Automated Benchmarking of Java APIs

Michael Kuperberg[1], Fouad Omri[1], Ralf Reussner[1]

[1]Chair Software Design & Quality, Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, 76131 Karlsruhe, Germany
{michael.kuperberg|fouad.omri|ralf.reussner}@kit.edu

Abstract: Performance is an extra-functional property of software systems which is often critical for achieving sufficient scalability or efficient resource utilisation. As many applications are built using application programmer interfaces (APIs) of execution platforms and external components, the performance of the used API implementations has a strong impact on the performance of the application itself. Yet the sheer size and complexity of today's APIs make it hard to manually benchmark them, while many semantical constraints and requirements (on method parameters, etc.) make it complicated to automate the creation of API benchmarks. Benchmarking the whole API is necessary since it is in the majority of the cases hard to exactly specify which parts of the API would be used by a given application. Additionally, modern execution platforms such as the Java Virtual Machine perform extensive nondeterministic runtime optimisations, which need to be considered and quantified for realistic benchmarking. In this paper, we present an automated solution for benchmarking any large APIs that are written in the Java programming language, not just the Java Platform API. Our implementation induces the optimisations of the Just-In-Time compiler to obtain realistic benchmarking results. We evaluate the approach on a large subset of the Java Platform API exposed by the base libraries of the Java Virtual Machine.

1 Introduction

Performance (e.g. response time) is an important extra-functional property of software systems, and is one of the key properties perceived by the users. Performance influences further software qualities, such as scalability or efficiency of resource usage. Addressing the performance of an application should not be postponed to the end of the implementation phase, because the cost of fixing performance issues increases as the application grows. While architecture-based performance prediction approaches such as Palladio [BKR09] exist, the factual performance is determined by the implementation of an application.

Therefore, software engineers should address software performance during the entire implementation phase. As many applications are built using application programmer interfaces (APIs) of execution platforms and external components, the performance of these APIs has a strong impact on the performance of the application itself. When an application itself offers APIs, their performance has to be benchmarked and controlled as well.

While profiling tools such as VTune [Int09] help with finding performance issues and "hot spots", they are not suitable for performance testing of entire APIs created by software engineers. Additionally, many applications target platform-independent environments such

as Java Virtual Machine (JVM) or .NET Common Language Runtime (CLR).

These environments provide a large set of utility classes, e.g. the Java Platform API includes thousands of methods. A given functionality is often provided by several alternative API methods, but there is no performance specification to help choose between them. Altogether, benchmarking both the required and the provided APIs completely by hand is an unrealistic task: for example, the Java platform API has several thousands of methods.

Thus, to benchmark methods of APIs, developers and researchers often manually create microbenchmarks that cover only tiny portions of the APIs (e.g. 30 "popular" methods [ZS00]). Also, the statistical impact of measurements error is ignored and the developers must manually adapt their (micro)benchmarks when the API changes. Additionally, benchmarking API methods to quantify their performance is a task that is hard to automate because of many semantical constraints and requirements (on method parameters, type inheritance, etc.). To obtain realistic results, extensive runtime optimisations such as Just-in-Time compilation (JIT) that are provided by the JVM and the CLR need to be induced during benchmarking and quantified. Thus, there exists no standard automated API benchmarking tool or strategy, even for a particular language such as Java.

The contribution of this paper is an automated, modular solution to create benchmarks for very large black-box APIs that are written in the Java programming language. Our solution is implemented to induce the optimisations of the Java JIT compiler, and quantifies its effects on the performance of benchmarked methods. The execution of benchmarks is also automated, and when the API or its implementation change, the benchmarks can be regenerated quickly, e.g. to be used for regression benchmarking. Our solution is called APIBENCHJ and it requires neither the source code of the API, nor a formal model of method input parameters. Also, in contrast to black-box *functional* testing, APIBENCHJ is not searching the parameter space for (unexpected) runtime exceptions and errors - instead, it uses existing techniques such as heuristic parameter generation [KOR09] to *avoid* errors and to find meaningful (legal and representative) method parameters for benchmarking.

