Using Genetic Search for Reverse Engineering of Parametric Behavior Models for Performance Prediction

Klaus Krogmann, Michael Kuperberg, *Member*, *IEEE*, and Ralf Reussner, *Member*, *IEEE*

Abstract—In component-based software engineering, existing components are often reused in new applications. Correspondingly, the response time of an entire component-based application can be predicted from the execution durations of individual component services. These execution durations depend on the runtime behavior of a component which itself is influenced by three factors: the execution platform, the usage profile, and the component wiring. To cover all relevant combinations of these influencing factors, conventional prediction of response times requires repeated deployment and measurements of component services for all such combinations, incurring a substantial effort. This paper presents a novel comprehensive approach for reverse engineering and performance prediction of components. In it, genetic programming is utilized for reconstructing a behavior model from monitoring data, runtime bytecode counts, and static bytecode analysis. The resulting behavior model is parameterized over all three performance-influencing factors, which are specified separately. This results in significantly fewer measurements: The behavior model is reconstructed only once per component service, and one application-independent bytecode benchmark run is sufficient to characterize an execution platform. To predict the execution durations for a concrete platform, our approach combines the behavior model with platform-specific benchmarking results. We validate our approach by predicting the performance of a file sharing application.

Index Terms—Genetic search, genetic programming, reverse engineering, performance prediction, bytecode benchmarking.

1 Introduction

During design and evolution of software systems, functional and extra-functional requirements must be fulfilled. While current object-oriented development approaches provide a largely defined, systematic, and repeatable way to translate functional requirements into an appropriate implementation, the systematic consideration of extra-functional requirements is still a matter of research. In the following, we concentrate on performance as an extra-functional requirement, while in general extra-functional requirements can also be concerned with reliability, security (to name a few), or internal quality attributes such as maintainability.

Various systematic approaches for architecture-based software performance prediction have been developed, with Software Performance Engineering (SPE) by Smith [34] being among the oldest. More modern approaches reflect properties of software architectures by componentization according to a predefined component model (e.g., [4], [36], [25]).

Such approaches share the idea that through the compositionality of components, the performance of a whole system design can be predicted based on *performance models* of single components before actually investing efforts in implementing, integrating, and deploying them.

• *The authors are with the Karlsruhe Institute of Technology (KIT), Institute for Program Structures and Data Organization, Am Fasanengarten 5, D-76131 Karlsruhe, Germany.*
E-mail: {krogmann, kuperberg, reussner}@kit.edu.

Manuscript received 1 Sept. 2008; revised 29 Aug. 2009; accepted 21 Sept. 2009; published online 21 July 2010.
Recommended for acceptance by M. Harman and A. Mansouri.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2008-09-0263.
Digital Object Identifier no. 10.1109/TSE.2010.69.

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

1. their *assembly context*, i.e., on the architecture of the software system and the interconnection of the components,
2. the runtime *usage context* of the application (e.g., values of input parameters, number of users, etc.),
3. the *deployment context*, i.e., the execution platform which comprises hardware (CPU and HDDs) and software (operating system, virtual machine, etc.),
4. the *implementation* of the individual components that comprise the software system.

Predicting a performance metric for *fixed* values of all four factors is not helpful for real-world software systems. The following scenarios, common in software engineering projects, can only be supported if these four factors (1-4) are made explicit:

- **Redeployment** of applications on execution platforms with different characteristics (e.g., to cope with increased usage, changes in performance requirements, or if more powerful execution platforms become available). Thus, their *deployment* and *usage contexts* are not fixed.
- **Sizing** of suitable execution platforms to fulfill changed performance targets or service-level agreements for an existing software system, for example, due to changes in the *usage context* (i.e., number of concurrent users, increased user activity, and different input).
- **Reuse** of a single component in another architecture or **architectural changes** in an existing software

system (e.g., if it is connected to a faster nameserver) imply changes in the *assembly context* of a component.

Hence, for performance prediction of component-based systems for changing scenarios, it is imperative to individually quantify the impact of the four factors listed above, i.e., to consider them as explicit *parameters*. Consequently, for design-time prediction, both the system architecture and the components must be represented as *parameterized models* to help in deriving architectural design decisions (e.g., "what is the impact of introducing a cache?"). However, for existing components or software systems, such parameterized models are usually not available.

Manual creation of performance models is impractical, and therefore automated approaches for reverse engineering of performance models have been developed. Unfortunately, existing methods for reverse engineering performance models are either monolithic and do not separate the four factors [11] or require full source code [23]. For design-time performance prediction, such models must be created manually from existing artifacts, such as execution traces, implementation of components, or system designs.

Also, existing performance prediction methods [4], [34] do not explicitly consider all four factors, assuming that they are (partially) fixed, or limit themselves to real-time/embedded scenarios [7]. To make the influence of these four factors on performance explicit (or even quantifiable), existing approaches would need to reconsider each relevant combination of these four factors because performance models and prediction methods are not parameterized over each individual factor. It is important to stress that although executable code is available, for conventional approaches this requires buying each potential execution environment and implies effort to integrate and configure applications in each execution environment. If v_i values of factor i ($i = 1..4$) have to be considered, existing approaches need to analyze $\prod_{i=1}^{4} v_i$ combinations.

In our approach, an application needs to be reverse engineered only once. Using general benchmarking results from the hardware vendor, our approach predicts the performance of an application for the execution environment (without actually touching hardware or software). The four factors can then be changed at the model level to enable what-if-analyses.

In this paper, we present a novel integrated approach that uses genetic search to reverse engineer parameterized behavior models of existing black box components. Support for black box components is important because, for many applications, no source code is available (e.g., for legacy components or due to intellectual property secrets). Our approach is based on static and dynamic analysis and search-based approaches (genetic programming) to create behavior models from monitored data. It is the first approach using genetic search to reverse engineer behavior models of software components for the sake of model-based performance prediction. Using bytecode benchmarking, our approach can predict platform-specific performance metrics (such as execution time) for all of the scenarios S.

The contribution of this paper is twofold: Our first contribution is a validated reverse engineering approach combining static and dynamic analysis that uses genetic programming on runtime monitoring data for creating behavior models of components. These models are parameterized over usage, assembly, and deployment context, and thus *platform-independent*, making them sensitive to changing load situations, connected components, and the underlying hardware. The evaluation section of this paper shows how genetic programming can be applied for finding abstractions of control and data flow and also for estimating resource consumptions of black box components for which only bytecode is available.

Our second contribution is the performance prediction for these behavior models. *Platform-specific* execution durations are predicted on the basis of bytecode benchmarking results, allowing performance prediction for components and also entire component-based applications. Instead of rebenchmarking an entire application for all relevant combinations of usage, assembly, and deployment contexts, it now suffices to specify concrete parameter values for the reverse-engineered behavior models. We also describe our tools to automatically obtain such parameter instances.

