual values
- for all *one*-dimensional arrays (e.g. `int[]`, `String[]`), `Collection`, or `Map` types: the number of their elements
- for one-dimensional arrays of primitive type (e.g. `int[]`), also aggregated data, such as number of occurrences of values in an array (e.g. the number of '0's and '1's in an `int[]`)
- for a *multi*-dimensional array (e.g. `String[][]`): its size, plus results of individual recording of each included array (as described above)

For each *provided* service, we additionally monitor which required services are called by it how often and with which parameters. The described data monitoring and recording can be applied to component interfaces without a-priori knowledge about their semantics, and without inspecting the internals of black-box components. Supporting and monitoring complex or self-defined types (e.g. objects, structs) requires domain expert knowledge to identify important properties of these data types. Still, generic data types are used very often, and our approach can handle these cases automatically.

3.3 Machine Learning for Recognition of Parametric Dependencies

Our approach utilises machine learning for estimating the bytecode counts on the basis of input data and for recovering functional dependencies in the monitored data. We use the Java Genetic Algorithm Package JGAP [22] to support machine learning (a general introduction for genetic programming, a special case of genetic algorithms, can be found in [17]). For our approach, we combine genes representing mathematical functions to express more complex dependencies. Simple approaches like linear regression could be applied as well, but cannot handle non-continuous functions or produce little readable approximations by polynomials.

For every gathered input data point (e.g. size of an input array, or value of a primitive type) a gene representing that parameter in the resulting model is introduced. In addition to default JGAP genes (e.g. mathematical operations for power, multiplication, addition, constants), we introduced new genes to support non-continuous behaviour (e.g. jumps caused by "if-then-else") as JGAP allows defining of additional genes.

Learning Counts of Bytecode Instructions and Method Invocations
Genetic programming tries to find the best estimation of functions of bytecode counts over input data. If an algorithm uses less `ILOAD` instructions for a 1 KB input file than for a 100 KB file, the dependency between input file size and the number of `ILOAD` instructions would be learned.

Our approach applies genetic programming for each used bytecode instruction. A simple example of the resulting estimation for the `ILOAD` instruction is $IF(filesize > 1024)\ THEN\ (filesize \cdot 1.4 + 300)\ ELSE\ (24000)$). For bytecode instructions and method invocation counts, learning such functions produces more helpful results as mere average counts, because non-linear dependencies can be described appropriately, and also because these results are not specific to one execution platform.

Learning Functional Dependencies between Provided and Required Services
Genetic programming is also applied for discovering functional dependencies between input and output data monitored at the component interface level. Informal examples of such dependencies are "a required service is executed for every element of an input array", "a required service is only executed if a certain threshold is passed" (data dependent control flow), or "the size of files passed to a required component service is 0.7x the size of an input file" (data flow).

To recover such dependencies from monitoring data, genetic programming builds chromosomes from its genes to express a function matching the monitored data as much as possible. The deviation between learned function and monitored data is used as "fitness function" during learning. Thereby, genetic programming is selecting appropriate input values and rejecting others, not relevant for the functional dependency. Finally, the resulting function is an approximation of a component's internal control and data flow, where each dependency is represented by an own chromosome.

3.4 Parameterised Model of Component Behaviour

The target model (named "Parameterised Behavioural Component Model" in Fig. 1) is an instance of the Palladio Component Model [3][1]. The model instance has a representation for the static structure elements (software components, composition and wiring; not described here) and a behavioural model for each provided service of a component (an example is shown in Fig. 3).

A component service's behaviour model consists of *internal actions* (i.e. algorithms) and *external calls* (to services of other components). For internal actions (cf. left box in Fig. 3), reverse-engineered annotations for each bytecode instruction specify how often that instruction is executed at runtime, depending on component service's input parameters. The parameterised counts that form these annotations are platform-independent and do not contain platform-specific timing values.

For external calls (e.g. `add` and `store` in Fig. 3), the model includes dependencies between component service input and external call parameters, with one formula per input parameter of an external call (e.g. `a = input1 * 2` in Fig. 3). Also, the number of calls to each required (external) service is annotated using parameterisation over input data (cf. "Number of loops" grey box in 3).