We evaluate the presented framework by automatically benchmarking 1458 methods of the frequently-used, significant Java Platform API packages `java.util` and `java.lang`. Afterwards, we compare APIBENCHJ benchmarking results to results of fully manual benchmarking. In this context, we discuss how APIBENCHJ decreases the need for manual work during API benchmarking, and also how APIBENCHJ should be extended to better quantify parametric performance dependencies.

The remainder of the paper is organised as follows: in Sec. 2, we outline the foundations of our approach. In Sec. 3, related work is presented and compared to APIBENCHJ. An overview of APIBENCHJ is given in Sec. 4, while the details of the implementation are described in Sec. 5. Sec. 6 describes our evaluation. In Sec. 7, we outline the assumptions and limitations of our approach. The paper concludes with Sec. 8.

2 Foundations

The implementation of an API method in Java is usually provided by a library, i.e. as bytecode of Java classes (we do not consider native methods or web services here). Benchmarking a method means systematically measuring its response time as it is executed.

To execute a method, it must be called by some custom-written Java class, i.e. the bytecode of such a suitable caller class must be loaded and executed by the JVM (in addition to the callee bytecode). There are three different techniques for caller construction: (1) using the *Java Reflection API* to dynamically call methods at runtime, (2) using *code generation* to create caller source code that is compiled to executable caller classes, and (3) using *bytecode engineering* techniques to directly construct the binary Java classes that call the benchmarked methods

All these three techniques need to be examined with respect to their scalability and their impact on the behaviour of the JVM (just-in-time compilation, etc.) and on the measurement itself (e.g., whether the overhead of Java Reflection API usage can be clearly separated from the execution duration of the benchmarked method). The measurements have to be carried out with respect to statistical validity, which is influenced by the resolution of the used timer and the duration of the benchmarked method. JIT compiler optimisations can cause significant problems when benchmarking. The benchmarked piece of code is for example usually eliminated after JIT optimisations are performed by eliminating Dead-Code. The constant folding algorithm implemented in JIT can identify a simplification possibility by replacing successive calls to an arithmetic operation by a constant node in the dependency graph of the JIT compiler [**?**]. In order to avoid constant folding during benchmarking, the JIT compiler should not identify input parameters of the benchmarked methods as constants. Such challenges has to be met in order to avoid misleading benchmarking results.

During benchmarking, in order to execute a method that has one or several input parameters, these parameters *must* be supplied by the caller and they must be *appropriate*. For example, a `IndexOutOfBoundsException` is thrown for `String.substring(int beginIndex)` if `beginIndex < 0` or if `beginIndex > string.length()` since these parameters are inappropriate. In general, method parameters can be of several types: primitive types (`int`, `long` etc.), object types that are 'boxed' versions of primitive types (e.g. `Integer`), array types (e.g. `int[]` or `Object[]`) and finally of general object or interface types (e.g. `StringBuffer`, `List`, etc.)

For primitive parameter types, often only specific values are accepted (cf. `String.substring` method above), and if a 'wrong' parameter value is used, the invoked method will throw a runtime exception. Very often, such exceptions do not appear in method signatures, and are also undocumented in the API documentation. Even for this single integer parameter, randomly guessing a value (until no runtime exception is thrown) is not recommended: the parameter can assume 2^{32} different values.

For parameters of types extending `java.lang.Object`, additional challenges arise [KOR09]. Unfortunately, almost all APIs provide no formal specification of parameter value information, and also provide no suitable (functional) test suites or annotations from which parameters suitable for benchmarking could be extracted.

In our previous work [KOR09], we have addressed the issue of automated parameter generation using novel heuristics, and have successfully evaluated it for several Java Platform API packages. By devising a modular approach, we have also proposed pluggable alternatives to these heuristics, e.g. recording parameters used in functional tests, or using values specified by humans. Therefore, in this paper, we concentrate on the actual generation, execution and evaluation of the API benchmarks, and assume the parameters as given.

An API can cover a vast range of functionalities, ranging from simple data operations and analysis up to network and database access, security-related settings, hardware access, and even system settings. Hence, the first consideration in the context of automated benchmarking is to set the limits of what is admissible for automated benchmarking.