We validate our approach by reverse engineering a performance-oriented behavior model for a component-based file sharing application utilizing SPECjvm2008 compress. We then predict the performance of this application on several platforms and for several usage profiles. To the best of our knowledge, this is the first validated bytecode-based performance prediction approach which considers runtime performance optimizations of the Java Virtual Machine.

The benefit of using our approach is seen in the support of the above-mentioned scenarios: redeployment, sizing, changes in the usage context, and reuse of software components.

The paper is structured as follows: In Section 2, we describe related work. In Section 3, an overview of our approach is given, concepts are explained, 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

In this paper, search-based software engineering [18] is applied for reverse engineering of behavioral models.

Search-based approaches, such as simulated annealing, genetic algorithms, and genetic programming, have been widely used in software engineering. Applications of search-based techniques focus on problems like optimization of testing, finding concept boundaries, project planning and cost estimation, or quality assessment [18].

For reverse engineering purposes, search-based techniques have predominantly been applied to the recovery of system structures such as architectures [33] or for clustering system dependency graphs [14]. These approaches require the full source code of systems under investigation as input, while our approach works on bytecode, maintaining the black box principle of software components [35]. To the best of our knowledge, search-based approaches have not been applied to reverse engineering of control and data flow abstractions nor bytecode estimations.

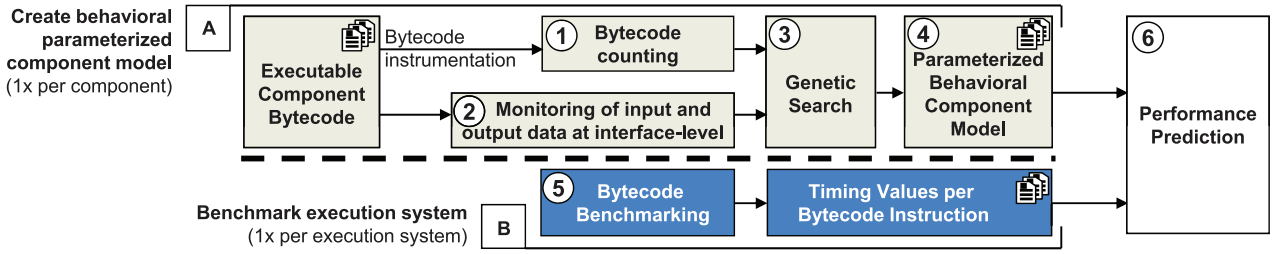

Fig. 1. Overview of the approach.

Daikon by Ernst et al. [15] focuses on detection of invariants from running programs, while our approach aims at detecting parametric propagation and parametric dependencies of runtime behavior w.r.t. performance abstractions. For invariant generation, Daikon uses a mechanism comparable to random generation of individuals in SBSE.

Regression splines are used by Courtois and Woodside in [11] to recognize 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. Yet their fully automated approach assumes fixed external dependencies of software modules and fixed hardware.

Trace-based approaches. Reverse engineering of performance models using traces is performed by Hrischuk et al. [20] in the scope of "Trace-Based Load Characterization (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 [22] using pattern matching. Similarly to our approach, Israr et al. support components and have no explicit control flow, yet they do not support intercomponent data flow.

Other approaches. Zheng et al. [38] focus on runtime monitoring and online prediction of performance, while our approach is suitable for design-time. Their models are reverse engineered by the use of Kalman filters. They are thereby not required to directly monitor performance values of interest, but can estimate them based on known (and easily available) metrics such as response time and resource utilization.

Existing reverse engineering approaches for control flow such as [8] require information on component internals—black box components are not supported. For our approach, bytecode is sufficient.

This paper also is related to **bytecode-based performance prediction**. Performance prediction on the basis of bytecode benchmarking has been proposed by several researchers [19], [32], [37]. However, they consider only *interpreted* execution of bytecode, which is prohibitively slow and which has been superseded by runtime *just-in-time* (JIT) *compilation* of bytecode to fast native code inside the JVM. But, apart from the work presented in this paper, there exists no approach that can account for JIT-induced performance impacts. Even for interpretation-only bytecode execution, no *validated* results have been published: Validation has been attempted in [37], but it was restricted to very few Java API methods, and the nonmethod bytecode instructions were neither analyzed nor benchmarked. In [26], bytecode-based performance prediction that explicitly distinguishes between method invocations and other bytecode instructions has been proposed.

Obtaining execution counts of bytecode instructions has been addressed in commercial tools (e.g., in profilers, such as Intel VTune [13]) as well as by researchers (e.g., [5]).

Execution durations of individual bytecode instructions have been studied independently from performance prediction in [29]; however, its 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 are ignored, which is not realistic. Hu et al. derive worst-case execution time of Java bytecode in [21], but their work is limited to real-time JVMs. Cost analysis of bytecode-based programs is presented in [1], but neither bytecode benchmarks nor actual realistic performance values can be obtained since the performance is assumed to be equal for all bytecode instructions.

This paper goes beyond a former version [27] by using the SPECjvm2008 compress component in the case study, explicitly handling method calls in bytecode, using static bytecode analysis to improve genetic search, introducing a new mutation operator for forcing abstraction, utilizating Multivariate Adaptive Regression Splines, supporting long-running executions, and increasing automation.

3 Reverse Engineering and Performance Prediction

Our approach (see Fig. 1) consists of two parts, separated in Fig. 1 by the dashed line: part A (upper, light shade) reverse engineers behavioral performance models for components through bytecode analysis and genetic programming. These models are platform-independent because they do not contain platform-specific timing values. Such platform-specific timing values are produced by part B (lower, darker), which uses bytecode benchmarking to evaluate the performance of an execution environment. Results of A and B are then fed into the prediction, which results in execution durations of a component-based application for a certain execution environment without actually implementing or integrating components. Here, what-if scenarios at the model level can be investigated.

In Step (1) of Part [A], 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 (2), inputs and outputs of the considered component are monitored at the interface level. (1) and (2) can be executed independently of each other.

Genetic search (genetic programming) in (3) is used to

1. estimate parametric dependencies between input data and the number of executions for each bytecode instruction,
2. find control flow dependencies between provided and required services depending on input data, which is important as, depending on input data, other components might be called or the number of calls can vary (for example, the number of loops another component is called in), and
3. find abstractions of the data flow, which is also important for components since the output of one service will be the input of another one.

(4) uses results from genetic search in (3) and constructs a behavioral component model which is parameterized over usage context (input data), external services, and also over the execution platform. Such a model includes how input data are populated through an architecture, a specification how often external services are called with which parameters, and how an execution platform is utilized. We go into details of the model later.