3.5 Benchmarking Java Bytecode Instructions and Method Invocations

For performance prediction, platform-specific timings of all bytecode instructions and all executed API methods must be obtained, since the reverse engineered model from

[1] See http://www.palladio-approach.net

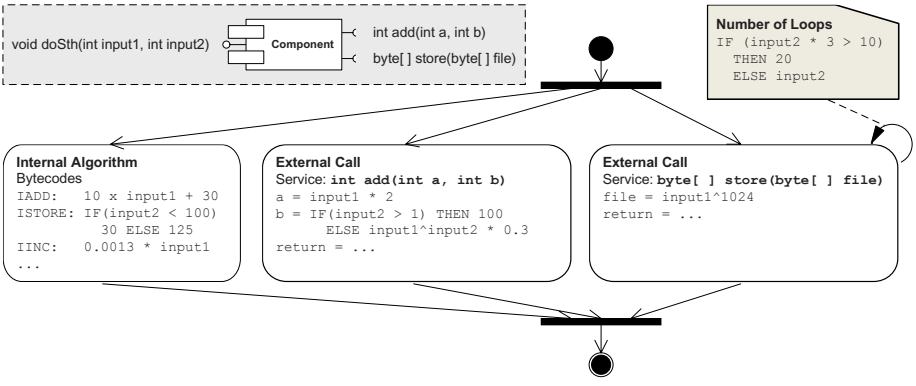

Fig. 3. Behavioural model of the provided service `void doSth(int input1, int input2)`

part **A** in Fig. 1 only contains their (estimated) counts. As timings for bytecode instructions and API methods are not provided by the execution platform and no appropriate approach exists to obtain them (cf. Section 2), we have implemented our own benchmark suite, which is an essential contribution of this paper.

We illustrate our approach by first considering the example of the Java bytecode instruction `ALOAD`. This instruction loads an object reference onto the stack of the Java Virtual Machine (JVM). To measure the duration of `ALOAD`, a naive approach would insert one `ALOAD` between two timer calls and compute the difference of their results. However, writing such a microbenchmark in Java *source code* is not possible, since there is no source code-level construct which is compiled *exactly* to `ALOAD`.

Furthermore, the resolution of the most precise Java API timer (`System.nanoTime()`) of ca. 280ns is more than two orders of magnitude larger than the duration of `ALOAD` (as shown by our microbenchmarks results). Hence, bytecode microbenchmarks must be constructed through *bytecode engineering* (rather than *source code writing*) and must consider the timer resolution.

Using bytecode engineering toolkits like ASM [8], we could construct microbenchmarks that execute a large number of `ALOAD` instructions between two timer calls. However, to fulfill the bytecode correctness requirements which are enforced by the JVM bytecode verifier, attention must be paid to pre- and postconditions. Specifically, `ALOAD` loads a reference from a register and puts it on the Java stack. However, at the end of the method execution, the stack must be empty again. The microbenchmark must take care of such stack cleanup and stack preparation explicitly.

In reality, creating pre- and postconditions for the entire Java bytecode instruction set is difficult. Often, "helper" instructions for pre-/postconditions must be inserted *between* the two timer calls. In such a case, "helper" instructions are measured together with the actually benchmarked instructions. Thus, *separate* additional "helper" mircobenchmarks must be created to be able to subtract the duration of "helper" instructions from the results of the actual microbenchmarks. Making sure all such dependencies are met and resolved is a challenging task.

Due to space restrictions, we cannot go into further details by describing the design and the implementation of our microbenchmarks. In fact, we have encapsulated the benchmarking into a toolchain that can be used without adaptation on any JVM. End users are not required to understand the toolchain unless they want to modify or to extend it. Selected results of microbenchmarks for instructions and methods will be presented in Section 4 in the context of a case study which evaluates our approach and thus also the microbenchmark results.

3.6 Performance Prediction

Step 6 performs an elementwise multiplication of all N relevant instruction/method counts c_i from step 4 with the corresponding benchmark results (execution durations) t_i from step 5. The multiplication results are summed up to produce a prediction P for execution duration: $\sum_{i=0}^{N} c_i \cdot t_i \;=:\; P$. The parametrisation over input can be carried over from c_i to the performance prediction result P, for example by computing that an algorithm implementation runs in $(n \cdot 5000 + m \cdot 3500)$ ns, depending on n and m which characterise the input to the algorithm implementation.

4 Evaluation

We evaluated our approach in a case study on the PALLADIOFILESHARE system, which is modeled after file sharing services such as RapidShare, where users upload a number of files to share them with other users. In PALLADIOFILESHARE, the uploaded files are checked w.r.t. copyright issues and whether they already are stored in PALLADIOFILESHARE. For our case study, we consider the upload scenario and how PALLADIOFILESHARE processes the uploaded files.