For example, an automated approach should be barred from benchmarking the method `java.lang.System.exit`, which shuts down the Java Virtual Machine. Likewise, benchmarking the Java Database Connectivity (JDBC) API would report the performance of accessed database, not the performance of the JDBC API, and it is likely to induce damage on database data. Thus, JDBC as part of the Java platform API is an example of an API part that should be excluded from automated benchmarking. APIBENCHJ handles exclusion using patterns that can be specified by its users. From the elements of an API that are *allowed* for automated benchmarking, the only two element types that can be executed and measured are non-abstract methods (both static and non-static) and constructors (which are represented in bytecode as special methods). Opposed to that, neither class fields nor interface methods (which are unimplemented) can be benchmarked.

3 Related Work

A considerable number of benchmarks for Java has been developed which differ in target JVM type: there are benchmarks for Java SE (e.g. [Cor08]), Java EE (e.g. [Sta08]), Java ME [Pie], etc. Java benchmarks also differ in scope (performance comparison of algorithms vs. performance comparison of a platform), size/complexity, coverage, and actuality; [BSW+00] provides an extensive but incomplete list.

While *comparative* benchmarking yields "performance *proportions*" or "performance *ordering*" of alternatives, our work needs to yield precise quantitative metrics (execution duration), parameterised over the input parameters of methods. Quantitative method benchmarking was done in HBench:Java [ZS00], where Zhang and Seltzer have selected and *manually* benchmarked only 30 API methods, but they did not consider or quantify the JITting of the JVM.

In the following, we do not discuss Java EE benchmarks because they target high-level functionality such as persistence, which is used transparently provided by the application servers using dependency injection, rather than direct API method invocations.

For Java SE, SPECjvm2008 [Cor08] is a popular benchmark suite for comparing the performance of Java SE Runtime Environments. It contains 10 small applications and benchmarks focusing on core Java functionality, yet the granularity of SPECjvm2008 is large in comparison to API methods benchmarking of the benchmark we present in this paper. Additionally, the Java Platform API coverage of SPECjvm2008 is unknown, and the

performance of individual API methods cannot be derived from SPECjvm2008 results.

Other Java SE benchmarks such as Linpack [lin07] or SciMark [sci07] are concerned with performance of both numeric and non-numeric computational "kernels" such as Monte Carlo integration, or Sparse Matrix multiplication. Some Java SE benchmarks (e.g. from JavaWorld [Bel97]) focus on highlighting the differences between Java environments, determining the performance of high-level constructs such as loops, arithmetic operations, exception handling and so on. The UCSD Benchmarks for Java [GP] consist of a set of low-level benchmarks that examine exception throwing, thread switching and garbage collection. All of these benchmarks have in common that they neither attempt to benchmark atomic API methods nor benchmark *any* API as a whole (most of them benchmark mathematical kernels or a few Java platform methods). Additionally, they do not consider runtime effects of JVM optimisations (e.g. JIT) systematically and they have not been designed to support non-comparative performance evaluation or prediction.

4 Overview of the APIBENCHJ Framework

In this section, we give a high-level overview of APIBENCHJ, while relevant details of its implementation are left to Sec. 5. The output for APIBENCHJ is a platform-independent suite of executable microbenchmarks for the considered API which runs on any Java SE JVM. Fig. 1 summarises the main steps of control flow in APIBENCHJ, and we explain it in the following. The steps highlighted with bold boxes form the focus of this paper, and are presented in more detail.