Part [A] of our approach is executed once per component-based application and also works for components and for component-based applications, for which only bytecode and no source code maybe available.

In (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 a certain target platform. The benchmarking step is completely independent of the previous steps of our approach. Hence, benchmarking needs to be executed only once per 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 (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 (the behavioral model) and execution platform performance model allows estimating 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 costly setup and configuration of a complex software application,
- avoid implementing glue code for integrating components just for evaluating the performance of a architectural design option, and
- avoid buying expensive hardware (given a hardware vendor providing benchmarking results).

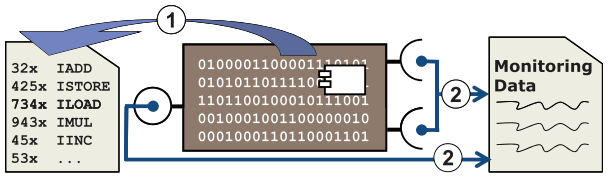

Fig. 2. Data extraction from executed black box components: (1) bytecode instrumentation and execution and (2) monitoring.

3.1 Counting of Bytecode Instructions and Method Invocations

To obtain runtime counts of executed bytecode instructions (cf. Fig. 1, (1)), we use the BYCOUNTER tool, which is described in detail in [28] 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. Code obfuscation is not an issue for the instrumentation, as any obfuscated bytecode must conform to the Java bytecode specification. BYCOUNTER can handle any valid bytecode.

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 (at most 250 percent, see [28]) and is comparable to 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 (5).

The counting results (cf. Fig. 2, (1)) are counts for each bytecode instruction and found method signature. 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 (reference) execution platform. Counting results are saved individually for each usage context and later serve as data on which genetic search is run (cf. Fig. 1, (3)).

3.2 Data Gathering from Running Code

To capture control and data flow between provided and required services of a component, we monitor at the level of component interfaces (cf. Fig. 1, (2)). Therefore, we gather runtime information about component behavior by executing the component in a testbed or executing the entire application (cf. Fig. 2, (2)).

Representative monitoring data are obtained by executing the monitored component services with a set of

representative inputs. Monitored data then serve as the input for the genetic search (Fig. 1, ③) to learn parametric dependencies in control and data flow (i.e., which required services of a component are executed for which input data and the number of their executions; which input data at the provided interfaces is forwarded or transformed to which required interfaces). Since our approach does not capture any timing information during monitoring, general approaches for generating test cases covering the input space of methods (e.g., [10]) are suitable for generating representative inputs (recent overview in [30]).

For each service of a component interface, our tool monitors every call (provided and required). The tool captures the input parameters of each component service call and the properties of the data that is returned by that service call.

Heuristics that depend on data types allow the tool to capture relevant data properties. 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`): the array size, i.e., the number 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 they are called, and which parameters are used. These data are used for reverse engineering the component-internal control flow.

It must be pointed out that for the approach, data are collected at the component level. For large components, thousands of line of code can be covered, resulting in only a single measuring point: Only component-external communication is covered. Thus, the granularity of components is suited to steering the amount of data collected since, for larger components, larger portions of code perform only component-internal communication.

Uniquely in our approach, there is a one-time collection of pure counts (and no timing information) for the application considered. Timing values are calculated in a second step. Hence, the accuracy of the approach is not influenced by monitoring overhead at all. The size of monitored data immediately results from the monitored data: For each measuring point, the data itself (e.g., 2 Bytes for a double value), an ID of the measurement position, and a measurement number are stored.

The described data monitoring and recording heuristics can be applied to component interfaces in an automated way without a priori knowledge about their semantics and without inspecting the internals of components. Heuristics for capturing data properties can be easily extended. Monitoring complex or self-defined types (e.g., objects and structs) requires domain expert knowledge to identify important properties of these data types. Generic data types are used very often, and our approach can handle these cases automatically.

Capturing the full data passed through an interface would result in bad monitoring performance (imagine, for example, service that exchanges large video files) without having extra information available for genetic search. For example, the full content of a video file would seldom result in insights on a compression ratio or length cut. Instead, *abstracted and aggregated* data properties (such as the file size) are important inputs for genetic search.

3.3 Genetic Search in the Context of Reverse Engineering

Recognition of parametric dependencies in the reverse engineering phase **A** of our approach utilizes genetic search for estimating the bytecode counts parameterized over input data. Additionally, genetic search is used for recovering functional dependencies in control and data flow in the monitored data. If, for example, a component service computes the hash of some data, our genetic search is used to discover

- Bytecode counts estimation: which bytecode instructions are executed how often during the execution of the hashing algorithm depending on the input data.
- Control flow: that a required external service is called to save the hashed files if they are uncompressed.
- Data flow: the hash values are passed to other component services (which implies that the performance of those services depends on the type and size of computed hash) only with compressed size.

For this paper, we decided to use *genetic programming* as the genetic search technique. Approaches like linear regression, regression splines [11], or support vector machines (SVMs, [12)] could be applied as well, but they do not handle noncontinuous functions well and produce approximations that do not lend themselves easily to human understanding. Noncontinuous functions are quite common in applications, e.g., the discontinuity caused by branching constructs (if-then-else, case statements, and the like).

Due to the black box setting considered in this paper, information on the internal control and data flow of components is not available. Genetic programming is designed to find solutions for a given programming task, which in our case is represented by monitoring data and bytecode counts. Thus, internal dependencies which originate from programming code must be guessed from data captured for black box components.

Genetic programming was invented to form programming code (for example, it uses an abstract syntax tree for data representation; see below) which makes it perfectly applicable for approximating components' control and data flow. To restrict prediction complexity, performance models require abstractions of the real components behavior, and genetic programming allows finding such abstractions. Monitoring data that is disturbed by component-internal effects (leading to contradictory data) also does not cause trouble as genetic programming is robust against disturbed input data [24, p. 569].

We use the Java Genetic Algorithm Package (JGAP) [31], which supports genetic programming as genetic search implementation. A general introduction for genetic programming, a special case of genetic algorithms, can be found in [24].

Selection of genes, mutation, and crossover. For our approach, we combine genes representing mathematical functions to express more complex dependencies. 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. We used a selection of JGAP out-of-the-box genes for mathematical operations: power, multiplication, addition, sine, exponential function, and constants. Additionally, we added support for inequations, if-then, and if-then-else genes to support noncontinuous behavior (e.g., jumps caused by "if-then-else" in code).