The static architecture of PALLADIOFILESHARE is depicted in Figure 4. The component that is subject of the evaluation is PalladioFileShare (the composite component shaded in grey), which provides the file sharing service interface and itself requires two external storage services (LARGEFILESTORAGE is optimized for handling large files and SMALLFILESTORAGE is for handling small files).

PALLADIOFILESHARE component is composed from five sub-components. The BUSINESSLOGIC is controlling file uploads by triggering services of sub-components. COMPRESSION (a Lempel-Ziv-Welch (LZW) implementation) allows to compress uploaded files, while HASHING allows to produce hashes for uploaded files. EXISTINGFILEDB is a database of all available files of the system; COPYRIGHTEDFILESDB holds a list of copyrighted files that are excluded from file sharing.

Fig. 5 shows the data dependent control flow of the BUSINESSLOGIC component which is executed for each uploaded file. First, based on a flag derived from each uploaded file, it is checked whether the file is already compressed (e.g., a JPEG file). An uncompressed file is processed by the COMPRESSION component.

Afterwards, it is checked whether the file has been uploaded before (using EXISTINGFILEDB and the hash calculated for the compressed file), since only new files are to be stored in PALLADIOFILESHARE. Then, for files not uploaded before, it is checked whether they are copyrighted using COPYRIGHTEDFILESDB. Finally, non-copyrighted

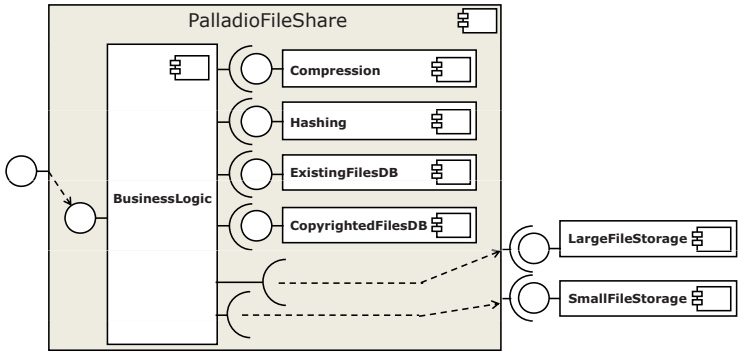

Fig. 4. Component Architecture of PALLADIOFILESHARE

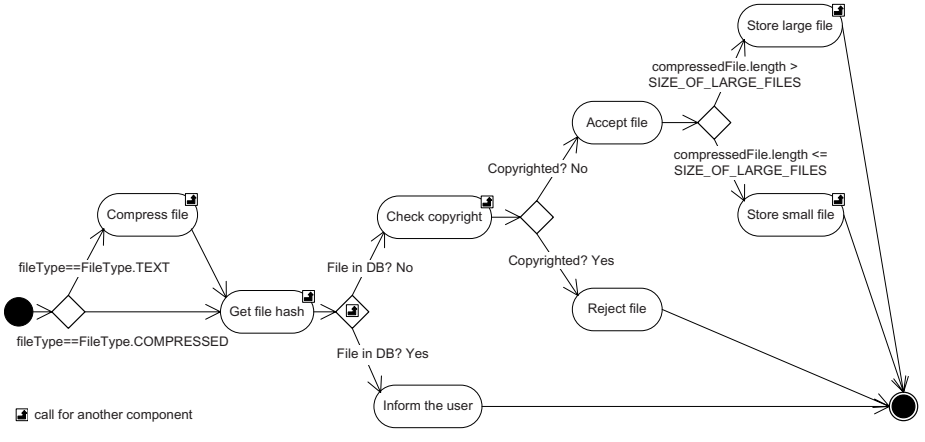

Fig. 5. Activity of BusinessLogic for each file per request (depicted here for readers convenience only; not seen by the tooling)

files are stored, either by LARGEFILESTORAGE for large files (if the file size is larger than a certain threshold) or SMALLFILESTORAGE otherwise.

4.1 Machine Learning for Recognition of Parametric Dependencies

In the case study, we monitored the behaviour of BusinessLogic in 19 test runs, each with different input data (number of uploaded files, characterisation of files (text or compressed), and file sizes). The test runs were designed to cover the application input space as far as possible. In the rest of this section, we show some interesting excerpts from the complete results.