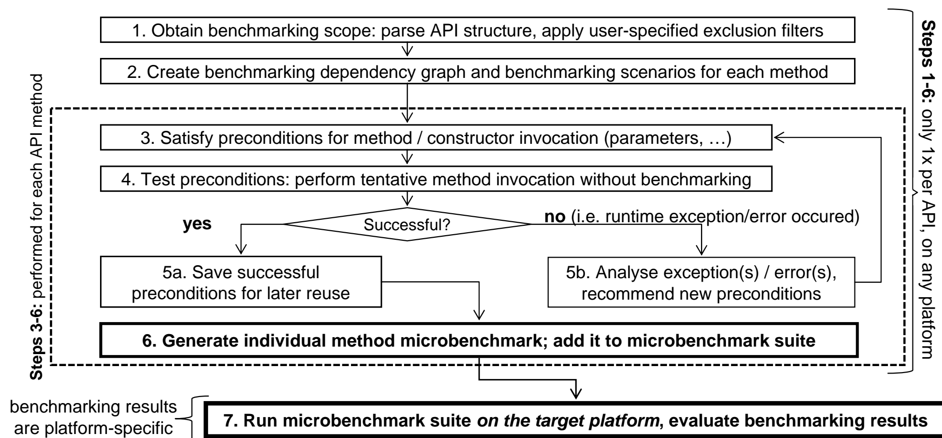

Figure 1: APIBENCHJ : overview of automated API benchmarking

Step 1 starts with *parsing and storing the API structure* to identify the relations between API elements, e.g. inheritance relations and package structure. APIBENCHJ can operate directly on bytecode and does not requires source code, i.e. it is suitable for black-box APIs whose implementation is not exposed. The Java platform and its Reflection API

do not provide sufficient functionality for this task, e.g. one cannot programmatically retrieve all implementers of an interface. Thus, APIBENCHJ has its additional tools to parse the API structure using the bytecode classfiles of its implementation. Step 1 also applies *user-specified exclusion filters* to exclude entities that must not be benchmarked automatically, as described in Sec. 2. The exclusion filters are specified beforehand by users (i.e. APIBENCHJ does not try to exclude such entities itself). Filters can be package names, classes implementing a specific interface or extending a given class, etc.

Step 2 in Fig. 1 creates *benchmarking scenario(s)* for each method. Scenarios describe the *requirements* for benchmarking, e.g. which parameters are needed and which classes must be instantiated *before* the considered method can be benchmarked. Actual runtime *values and objects* are created/instantiated later, in steps 3 through 7. In APIBENCHJ, a scenario consists of *preconditions*, the actual *benchmarked operation* and the *postconditions* for a method invocation. At the beginning, step 2 creates a *benchmarking dependency graph*, which holds relations such as "`String.contentEquals` must be preceded by initialisation of a `String` instance", or "the constructor `String()` has no preconditions". As several constructors for `String` and `StringBuffer` exist, several scenarios can be created which differ in the choice of constructors used to satisfy preconditions, and which allow the quantitative comparison of these choices. Step 2 can also compute metrics for the complexity of benchmarking methods, so that step 3 can start with the methods having lowest complexity.

Step 3 starts with *trying to satisfy the precondition requirements* of a benchmarking scenario. Satisfying benchmarking requirements from Step 2 means generating appropriate method parameters, invocation targets, etc. A precondition may have its own preconditions, which APIBENCHJ must then satisfy first. As discussed in Sections 1 and 2 as well as in our previous work [KOR09], automating of these tasks is challenging due to runtime exceptions and the complexity of the Java type hierarchy/polymorphism. APIBENCHJ incorporates a combined approach to this challenge by providing a plug-in mechanism with different precondition sources which can be ranked by their usefulness. For example, *manual specification* has a higher rank than *heuristic search*, with *directed brute-force search* having the lowest ranking of the three. If, for example, APIBENCHJ finds that no manual plug-in exists for a precondition type, it could choose the heuristic search plug-in described in [KOR09]. The generated preconditions are likely to provoke runtime exceptions, as discussed in Sec. 2. Hence, before they are accepted as benchmarking-ready, they must be tested.

Step 4 performs a *tentative method invocation* to test that using the generated preconditions does not lead to runtime exceptions (if such an exceptions occurs APIBENCHJ proceeds with **step 5b**). The error handler in step 5b triggers a new attempt to satisfy preconditions of the considered benchmarking scenario, or gives up the scenario if a repetition threshold is surpassed (this threshold serves to prevent infinite or overly long occupation with one scenario, especially if using brute-force parameter search).

Step 5a is entered if the tentative invocation succeeds, and the information on successful precondition values are internally saved for future reuse. The saved information may be a pointer to the successful heuristic, pointer to a code section that has been manually specified by a human, or a serialised parameter value. Due to space limitations, we do not

describe the design rationale and the implementation of the mechanisms in step 5a/5b here but refer the reader to [KOR09].