To increase convergence speed, we introduced special constants genes that have a predefined set of possible values or ranges of values (either integer or float). For example, float values in between 0.0 and 1.0 can be used instead of the full float range. Integer sets like 10, 100, 10,000, ... can be used to scale up values and then can be refined by mutations. Our findings show that the convergence speed and precision of results of genetic programming can be heavily increased through an antecedent *static code analysis* that searches for constants in code. If those constants—which are found in bytecode—are used in genes for genetic search, the time needed for search decreases and more optimal solutions can be found more easily. Code obfuscation could replace constants with summands of constants. Generally, this affects the convergence speed but not the precision of results.

Our approach uses Multivariate Adaptive Regression Splines (MARS) by Friedman [16], a high-quality [16, p. 68] regression technique from statistics which automatically models nonlinear behavior, to create a nonrandom initial generation. Results from MARS are therefore translated to an individual of the initial generation. Since such individuals are comprised of regular genes, they are subject to mutation and crossover and can be further refined. Results of genetic programming are never worse than MARS results and also convergence speed increases.

Genetic programming uses all available genes to build a tree structure which represents a mathematical expression. Mutation is applied to constants genes, while crossover operates on subtrees of individuals. Genetic programming selects appropriate input values and rejecting others, not relevant for the functional dependency. This is especially important for estimating control and data flow as the potential input space is relatively large due to multiple data characteristics monitored, of which not all need to be important. An additional special mutation operating on whole individuals forces variable selection. The mutation removes single variables (dimensions) from individuals and thus contributes to the abstraction of expressions.

Fitness function and break condition. The fitness of individuals is judged according to two basic criteria: *prediction error* and *complexity of the found expression*. The prediction error (mean-squared error, MSE) is given by the deviation between monitored values (data/bytecode counts) and values predicted by the mathematical expression found by genetic search. To avoid an impact of large measured numbers on the fitness, the error is normalized over the median of measured values. Then, an MSE of the median value results in an error of 1.0. Complexity should be as low as possible so that an increased abstraction level of the overall expression is forced, expressions remain understandable for humans if possible, and prediction overhead is low. For computing expression complexity, we consider the depth of the tree of genes, the length of the resulting mathematical expression (number of genes), and the number of dimensions (i.e., variables representing input parameters) involved. If a certain threshold is passed (two dimensions, a depth of 6, six genes), the fitness of individuals is reduced for the degree excessing the threshold. The prediction error has a heavier weight (15×) than complexity to prefer good prediction results over low complexity:

$$\begin{aligned} Error := 15 * MSE - Normalized(prediction,\\ monitoringdata) + DepthOfTree(6)\\ + LengthOfExpression(6)\\ + NumberOfDimensions(2), \end{aligned}$$

where the number in brackets specifies the threshold. Reduced complexity is needed for two reasons: Complex expressions result in time-consuming predictions and simple expressions are easier for humans to read, which is necessary for performance optimizations of models in later stages.

The Generalized Cross Validation (GCV, [17]) is a commonly used smoothing parameter selection method in nonparametric regression, reflecting ideas similar to our fitness function. It also addresses *complexity* and overcomes the limitations of error measures like least-square error which take only the error into account. As it is a general error measure, it does not include domain knowledge on the complexity.

As the break condition for evolution of genes, our approach uses a simple rule: either when the prediction error is zero (optimal solution) or when a fixed number of generations has been evaluated (default values are 1,500 generations and 200 individuals per generation). Users are asked to have a look at the best solution found after a certain number of generations and can decide to extend the search time, so that more generations are evaluated. For control and data flow, optimal solutions have been found in the majority of cases that we have studied in the past; for bytecode counts, a (good) approximation is typically the best that can be achieved, due to the complexity of bytecode implementations of components and services.

3.3.1 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, for example, 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 bytecode instruction used. For bytecode instructions and method invocation counts, learning such functions produces more helpful results than mere average counts because dependencies between input data and execution counts can be expressed even in nonlinear cases.

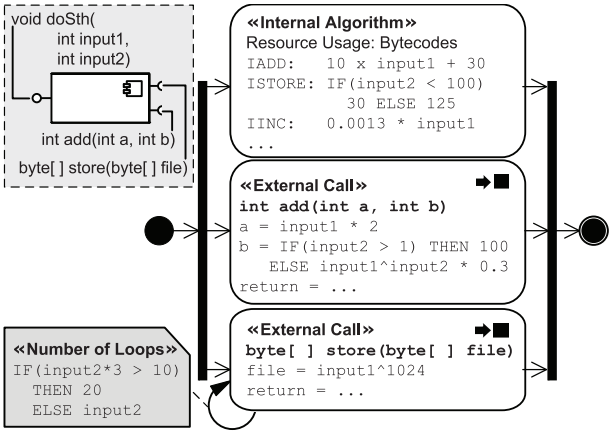

Fig. 3. Behavioral model of the provided service `void doSth(int input1, int input2)`.

A simple example of the resulting estimation for the `ILOAD` instruction is $IF(filesize > 1{,}024)\ THEN\ (filesize \cdot 1.4 + 300)\ ELSE\ (24{,}000)$. Here, up to a certain file size (1,024 KB), a constant number of bytecode instructions is used, otherwise a linear function specifies the number of executed bytecode instructions.

3.3.2 Learning Functional Dependencies between Provided and Required Services: Data and Control Flow

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 0.7 times the size of an input file" (data flow). Data and control flow can depend on input parameters of the provided service but also on return data of required services—both variants are supported.

3.4 Parameterized Model of Component Behavior

Our approach utilizes the Palladio Component Model[1] [3] in performance prediction. It is an ECORE metamodel referred to as "Parameterized Behavioral Component Model" in Fig. 1. The metamodel offers a representation for the static structure elements (software components, composition, and wiring; not described here) and a behavioral model for each provided service of a component (an example is shown in Fig. 3).

A component service's behavior model consists of *internal algorithms* (i.e., algorithms realized within a component) and *external calls* (calls to services of other components). For *internal algorithms* (topmost box in Fig. 3), the reverse-engineered formulas from the genetic search step estimate for each bytecode instruction how often an instruction is executed, depending on component service's input parameters. In the behavioral model, parameterized counts that form these annotations are platform-independent and do not contain platform-specific timing values.

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

For *external calls* (e.g., `add` and `store`, in Fig. 3), the model includes:

- **Parametric data flow descriptions:** Dependencies between component service input and external call parameters, with one formula per input parameter of an external call (e.g., $\texttt{a} = \texttt{input1} * 2$ in Fig. 3).
- **Data-flow-dependent control flow descriptions:** The number of calls to each required (external) service is annotated using parameterization over input data (cf. "Number of loops" gray box in Fig. 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 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.