As the names of input parameters are used hereafter, we use the signature of the file sharing service, `void uploadFiles(byte[][] inputFiles,int[] file`

`Types)`. In the signature, `inputFiles` contains the byte arrays of multiple files for upload and `fileTypes` is an array indicating corresponding types of the files, e.g. `FileType.COMPRESSED` or `FileTypes.TEXT` (i.e., uncompressed).

Data dependent control flow: Use of Compression component for multiple files
In the BusinessLogic sub-component, the number of calls of the Compress component depends on the number of uncompressed files (`FileType.TEXT`) uploaded. Genetic programming (JGAP) found the correct solution for the number of calls: *inputFiles.length - fileTypes.SUM(FileType.TEXT)*, where SUM is aggregated data from the monitoring step. The search time was less than one second.

Learning of Bytecode Instruction Counts
For estimating the functional dependencies of bytecode counts, two input variables were monitored: (i) *X1* as the size of each file and (ii) *X2* as flag showing whether the input was already compressed (*X2=1*) or not (*X2=0*).

As the behaviour of data compression algorithm strongly depends on the inner characteristics of the compressed data (and not only on its size and type), 100% precision of learned functions cannot be expected in the general case. Optimal solutions were found only for a few bytecode counts; in most cases, results of machine learning are good estimators. An optimal solution was found within about 1 sec. for bytecode instruction`ICONST_M1` : $X1 + X1 + 3.0$.

For a more complex case such as the bytecode `ICONST_0`, after evolving 15,000 generations, the following approximation (which could be simplified by a subsequent step) was found:

$0.1 + (((X1/(89.0 + 241.0 + 100.0)) * X1) + (((343.0 * X2) + (IF(267.0 >= 10.0)THEN(1.3)) + (241.0 + (X1/(241.0 + (343.0 * X2) + 100.0)) + 100.0)) * X1) + ((IF(X1 >= 0.024766982)THEN((267.0 * X2))) * X2)) + (241.0 + (200.0 + ((89.0 + (((((X1/(X1 + 241.0 + 100.0)) + 30.0 + 241.0) + X2 + 1.9) * X2) * X2) + 100.0) * ((IF(267.0 >= 10.0)THEN((1.3 * X2))) + X1 + 241.0)) + 400000.0))$

The complexity of these functions will be hidden from the user in the performance prediction toolchain.

When to use LargeFileStorage or SmallFileStorage
For answering this question, monitoring data from uploads with just one file was analysed. A set of eleven different input files (different file types, different size) was used as test data. JGAP found an optimal solution: If the file size is larger than 200,000 (bytes), a file with the same size like the file passed to the Hashing component is passed to LargeFileStorage, else nothing is stored with LargeFileStorage (an opposite dependency was found for the usage of SmallFileStorage). The search time was less than five seconds. The implementation-defined constant '200,000' was not always identified correctly, due to the limited number of input files, yet the recovered function did not contradict the monitoring data.

We tested an additional run of JGAP where the monitoring data was disturbed by calls of *uploadFiles* that did not lead to a storage write because the file already existed in the database (one out of eleven calls did not lead to a write). Such effects depending

on component state are visible at the interface level only as statistical noise that cannot be explained based on interface monitoring data. In this case the optimal solution could still be found, but within more time: less than 20 seconds (in average). In this case the confidence in the correctness ("fitness function" calculated by JGAP) of the result decreased. The average behavioural impact of uploads where no storage takes place can be captured by computing the long-term probability of such uploads independently of the uploaded files.

Estimation of the compression ratio
As the compression ratio of LZW strongly depends on the data characteristics (e.g. entropy, used encoding), no optimal solution exists to describe the compression ratio. Therefore, JGAP produces a large variety of approximations of the compression ratio. A good approximation found after 30 seconds had the following form: $0.9 * 0.5 * (X3 - (0.9 * 0.5 * (X3 - (0.9 * (0.9 * 0.5 * X3) * 1.0)))))$ where $X3$ is the size of the file input for the Compression component, which was found to be significant.

4.2 Benchmarking of Bytecode Instructions and Method Invocations

We have benchmarked bytecode instructions and methods (as described in Section 3.5) on two significantly different execution platforms to make performance prediction for the redeployment scenario (cf. Section 1). The first platform ("P1") featured a single-core Intel Pentium M 1.5 GHz CPU, 1 GB of main memory, Windows XP and Sun JDK 1.5.0_15. The second platform ("P2") was an Intel T2400 CPU (two cores at 1.83GHz each), 1.5GB of main memory and Windows XP Pro with Sun JDK 1.6.0_06.