Step 6 generates an executable microbenchmark for the considered scenario, using successfully tested precondition values. The generated microbenchmark implementation explicitly addresses measurement details such as timer resolution [KKR09b], JVM optimisations, etc. The *execution* of the resulting microbenchmark does not require the APIBENCHJ infrastructure that implements steps 1 through 6 - each microbenchmark is a portable Java class that forms a part of the final *microbenchmark suite*. The microbenchmark suite includes the microbenchmarks plus additional infrastructure for collecting microbenchmark results and evaluating them. In Section 7, we discuss the parameter representability of the generated benchmarks.

In the next section, we describe the implementation of step 6 as it is the most complex and interesting part of the current prototypic implementation of APIBENCHJ.

5 Implementation of APIBENCHJ

In this section, we assume that appropriate method parameters are known, and it is known how to obtain the invocation targets for non-static methods (see steps 1-5 in Sec. 4). Using the results of [KKR09b], we know the accuracy and invocation cost of the timer method used for measurements, and thus can compute the number of measurements needed for a given confidence level (see [Omr07] for details).

The remaining steps 6 (generating individual microbenchmarks) and 7 (executing the benchmarks) are discussed in this section. First, we discuss the runtime JVM optimisations and how they are addressed (Sec. 5.1), followed by the discussion in Sec. 5.2 on why bytecode engineering is used to construct the microbenchmarks.

5.1 JIT and other JVM Runtime Optimisations

Java bytecode is platform-independent, but it is executed using interpretation which is significantly slower than execution of equivalent native code. Therefore, modern JVMs monitor the execution of bytecode to find out which methods are executed frequently and are computationally intensive (“hot”), and optimise these methods.

The most significant optimisation is Just-in-Time compilation (JIT), which translates the hot method(s) into native methods *on the fly*, parallel to the running interpretation of the “hot” method(s). To make benchmarked methods “hot” and eligible for JITting, they must be executed a significant number of times (10,000 and more, depending on the JIT compiler), before the actual measurements start. JIT optimisations lead to speedups surpassing one order of magnitude [KKR08], and an automated benchmarking approach has to obtain measurements for the unoptimised and the optimised execution as both are relevant.

Different objectives lead to different JITting strategies, e.g. the Sun Microsystems Server

JIT Compiler spends more initial effort on optimisations because it assumes long-running applications, while the Client JIT Compiler is geared towards faster startup times. We have observed that the Sun Server JIT Compiler performs multi-stage JITting, where a "hot" method may be repeatedly JIT-compiled to achieve even higher speedup if it is detected that the method is even "hotter" that originally judged.

Therefore, the benchmarks generated by APIBENCHJ can be configured with the *platform-specific* threshold number of executions ("warmup") after which a method is considered as "hot" and JITted by that platform's JIT compiler. To achieve it, we have implemented a calibrator which uses the `-XX:+PrintCompilation` JVM flag to find out a platform's calibration threshold, which is then passed to the generated benchmarks.

We must ensure that JIT does not "optimise away" the benchmarked operations, which it can do if a method call has no effect. To have *any* visible functional effect, a method either returns a value, changes the value(s) of its input parameter(s), or has side effects not visible in its signature. These effects can be either *deterministic* (same effect for the same combination of input parameters and the state of the invocation target in case of non-static methods) or *non-deterministic* (e.g. random number generation). If a method has non-deterministic effects, our approach simply has to record the effects of each method invocation to ensure that the invocation is not optimised away, and can use rare and selective logging of these values to prevent JIT from "optimising away" the invocations. But if the method has deterministic effects, the same input parameters cannot be used repeatedly, because the JVM detects the determinism and can replace *all* the method invocation(s) directly with a *single* execution (native) code sequence, e.g. using "constant folding". This forms an additional challenge to be solved in APIBENCHJ.

Thus, we would need to supply different *and* performance-equivalent parameters to methods with deterministic behaviour, and we have solved this challenge by using array elements as input parameters. By referencing the `i`th element of the arguments array `arg` in a special way (`arg[i%arg.length]`), we are able to "outwit" the JIT compiler, and also can use arrays that are significantly shorter than the number of measurements. Altogether, this prevents the JIT compiler from applying constant folding, identity optimisation and global value numbering optimisations where we do not want them to happen.