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 about 280 ns 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 [9], 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" microbenchmarks 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 detail for describing the design and the implementation of our microbenchmarks. In fact, we have encapsulated the

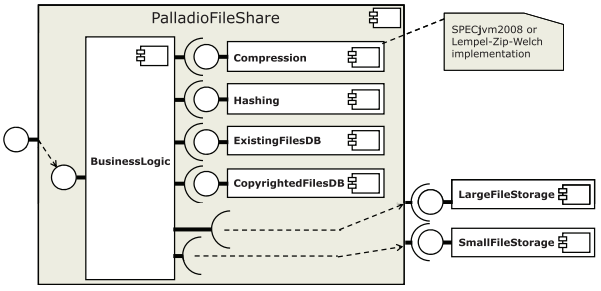

Fig. 4. Component architecture of PALLADIOFILESHARE.

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 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 microbenchmarking results.

3.6 Performance Prediction

(6) performs an elementwise multiplication of all N relevant instruction/method counts c_i from (4) with the corresponding benchmark results (execution durations) t_i from (5). The multiplication results are summed up to produce a prediction P for execution duration: $P := \sum_{i=0}^{N} c_i \cdot t_i$. The parameterization 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 $(n \cdot 5{,}000 + m \cdot 3{,}500)$ ns, depending on n and m which characterize the input to the algorithm implementation.

4 EVALUATION

We evaluated our approach in a case study on the PALLADIOFILESHARE system, which is comparable to file sharing services such as RapidShare, where users upload a number of files to share them with other users. In PALLADIOFILESHARE, uploaded files are checked for copyright issues. Storing duplicate files is avoided by an additional check. For our case study, we consider the upload scenario and how PALLADIOFILESHARE processes the uploaded files.

The static architecture of PALLADIOFILESHARE is depicted in Fig. 4. The component that is subject of the evaluation is PALLADIOFILESHARE (the composite component shaded in gray), 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).

The PALLADIOFILESHARE component is composed of five subcomponents. The BUSINESSLOGIC controls file uploads by triggering services of subcomponents. COMPRESSION allows compression of uploaded files, while HASHING produces hashes for uploaded files. EXISTING FILEDB is a database of all available files of the system; COPYRIGHTEDFILESDB holds a list of copyrighted files that are excluded from file sharing. COMPRESSION can be chosen between two versions: the SPECjvm2008 and a Lempel-Zip-Welch (LZW) implementation—we will later compare the predictions for both.

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.

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

4.1 Genetic Search for Recognition of Parametric Dependencies

In the case study, we monitored the behavior of the BUSINESSLOGIC component in 19 test runs, each with different input data (number of uploaded files, characterization of files (text or compressed), and file sizes) on a reference platform. The test runs were designed to cover the application input space as far as possible. For evaluating genetic search success, we compared manually reverse-engineered parametric dependencies (humans having a look into the original source code) with those from genetic search. In the rest of this section, we show some interesting excerpts from the complete results.

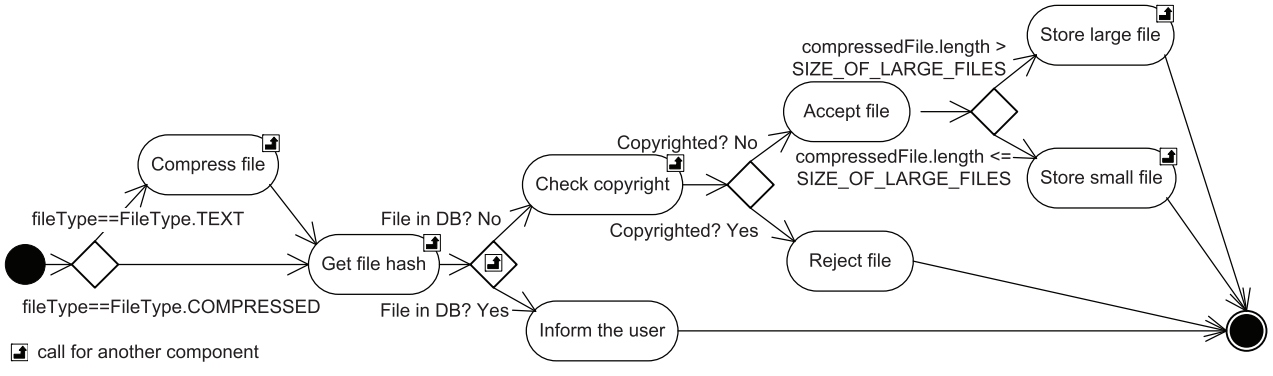

Fig. 5. Activity of BusinessLogic for each file per request (depicted here for the readers convenience only; not seen by the tooling).

As the names of input parameters are used hereafter, we use the signature of the file sharing service, `void uploadFiles(byte[][] inputFiles, int[] fileTypes)`. In the signature, `inputFiles` contains the byte arrays of multiple files for upload and `fileTypes` is an array indicating the corresponding types of files represented by constants, e.g., `FileType.COMPRESSED` or `FileTypes.TEXT` (i.e., uncompressed).

4.1.1 Data-Dependent Control Flow: Use of Compression Component for Multiple Files

The BusinessLogic subcomponent compresses noncompressed files only. Thus, the number of calls of the Compress component depends on the number of uncompressed files (`FileType.TEXT`) uploaded. Through genetic programming, the correct solution for the number of calls of the COMPRESSION component was found:

$$inputFiles.length - fileTypes.COUNT(FileType.COMPRESSED),$$

where `SUM` is aggregated data from the monitoring step specifying the number of uncompressed files. The reverse-engineered dependency exactly fits the expected one. The search time was less than one second, meaning that the fitness function indicated an optimal solution being found after that time, leading genetic programming to terminate.

4.1.2 When to Use LargeFileStorage or SmallFileStorage

A set of 11 different input files (different file types, different file sizes) was used as test data for answering this question. To have a unique control flow trace, monitoring data from uploads with just one file is analyzed for such questions regarding path selections in the control flow.

Genetic programming found an optimal solution for describing the control flow: If the input 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. For SMALLFILESTORAGE, an opposite dependency was found. The genetic programming run took less than five seconds to obtain this result.

For LARGEFILESTORAGE, the following expression resulted:

$$if(hashingInputFiles.length > 200{,}000)\{..\}$$

where `hashingInputFiles` is the size of an individual uploaded file passed to HASHING. In this case, genetic programming selected the correct input variable (`hashingInputFiles` instead of, for example, `inputFiles`, or one of the other measuring data points).

Through *static bytecode analysis*, a gene representing the constant "200,000" (and others extracted from the bytecode) was available to genetic programming. This resulted in faster convergence and increased precision compared to genetic programming without a previous static bytecode analysis. Thus, the use of static bytecode analysis can yield better results with less monitored data.

For cases where static bytecode analysis was *not* performed, the constant "200,000" was not always identified correctly due to the limited number of input files. No files with 199,999 and 200,000 bytes of size were part of the input data. Yet the recovered function did not contradict the monitoring data so that genetic programming could not detect an error. Adding additional files would also have overcome this issue for cases without static bytecode analysis.