All microbenchmarks have been repeated systematically and median of measurements has been taken for each microbenchmark. Fig. 6 is an excerpt of the results of our microbenchmark for P1 and P2. It lists execution durations of 9 bytecode instructions among those with highest runtime counts for the compression service.

Due to the lack of space, full results of our microbenchmarks cannot be presented here, but even from this small subset of results, it can be seen that individual results differ by a factor of three (`ARRAYLENGTH` and `ICONST_0`). Computationally expensive instructions like `NEWARRAY` have performance results that depend on the passed parameters (e.g. size of the array to allocate), and our benchmarking results have shown that the duration of such instructions can be several orders of magnitude larger than that of simpler instructions like `ICONST_0`.

The most important observation we made when running the microbenchmarks was that the JVM did not apply just-in-time compilation (JIT) during microbenchmark execution, despite the fact that JIT was enabled in the JVM. Hence, prediction on the basis of these benchmarking must account for the "speedup" effect of JIT optimisations that are applied during the execution of "normal" applications.

Some steps in Fig. 5 (such as "Get file hash") make heavy use of methods provided by the Java API. To benchmark such API calls and to investigate whether their execution durations have parametric dependencies on method input parameters, we have *manually* created and run microbenchmarks that vary the size of the hashed files, algorithm type etc. Due to aforementioned space limitations, we cannot describe the results

	ALOAD	ARRAYLENGTH	ANEWARRAY	BALOAD	ICONST_0	IF_ICMPLT	IINC	ILOAD	ISTORE
P1	1.95	5.47	220.42	6.98	1.41	5.08	3.10	3.21	3.45
P2	3.77	2.01	178.79	3.49	1.68	4.30	3.01	2.10	3.05

Fig. 6. Excerpt of microbenchmark results for platforms P1 and P2: instruction durations [ns]

of API microbenchmarks here. To simplify working with the Java API, we are currently working towards automating the benchmarking of Java API methods.

4.3 Performance Prediction

After counting and benchmarking have been performed, our approach predicts the execution durations of the activities in Fig. 5. From these individual execution durations, response time of the entire service will be predicted. These prediction results are platform-specific because underlying bytecode timings are platform-specific.

First, for source platform P1, we predict the duration of compressing a text file (randomly chosen) with a size of 25 KB on the basis of bytecode microbenchmarks, yielding 1605 ms. Then, we measure the duration of compressing that file on P1 (124 ms) and calculate the ratio $R := \frac{bytecode-based\ \ prediction}{measurement}$. R is a constant, algorithm-specific, multiplicative factor which quantifies the JIT speedup and also runtime effects, i.e. effects that are not captured by microbenchmarks (e.g. reuse of memory by the compression algorithm). R's value on P1 for the compression algorithm was 12.9.

Hence, R serves to *calibrate* our prediction. In our case study, R proved to be algorithm-specific, but platform-independent and also valid for any input to the considered algorithm. Using R obtained on platform P1, we have predicted the compression of the same 25 KB text file for its relocation to platform P2: 113 ms were predicted, and 121 ms were measured (note that to obtain the prediction, the compression algorithm was neither measured nor executed on P2 !). We then used the *same* calibration factor R for predicting the duration of compressing 9 additional, different files on platform P2 (these files varied in contents, size and achievable compression rate). For each of these 9 files, the prediction accuracy was within 10% (except one outlier which was still predicted with 30% accuracy).

This shows that the calibration factor R is input-agnostic. Also, R can be easily obtained in the presented relocation scenario because an instance of the application is already running on the "source" execution platform P1 (note that the prediction of performance on P1 is only needed for relocation, as the real performance on P1 is available by measuring the already deployed application).

The performance of the hashing action in Fig. 5 was predicted by benchmarking the underlying Java API calls, whereby a linear parametric dependency on the size of input data was discovered. The JIT was carried out by the JVM during benchmarking of these API calls, which means that R does not need to express the JIT speedup. For example, hashing 36747 bytes of data on P2 was predicted to 1.71 ms while 1.69 ms were measured, i.e. with $< 2\%$ error. Similar accuracy for predicting hashing duration is obtained for other file sizes and types.