Other JVM optimisations such as Garbage Collection interfere with measurements and the resulting outliers are detected by our implementation in the context of statistical evaluation and execution control.

5.2 Generating Executable Microbenchmarks

Using the Java Reflection API, it is possible to design a common flexible microbenchmark for all methods of the benchmarked API, provided that the latter are invoked with the Reflection API method `method.invoke(instanceObj, params)`. However, invoking benchmarked API methods dynamically with the Reflection API is very costly [FF04] and will significantly bias the measured performance.

Source code generation is the straightforward way to construct reliable microbenchmarks.

Source code is generated based on models that represent the code to render. In case of benchmarking, each microbenchmark is specific to a single method of the Java API. Hence, for each method to benchmark, a model has to be manually prepared. However, the manual generation of the models and code templates for each API method would be extremely work-intensive and would contradict the goal of the presented work which is to automate of the Java benchmarking. In addition, if the API changes, the generation models must be manually adapted. Consequently, the scope of the benchmark would be limited to specific Java implementations. Using dynamic bytecode instrumentation (DBI) however, the insertion of instrumentation bytecodes is done at runtime. There is no need to restart an application each time a modification is done because of compilation issues.

APIBENCHJ directly creates the 'skeleton' bytecode for a microbenchmark, using the Javassist bytecode instrumentation API [Chi]. This 'skeleton' contains timer method invocations (e.g. calls to `nanoTime()`) for measuring the execution durations. The 'skeleton' also contains control flow for a warmup phase which is need to induce the JIT compilation (cf. Sec. 5.1). Thus, two benchmarking phases are performed: one for the 'cold' method (before JIT), and one for the hot (after JIT).

For each benchmarking scenario with appropriate preconditions, APIBENCHJ creates a dedicated microbenchmark that starts as a bytecode copy of the 'skeleton'. Then, the actual method invocations and preconditions are added to the 'skeleton' using Javassist instrumentation. Finally, APIBENCHJ renames the completed microbenchmark instance, so that each microbenchmark has a globally unique class name/class type, and all microbenchmarks can be loaded independently at runtime. An infrastructure to execute the microbenchmarks and to collect and their results is also part of APIBENCHJ.

6 Evaluation

We have identified the following three metrics for evaluating APIBENCHJ :

- **Precision**: compare APIBENCHJ results to "best-effort"manual benchmarking
- **Effective coverage** of packages/classes/methods
- **Effort** (time and space) of microbenchmark generation and execution

Once the APIBENCHJ implementation will be complemented by a component to detect parametric performance dependencies, a fourth metric (detectability of linear parametric dependencies) can be added. Of course, detecting parametric performance dependencies requires enough input data (different parameter, different invocation targets). This aspect will be addressed in future work. All following measurements were performed on a computer with Intel Pentium 4 2.4 GHz CPU, 1.25 GB of main memory and Windows Vista OS running Sun JRE 1.6.0_03, in `-server` JVM mode.

6.1 Precision of Automated Benchmarking

To evaluate the precision of automated benchmarking performed by APIBENCHJ, we had to compare its results to results of manual benchmarks (which were not readily available and had to be created for the evaluation). This comparison is needed to evaluate the

benchmark generation mechanism of APIBENCHJ; to enable a fair comparison, method parameters (and also method invocation targets) must be identical in both cases.

Hence, automated benchmarking was done first, and method preconditions during its execution were recorded and afterwards reused during manual benchmarking. This comparison is an indicator of whether the microbenchmark generation mechanism (cf. Sec. 4) generates microbenchmarks which will produce realistic results w.r.t JIT etc.

The method `java.lang.String.substring(int beginIndex, int endIndex)` was selected as a representative, because it is performance-intensive and because its declaring class is used very often. This method was benchmarked with an invocation target `String` of length 14, `beginIndex` 4 and `endIndex` 8.

The result of manual "best-effort" benchmarking performing by a MSc student with profound knowledge of the JVM was 9 ns. The benchmarking result of APIBENCHJ (after removing GC-caused outliers) had the following distribution: 7 ns for 19% of measurements, 8 ns: 40%, 9 ns: 22.5%, 10 ns: 9%, 11 ns: 4%, and 12 ns for 5.5% of measurements. Thus, the average results of APIBENCHJ are 8.555ns, which constitutes a deviation of 5% compared to manual benchmarking. Note that a distribution and not just a single value is observed in APIBENCHJ because the JVM execution on a single-core machine is interrupted by the OS scheduler to allow the OS other applications to use the CPU.