In an additional run of genetic programming, we used monitoring data that was disturbed to test *robustness* of the approach. Here, not every write led to a call of a storage component because the file already existed in the database (1 out of 11 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. The optimal solution could still be found, but it required nearly 20 seconds (on average) to find the solution. In this case, the confidence in the correctness (evaluated by the "fitness function") of the result decreased as not every single date measured could also be exactly predicted. The average behavioral 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.

4.1.3 SingleFileAnalysis—Size of Files Passed to the Compress Component

The size of single files passed to the compression service was learned in another run of genetic programming. The input consisted of a series of 11 input files again. The dependency recovered was

$$inputFiles[x].length \cdot fileTypes[x],$$

where `inputFiles[x].length` is the size of an individual uploaded file and `fileTypes[x]` the integer encoded type of a file, where 0 represented noncompressed files. Within a search time of less than one second, the optimal solution could always be found.

4.1.4 Estimation of the Compression Ratio

In addition to applying the approach for the BUSINESS LOGIC component, we used it for estimating the compression ratio of the COMPRESSION component. BUSINESS LOGIC itself relies on the compression ratio of this component, as the size of noncompressed input files that are passed to the storage components is larger than the ones being compressed by COMPRESSION beforehand. The compression ratio of a COMPRESSION algorithm strongly depends on the data characteristics (e.g., entropy, used encoding)—no optimal solution exists to describe the compression ratio. Therefore, genetic programming produced a large variety of approximations of the compression ratio. A good approximation, found after 30 seconds, had the following form (here, for the LZW implementation):

$$0.9 \cdot 0.5 \cdot (X3 - (0.9 \cdot 0.5 \cdot (X3 - (0.9 \cdot (0.9 \cdot 0.5 \cdot X3) \cdot 1.0)))),$$

where $X3$ is the size of the file input for the COMPRESSION component, which was found (by the genetic programming) to be significant; for SPECjvm2008, comparable results were produced.

	ALOAD	ARRAYLENGTH	ANEWARRAY	BALOAD	ICONST_0	IF_ICMPLT	IINC	ILOAD	ISTORE
P1	3,81	1,70	200,93	1,63	1,63	2,45	3,00	2,18	4,29
P2	4,09	1,90	226,16	1,63	1,63	3,00	3,54	2,18	5,45

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

4.1.5 Learning of Bytecode Instruction Counts

For the COMPRESSION component, bytecode counts were estimated as functions of counts on input data. As the behavior of the data compression algorithm strongly depends on inner characteristics of the compressed data (and not only on its size and type), a fully precise (no error) learned function cannot be expected in the general case. Optimal solutions were found only for a few bytecode counts. Bytecode count approximations created by genetic programming still are good estimators. In 98 percent of the cases, the count approximations found by genetic programming outperform MARS approximations. The high quality of performance predictions (less than 30 percent error; see below) also indicate good approximations since the count approximations are used in the performance prediction.

An optimal solution was, for example, found in less than one second for the bytecode instruction `ICONST_M1` (here, SPEC compress):

$$X1 + X1 + 3.0,$$

where *X1* is the monitored size of each input file.

For a more complex case, such as the bytecode `IALOAD` (here, SPEC compress), after evolving 700 generations, the following approximation was found

$$\begin{aligned} &20 \cdot X1 + 5 \cdot X2 + 30285.9 + IF(X2 > 0.9)\{ \\ &\quad 4 \cdot X2 + 13 \cdot X1 + 160000.6 + \\ &\quad IF(X1 > 0.9)\{X1\}ELSE\{X2 + 2 \cdot X1\} \\ &\}ELSE\{100000.6040835\}, \end{aligned}$$

where *X1* is the monitored size of each input file and *X2* a flag showing whether the input file was already compressed ($X2 = 1$) or not ($X2 = 0$). It can be seen that the estimation of bytecode counts for nontrivial components in general is nonlinear, which requires expression power beyond multilinear functions for estimating counts.

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

4.1.6 Findings

Manual specification of dependencies tends to be error-prone since dependencies can cover multiple classes for which data and control flow must be tracked. In the investigated example, all dependencies (11 in total) identified by manual reverse engineering by a human were also found by our automated genetic search-based approach. Hence, we consider our approach and genetic programming to be well applicable for reverse engineering of parameterizations from business components comparable to BUSINESSLOGIC. Additionally, bytecode counts were estimated in a way that predictions based on it result in little error (see section 4.3 for more details on overall prediction results).

The automation of our approach leads to less effort for reverse engineering and makes it much faster: Compared to manual reverse engineering, which took more than 24 hours (cf. case study in [23]), running the reverse engineering tools of our approach takes only about 40 minutes, including the manual setup.

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 1.83 GHz each), 1.5 GB of main memory, and Windows XP Pro with Sun JDK 1.6.0_06.

All microbenchmarks have been repeated systematically and the median measurement has been taken for each microbenchmark. Fig. 6 is an excerpt of the results of our microbenchmarking on P1 and P2. It lists execution durations of nine bytecode instructions among those with the 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 during microbenchmark execution, despite the fact that JIT was enabled in the JVM. Hence, prediction on the basis of such benchmarking must account for the "speedup" effect of JIT optimizations 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 the aforementioned space limitations, we cannot describe the results of API microbenchmarks here. To simplify working with the Java API, we are currently working toward 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. It must be pointed out that for business information systems with non-real-time hardware and software environments, one is usually not interested in the worst-case execution time but in average values (opposed to embedded systems). Imagine the worst-case execution time in a Windows environment, which might be several minutes, while that execution time rather unlikely in practice.

For explaining the performance prediction, we use the LZW COMPRESS implementation. First, for source platform P1, we predict the duration of compressing a text file (randomly chosen) with a size of 25 KB and compress on the basis of bytecode microbenchmarks, yielding 1,605 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 nine additional, different files on platform P2 (these files varied in contents, size, and achievable compression rate). For each of these nine files, the prediction accuracy was within 10 percent (except one outlier which was still predicted with 30 percent 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 36,747 bytes of data on P2 was predicted to 1.71 ms while 1.69 ms were measured (cf. Fig. 7), i.e., with <2 percent error. Similar accuracy for predicting hashing duration is obtained for other file sizes and types.