The total upload process for the above 25KB text file on P2 was predicted to take 115 ms, and 123 ms were measured. Upload of 37 KB JPEG (i.e. already compressed)

file took 1.82 ms, while 1.79 ms were predicted. For all files used in our case study, the prediction of the entire upload process for one file had an average deviation of $< 15\%$.

Ultimately, our bytecode-based prediction methodology can deal with all four factors discussed in Sec. 1: *execution platform* (as demonstrated by relocation from P1 to P2), *runtime usage context* (as demonstrated by the prediction for different input files), *architecture* of the software system (as we predict for individual component services and not a monolithic application), and the *implementation* of the components (as our predictions are based on the bytecode implementation of components). From these results, we have concluded that a mix of upload files can be predicted if it is processed sequentially. However, for capturing effects such as multithreaded execution, further research is needed to study performance behaviour of concurrent requests w.r.t. bytecode execution. In the next section, we discuss the assumptions and the limitations of our approach.

5 Limitations and Assumptions

For the monitoring step, we assume that a representative workload (including input parameter values) can be provided, for example by a test driver. This workload has to be representative for both current and planned usage of the component. For running systems, this data can be obtained using runtime monitoring; otherwise, a domain expert judges which scenarios are interesting or critical, and she should select or specify the corresponding workloads ([21] provides an overview on test data generation).

To predict performance on a new (or previously unknown) execution platform, our approach does not need to run the application there, but must run the microbenchmark suite on the new platform. Hence, we assume that either this is possible for the predicting party, or that the microbenchmark results are provided by a third party (for example, by the execution platform vendor).

One of the current limitations of our approach is that it is not fully automated. For example, the parts **A** and **B** in Fig. 1 are not integrated for an automated performance prediction. Also, API calls must be measured manually to consider parametric dependencies and complicated parameter conditions; hence, only a limited number of API calls can be supported realistically.

In the data gathering step of our approach, asynchronous communication (e.g. message-based information exchange) is not supported by the used logging framework. Hence, if there is asynchronous communication inside the component under investigation, monitored results will be misleading. This limitation will be addressed in next versions of our implementation.

To support the black-box component principle (end-users do not have to deal with code), monitoring should be performed in an automated way. In general, collecting dozens of metrics for input and output data is not justified by the requirements of our approach. At the moment, we assume that all input and output data is composed from primitive types or general collection types like `List`. In more elaborate cases, a domain expert can specify important data characteristics manually to improve the monitoring data base.

In the machine learning step, heavily disturbed results (i.e. those having causes not visible at the interface-level) lead to decreased convergence speed and smaller probability of finding a good solution.

6 Conclusions

In this paper, we have presented a performance prediction approach supporting black-box software components by creating platform-independent parametric performance models. The approach requires no a-priori knowledge on the components under investigation. By explicitly considering parameters in the performance model, the approach enables prediction for different execution platforms, different usage contexts, and changing assembly contexts.

In the described approach, bytecode is monitored at runtime to count executed bytecode instructions and method calls, and also for gathering data information at component interface level to create the parametric performance model. Then, bytecode instructions and methods are benchmarked to obtain their performance values for a certain platform. The advantage of separating behaviour model from platform-specific benchmarking is that the performance model must be created only *once* for a component-based application, but can be used for predicting performance for any execution platform by using platform-specific benchmark results.

We evaluated the presented approach using a case study for the Java implementation of a file-sharing application. The evaluation shows that the approach yields accurate prediction results for (i) different execution platforms and (ii) different usage contexts. In fact, the accuracy of predicting the execution duration of the entire upload process after redeployment to a new execution platform lies within 15% for all considered usage contexts (i.e. uploaded files), and even within 5% in all but three contexts. The average accuracy is therefore also very good.

For our future work, we plan to automate the entire approach and to merge bytecode counting in Step 1 of our approach with data monitoring and recording in Step 2. The manual execution of the approach took ca. five hours for the case study. Also, we plan to automate creating microbenchmarks for methods, which currently must be created by hand and also do not cover the entire Java API. We also plan to consider parameters at bytecode level both for bytecode microbenchmarks and method microbenchmarks.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Experiments in Cost Analysis of Java Bytecode. Electr. Notes Theor. Comput. Sci. 190(1), 67–83 (2007)
2. Becker, S., Happe, J., Koziolek, H.: Putting Components into Context - Supporting QoS-Predictions with an explicit Context Model. In: Reussner, R., Szyperski, C., Weck, W. (eds.) WCOP 2006 (June 2006)
3. Becker, S., Koziolek, H., Reussner, R.: The Palladio Component Model for Model-Driven Performance Prediction. Journal of Systems and Software (in press, 2008) (accepted manuscript)
4. Bertolino, A., Mirandola, R.: CB-SPE Tool: Putting Component-Based Performance Engineering into Practice. In: Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 233–248. Springer, Heidelberg (2004)
5. Binder, W., Hulaas, J.: Flexible and Efficient Measurement of Dynamic Bytecode Metrics. In: GPCE 2006, pp. 171–180. ACM, New York (2006)
6. Binder, W., Hulaas, J.: Using Bytecode Instruction Counting as Portable CPU Consumption Metric. Electr. Notes Theor. Comput. Sci. 153(2), 57–77 (2006)