Clearly, this is a promising result, but it does not give any guarantees for other parameter values of `substring`, or for other API methods. At the same time, we think that it is a strong argument for the generation mechanism described in Sec. 5.

6.2 Effective Coverage of Packages/Classes/Methods

APIBENCHJ can benchmark *all* the methods for which correct (*appropriate*) and *sufficient* input parameters (and invocations targets for non-static methods) are given. By *sufficient*, we mean that the benchmarking method can be executed repeatedly with the input parameters. For example, the `java.util.Stack` class contains the method *pop()* which should be benchmarked, which means that the method must be called several hundred times to account for timer resolution. If the `Stack` does not contain enough elements to call `pop`, an `EmptyStackException` is thrown - thus, the invocation target (the used Stack instance) must be sufficiently pre-filled.

If parameter generation is automated, the coverage is less than 100% because not all parameters are generated successfully. In [KOR09], we have evaluated the coverage of heuristic parameter finding, which we summarise in this subsection since APIBENCHJ can successfully generate and run benchmarks for all parameters that were found in [KOR09].

For the `java.util` package of the Java platform API, APIBENCHJ can thus benchmark 668 out of 738 public non-abstract methods, which is a success rate of 90.51%. Similarly, for the `java.lang` package, APIBENCHJ can benchmark 790 out of 861 public non-abstract methods, which is a success rate of 91.75%, with an effort comparable to `java.util`.

6.3 Effort

The benchmarking for `java.util` took 101 min due to extensive warmup for inducing JIT optimisations; the heuristic parameter generation for it took additional 6 minutes. The serialised (persisted) input parameters (incl. parameters to create invocation targets) together with persisted benchmarking results occupy 1148 MB on hard disk for the `java.util` package, but only 75 MB for the `java.lang` package.

The generation of microbenchmarks using bytecode engineering is very fast. For the `String` method `contains(CharSequence s)`, the generation of the microbenchmark took less than 10 ms. The actual benchmarking took ca. 5000 ms: APIBENCHJ repeated the microbenchmark until a predefined confidence interval of 0.95 was reached (which required 348 repetitions). The number of repetitions depends on occurrence of outliers and on the stability of measurements in general.

A comprehensive *comparison* of the total effort for benchmarking using manually created microbenchmarks and of benchmarking using APIBENCHJ is desirable. However, to get a reliable comparison, a controlled experiment needs to be set up according to scientific standards and this would go beyond the focus and the scope of the presented paper.

7 Assumptions and Limitations

In this paper, we assume that appropriate and representative method parameters are provided by existing heuristics [KOR09], manual specification, or recorded traces. As described in Sec. 2, automated benchmarking should not be applied to API parts which can produce a security issue or data loss. The decision to exclude such API parts from benchmarking can only be produced by a human, not by APIBENCHJ itself.

As interface methods and abstract methods have no implementation, they cannot be benchmarked directly. At runtime, one or several non-abstract type(s) that implements the interface are candidates to provides implementation(s) for the interface methods. Performance prediction thus requires that these candidate type(s) are considered - then, the results of the corresponding microbenchmark(s) can be used. If such runtime information cannot be obtained in a reliable way, the performance of invoking an interface method cannot be specified for performance prediction, or *all* possible alternatives must be considered.

8 Conclusions

In this paper, we have presented APIBENCHJ, a novel modular approach for automated benchmarking of large APIs written in Java. It benefits researchers and practitioners who need to evaluate the performance of API methods used in their applications, but also for evaluating the performance of APIs that they create. The presented approach is suitable for the Java Platform API, but also for third-party APIs accessible from Java.

This paper is a starting point into research on API benchmarking, as there exists no comparable approach for generic, large high-level object-oriented (Java) APIs. It addresses such quantitatively important issues as Just-In-Time compilation (JIT) and the influence of method parameters, assuming that appropriate method parameters are provided by existing heuristics [KOR09], manual specification, or recorded traces. This paper provided a

first evaluation of APIBENCHJ on the basis of a subset of the Java Platform API including a comparison to the results of best-effort manual benchmarking.