The total upload process using LZW COMPRESS for the above 25 KB text file on P2 was predicted to take 115 ms, and 123 ms were measured. Upload of a 37 KB JPEG (i.e., already compressed) file took 1.82 ms, while 1.79 ms were predicted. For all files used in the case study, the prediction

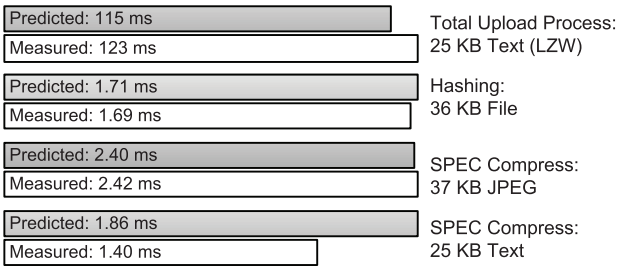

Fig. 7. Selected prediction results.

of the entire upload process for one file had an average deviation of <15 percent.

When using the SPEC compress component, the average prediction error was less than 30 percent; which also holds for the total upload process. For example, for the 37 KB JPEG from Fig. 7, the measurement was 2.42 ms and the prediction 2.40 ms; for the 25 KB text file, 1.40 ms were measured and 1.86 ms predicted. A possible explanation for the worse results with SPEC compress is the intensive use of memory which was not present in LZW COMPRESS implementation.

Ultimately, our bytecode-based prediction method can deal with all four factors discussed in Section 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 behavior of concurrent requests w.r.t. bytecode execution.

4.4 Discussion

SPECjvm2008 compress, which is part of our evaluation, is a representative benchmark developed by and in cooperation with industry which intentionally covers a broad range of industrial requirements. BUSINESSLOGIC of the file sharing application is comprised of typical business information system code with limited complexity. SPEC compress has challenging algorithmic complexity. Hence, we conclude that external validity of the case study is present as long as the applications under study are comparable to either the algorithmic complex or workflow intensive part (BUSINESS LOGIC) of our case study. This applies to a broad range of other real-world applications.

In the genetic search step, heavily disturbed results (i.e., those having causes not visible at the interface level) lead to a decreased convergence speed and a lower probability of finding a good solution. Through the usage of MARS, results never drop below the precision of that statistical approach. Genetic programming is robust against limited input values. When, for example, using only 15 out of 33 input files for the COMPRESSION component, bytecode count estimations were off less than 15 percent compared to the estimations based on the full input. Only eight input files were sufficient for correctly reverse engineering the control flow of BUSINESSLOGIC. The only requirement is

that the properties of these input files cover the input parameter space (binary and text files of varying size, presence in the copyright database).

Our approach can be completely automated, such that the behavioral models can be reverse engineered from code without user interaction, given that test cases and a testbed are available. The benchmarking step can run autonomously. For the prediction step, users can use graphical editors to specify which scenarios they like to investigate. This makes the approach als applicableo for software architects which have little experience with the approach.

5 Limitations and Assumptions

For the monitoring step, we assume that representative input parameter values can be provided, for example, by a test driver. This workload has to be representative for both the current and the planned input parameter space of the component. For running systems, these data can be obtained using runtime monitoring. Since our approach does not capture any timing information during the monitoring step, basically every approach capable of generating test cases covering the input parameter space of methods is suitable (overview in [30]). Our approach requires no special "performance test cases." Otherwise, a domain expert selects or specifies the scenarios and corresponding workloads.

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. The parts **A** and **B** in Fig. 1 are not integrated for an automated performance prediction.

For the monitoring step, we assume that all input and output data are composed from primitive types or general collection types like `List` for which our heuristics identifying important characteristics can be applied. In more elaborate cases, a domain expert can specify important data characteristics manually to improve the monitoring database.

Our current approach focuses on business information systems but could also be extended to embedded systems. For embedded systems, enhancements of the prediction step would be required to support real-time systems. Currently, our approach supports only components for which Java bytecode is available. Generally, it can be easily extended to any application based on a intermediate language (e.g., Microsoft Intermediate Language).

6 Conclusions

In this paper, we have presented a reverse engineering approach that combines genetic search, static and dynamic analysis with benchmarking to predict the performance of applications. By explicitly separating the platform-independent parametric performance model of components from the performance of execution platforms, the approach enables prediction for changing execution platforms, different usage contexts, and changing assembly contexts.

The advantage of separating the performance 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. The approach requires no a priori knowledge on the components under investigation.

In the described approach, dynamic analysis is used for monitoring data at the component interface level and to count executed bytecode instructions and method calls at runtime. Static analysis is used to find constants in component bytecode. Results from static and dynamic analysis are then fed into genetic programming to reverse engineer the platform-independent parametric performance model. Genetic programming identifies data flow and control flow parameterized over input data. Here, we point out how genetic programming can be applied for reverse engineering of parametric performance models. Next, bytecode instructions and methods are benchmarked to obtain their performance values for a certain platform.

We successfully evaluated the integrated reverse engineering and performance prediction approach using a case study for the Java implementation of a file sharing application which includes the SPECjvm2008 compress component. The evaluation shows that the approach yields accurate prediction results for 1) different execution platforms and 2) 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 30 percent for all considered usage contexts (i.e., uploaded files), assembly contexts (i.e., SPECjvm2008 compress versus LZW COMPRESS), and deployment context (P1 versus P2). The average accuracy is therefore very good.

For our future work, we plan to automate the entire approach and to merge bytecode counting with data monitoring and recording. The manual execution of the approach took about five hours for the case study.

Acknowledgments

The authors thank the Software Design and Quality group and Martin Krogmann for supporting the development of BYCOUNTER.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Experiments in Cost Analysis of Java Bytecode," *Electronic Notes in Theoretical Computer Science,* vol. 190, no. 1, pp. 67-83, 2007.

[2] S. Becker, J. Happe, and H. Koziolek, "Putting Components Into Context—Supporting QoS-Predictions with an Explicit Context Model," *Proc. Workshop Component Oriented Programming,* R. Reussner, C. Szyperski, and W. Weck, eds., June 2006.

[3] S. Becker, H. Koziolek, and R. Reussner, "The Palladio Component Model for Model-Driven Performance Prediction," *J. Systems and Software,* vol. 82, pp. 3-22, 2009.

[4] A. Bertolino and R. Mirandola, "CB-SPE Tool: Putting Component-Based Performance Engineering into Practice," *Proc. Seventh Int'l Symp. Component-Based Software Eng.,* I. Crnkovic, J.A. Stafford, H.W. Schmidt, and K.C. Wallnau, eds., pp. 233-248, 2004.

[5] W. Binder and J. Hulaas, "Flexible and Efficient Measurement of Dynamic Bytecode Metrics," *Proc. Fifth Int'l Conf. Generative Programming and Component Eng.,* pp. 171-180, 2006.

[6] W. Binder and J. Hulaas, "Using Bytecode Instruction Counting as Portable CPU Consumption Metric," *Electronic Notes in Theoretical Computer Science,* vol. 153, no. 2, pp. 57-77, 2006.