7. Bondarev, E., de With, P., Chaudron, M., Musken, J.: Modelling of Input- Parameter Dependency for Performance Predictions of Component-Based Embedded Systems. In: Proceedings of the 31th EUROMICRO Conference (EUROMICRO 2005) (2005)
8. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems. Adaptable and Extensible Component Systems (2002)
9. Courtois, M., Woodside, C.M.: Using regression splines for software performance analysis. In: WOSP 2000, Ottawa, Canada, September 2000, pp. 105–114. ACM, New York (2000)
10. Donnell, J.: Java Performance Profiling using the VTune Performance Analyzer (Retrieved 2007-01-18) (2004)
11. Ernst, M.D., et al.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1-3), 35–45 (2007)
12. Harman, M.: The Current State and Future of Search Based Software Engineering. In: Future of Software Engineering, 2007. FOSE 2007, May 23-25, 2007, pp. 342–357 (2007)
13. Herder, C., Dujmovic, J.J.: Frequency Analysis and Timing of Java Bytecodes. Technical report, Computer Science Department, San Francisco State University, Technical Report SFSU-CS-TR-00.02 (2000)
14. Hrischuk, C.E., Murray Woodside, C., Rolia, J.A.: Trace-based load characterization for generating performance software models. IEEE Transactions Software Engineering 25(1), 122–135 (1999)
15. Hu, E.Y.-S., Wellings, A.J., Bernat, G.: Deriving Java Virtual Machine Timing Models for PortableWorst-Case Execution Time Analysis. In: Meersman, R., Tari, Z. (eds.) OTM-WS 2003. LNCS, vol. 2889, pp. 411–424. Springer, Heidelberg (2003)
16. Israr, T., Woodside, M., Franks, G.: Interaction tree algorithms to extract effective architecture and layered performance models from traces. Journal of Systems and Software, 5th International Workshop on Software and Performance 80(4), 474–492 (2007)
17. Koza, J.R.: Genetic Programming – On the Programming of Computers by Means of Natural Selection, 3rd edn. MIT Press, Cambridge (1993)
18. Kuperberg, M., Becker, S.: Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs. In: Reussner, R., Czyperski, C., Weck, W. (eds.) WCOP 2007 (July 2007)
19. Kuperberg, M., Krogmann, M., Reussner, R.: ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations. In: BYTECODE 2008 (2008)
20. Lambert, J., Power, J.F.: Platform Independent Timing of Java Virtual Machine Bytecode Instructions. In: Workshop on Quantitative Aspects of Programming Languages, Budapest, Hungary, March 29-30 (2008)
21. McMinn, P.: Search-based software test data generation: a survey. Software Testing, Verification and Reliability 14(2), 105–156 (2004)
22. Meffert, K.: JGAP - Java Genetic Algorithms Package (last retrieved: 2008-03-18), `http://jgap.sourceforge.net/`
23. Meyerhöfer, M., Meyer-Wegener, K.: Estimating Non-functional Properties of Component-based Software Based on Resource Consumption. Electr. Notes Theor. Comput. Sci. 114, 25–45 (2005)
24. Smith, C.U., Williams, L.G.: Performance Engineering Evaluation of Object- Oriented Systems with SPEED. In: Marie, R., Plateau, B., Calzarossa, M.C., Rubino, G.J. (eds.) TOOLS 1997. LNCS, vol. 1245, Springer, Heidelberg (1997)
25. Zhang, X., Seltzer, M.: HBench:Java: an application-specific benchmarking framework for Java virtual machines. In: JAVA 2000: Proceedings of the ACM 2000 conference on Java Grande, pp. 62–70. ACM Press, New York (2000)