Our next steps will include a large-scale study on the entire Java Platform API, and we also plan to automate the identification and quantification of parametric performance dependencies. In the future, APIBENCHJ can be extended by incorporating machine learning and other techniques of search-based software engineering for finding method parameters using APIBENCHJ 's plug-in mechanism, and for finding parametric dependencies [KKR09a]. It remains to be studied whether APIBENCHJ results can be used to *predict* the performance of a Java application in a real-life setting [KKR08], based on API usage by that application.

Acknowledgments: the authors thank Klaus Krogmann, Anne Martens and Jörg Henß and the entire SDQ research group for insightful discussions, suggestions and comments.

References

[Bel97] Doug Bell. Make Java fast: Optimize. *JavaWorld*, 2(4), 1997. http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html, last visit: October 9th, 2009.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[BSW+00] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[Chi] Shigeru Chiba. Javassist (Java Programming Assistant). http://www.csg.is.titech.ac.jp/projects/index.html, last visit: October 9th, 2009.

[Cor08] Standard Performance Evaluation Corp. SPECjvm2008 Benchmarks, 2008. URL: http://www.spec.org/jvm2008/, last visit: October 9th, 2009.

[CP95] Cliff Click and Michael Paleczny. A Simple Graph-Based Intermediate Representation. In *ACM SIGPLAN Workshop on Intermediate Representations*. ACM Press, 1995.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[GP] William Griswold and Paul Phillips. UCSD Benchmarks for Java. http://cseweb.ucsd.edu/users/wgg/JavaProf, last visited October 9th, 2009.

[Int09] Intel Corporation. Intel VTune Performance Analyzer, 2009. http://software.intel.com/en-us/articles/intel-vtune-performance-analyzer-for-windows-documentation/, last visit: October 9th, 2009.

[KKR08] Michael Kuperberg, Klaus Krogmann, and Ralf Reussner. Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models. In *Proceedings of CBSE 2008*, volume 5282 of *LNCS*, pages 48–63. Springer-Verlag, Berlin, Germany, October 2008.

[KKR09a] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. Using Genetic Search for Reverse Engineering of Parametric Behaviour Models for Performance Prediction. *IEEE Transactions on Software Engineering*, 2009. accepted for publication, to appear.

[KKR09b] Michael Kuperberg, Martin Krogmann, and Ralf Reussner. TimerMeter: Quantifying Accuracy of Software Times for System Analysis. In *Proceedings of the 6th International Conference on Quantitative Evaluation of SysTems (QEST) 2009*, 2009. to appear.

[KOR09] Michael Kuperberg, Fouad Omri, and Ralf Reussner. Using Heuristics to Automate Parameter Generation for Benchmarking of Java Methods. In *Proceedings of the 6th International Workshop on Formal Engineering approaches to Software Components and Architectures, York, UK, 28th March 2009 (ETAPS 2009, 12th European Joint Conferences on Theory and Practice of Software)*, 2009.

[lin07] Linpack Benchmark (Java Version), 2007. URL: http://www.netlib.org/benchmark/linpackjava/, last visit: October 9th, 2009.

[Omr07] Fouad Omri. Design and Implementation of a fine-grained Benchmark for the Java API. Study thesis at chair 'Software Design and Quality' Prof. Reussner, February 2007.

[Pie] Darryl L. Pierce. J2ME Benchmark and Test Suite. http://sourceforge.net/projects/j2metest/, last visit: October 9th, 2009.

[sci07] Java SciMark 2.0, 2007. URL: http://math.nist.gov/scimark2/, last visit: Oct. 9th, 2009.

[Sta08] Standard Performance Evaluation Corp. SPECjAppServer2004 Benchmarks, 2008. http://www.spec.org/jAppServer2004/, last visit: October 9th, 2009.

[ZS00] Xiaolan Zhang and Margo Seltzer. HBench:Java: an application-specific benchmarking framework for Java virtual machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 62–70, New York, NY, USA, 2000. ACM Press.