[7] E. Bondarev, P. de With, M. Chaudron, and J. Musken, "Modelling of Input-Parameter Dependency for Performance Predictions of Component-Based Embedded Systems," *Proc. 31st EUROMICRO Conf. Software Eng. and Advanced Applications,* 2005.

[8] L.C. Briand, Y. Labiche, and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE Trans. Software Eng.,* vol. 32, no. 9, pp. 642-663, Sept. 2006.

[9] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A Code Manipulation Tool to Implement Adaptable Systems," Adaptable and Extensible Component Systems, 2002.

[10] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proc. Eighth USENIX Symp. Operating Systems Design and Implementation,* Dec. 2008.

[11] M. Courtois and C.M. Woodside, "Using Regression Splines for Software Performance Analysis," *Proc. Second Int'l Workshop Software and Performance,* pp. 105-114, Sept. 2000.

[12] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods.* Cambridge Univ. Press, 2000.

[13] J. Donnell, "Java Performance Profiling Using the VTune Performance Analyzer," 2004.

[14] D. Doval, S. Mancoridis, and B.S. Mitchell, "Automatic Clustering of Software Systems Using a Genetic Algorithm," *Proc. Conf. Software Technology and Eng. Practice,* pp. 73-81, 1999.

[15] M.D. Ernst et al., "The Daikon System for Dynamic Detection of Likely Invariants," *Science of Computer Programming,* vol. 69, nos. 1-3, pp. 35-45, Dec. 2007.

[16] J.H. Friedman, "Multivariate Adaptive Regression Splines," *The Annals of Statistics,* vol. 19, no. 1, pp. 1-141, 1991.

[17] G.H. Golub, M. Heath, and G. Wahba, "Generalized Cross-Validation as a Method for Choosing a Good Ridge Parameter," *Technometrics,* vol. 21, pp. 215-223, May 1979.

[18] M. Harman, "The Current State and Future of Search Based Software Engineering," *Proc. Future of Software Eng.,* pp. 342-357, May 2007.

[19] C. Herder and J.J. Dujmovic, "Frequency Analysis and Timing of Java Bytecodes," Technical Report SFSU-CS-TR-00.02, Computer Science Dept., San Francisco State Univ., 2000.

[20] C.E. Hrischuk, C.M. Woodside, and J.A. Rolia, "Trace-Based Load Characterization for Generating Performance Software Models," *IEEE Trans. Software Eng.,* vol. 25, no. 1, pp. 122-135, Jan./Feb. 1999.

[21] E.Y.-S. Hu, A.J. Wellings, and G. Bernat, "Deriving Java Virtual Machine Timing Models for Portable Worst-Case Execution Time Analysis," *Proc. OTM Workshops, On The Move to Meaningful Internet Systems,* R. Meersman and Z. Tari, eds., pp. 411-424, 2003.

[22] T. Israr, M. Woodside, and G. Franks, "Interaction Tree Algorithms to Extract Effective Architecture and Layered Performance Models from Traces," *J. Systems and Software,* vol. 80, no. 4, pp. 474-492, Apr. 2007.

[23] T. Kappler, H. Koziolek, K. Krogmann, and R.H. Reussner, "Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering," *Proc. Software Eng.,* pp. 140-154, Feb. 2008.

[24] J.R. Koza, *Genetic Programming—On the Programming of Computers by Means of Natural Selection,* third ed., MIT Press, 1993.

[25] K. Krogmann and R.H. Reussner, "Palladio: Prediction of Performance Properties," *The Common Component Modeling Example,* pp. 297-326, Springer-Verlag, 2008.

[26] M. Kuperberg and S. Becker, "Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs," *Proc. Workshop Component Oriented Programming,* R. Reussner, C. Czyperski, and W. Weck, eds., July 2007.

[27] M. Kuperberg, K. Krogmann, and R. Reussner, "Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models," *Proc. 11th Int'l Symp. Component-Based Software Eng.,* pp. 48-63, Oct. 2008.

[28] M. Kuperberg, M. Krogmann, and R. Reussner, "ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations," *Proc. Third Int'l Workshop Bytecode Semantics, Verification, Analysis and Transformation,* 2008.

[29] J. Lambert and J.F. Power, "Platform Independent Timing of Java Virtual Machine Bytecode Instructions," *Proc. Workshop Quantitative Aspects of Programming Languages,* Mar. 2008.

[30] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software Testing, Verification, and Reliability,* vol. 14, no. 2, pp. 105-156, June 2004.

[31] K. Meffert, "JGAP—Java Genetic Algorithms Package," http://jgap.sourceforge.net/, 2008.

[32] M. Meyerhöfer and K. Meyer-Wegener, "Estimating Non-Functional Properties of Component-Based Software Based on Resource Consumption," *Electronic Notes in Theoretical Computer Science,* vol. 114, pp. 25-45, 2005.

[33] B.S. Mitchell, S. Mancoridis, and M. Traverso, "Search Based Reverse Engineering," *Proc. 14th Int'l Conf. Software Eng. and Knowledge Eng.,* pp. 431-438, 2002.

[34] C.U. Smith, *Performance Engineering of Software Systems.* Addison-Wesley, 1990.

[35] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming,* second ed., ACM Press and Addison-Wesley, 2002.

[36] M. Woodside, D.C. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by Unified Model Analysis (PUMA)," *Proc. Fifth Int'l Workshop Software and Performance,* pp. 1-12, 2005.

[37] X. Zhang and M. Seltzer, "HBench:Java: An Application-Specific Benchmarking Framework for Java Virtual Machines," *Proc. ACM Conf. Java Grande,* pp. 62-70, 2000.

[38] T. Zheng, C.M. Woodside, and M. Litoiu, "Performance Model Estimation and Tracking Using Optimal Filters," *IEEE Trans. Software Eng.,* vol. 34, no. 3, pp. 391-406, May/June 2008.

Klaus Krogmann received the diploma in computer science. He is a researcher in software engineering at the Karlsruhe Institute of Technology (KIT), Germany. His research interests include component-based software architectures, software quality, reverse engineering, search-based software engineering, and model-driven development.

Michael Kuperberg received the diploma in computer science. He is a researcher in software engineering at the Karlsruhe Institute of Technology (KIT), Germany. His research interests include bytecode-based performance prediction, benchmarking, and modeling of execution environments. He is a member of the IEEE.

Ralf Reussner received the PhD degree from the University of Karlsruhe. He is a full professor of software engineering at the Karlsruhe Institute of Technology (KIT) and director at the IT Research Center in Karlsruhe (FZI). His research interests include software components, software architecture, and model-based quality prediction. He was with DSTC Pty Ltd, Melbourne, Australia. From 2003 to 2006, he held a junior professorship at the University of Oldenburg, Germany. He is a member of the IEEE.