An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance

Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck
Johannes Kepler University Linz, Austria
{stadler, duboscq, moessenboeck}@ssw.jku.at

Thomas Würthinger, Doug Simon
Oracle Labs
{thomas.wuerthinger, doug.simon}@oracle.com

ABSTRACT

Java Virtual Machines are optimized for performing well on traditional Java benchmarks, which consist almost exclusively of code generated by the Java source compiler (javac). Code generated by compilers for other languages has not received nearly as much attention, which results in performance problems for those languages.

One important specimen of "another language" is Scala, whose syntax and features encourage a programming style that differs significantly from traditional Java code. It suffers from the same problem – its code patterns are not optimized as well as the ones originating from Java code. JVM developers need to be aware of the differences between Java and Scala code, so that both types of code can be executed with optimal performance.

This paper presents a detailed investigation of the performance impact of a large number of optimizations on the Scala DaCapo and the Java DaCapo benchmark suites. It describes the optimization techniques and analyzes the differences between traditional Java applications and Scala applications. The results help compiler engineers in understanding the characteristics of Scala.

We performed these experiments on the work-in-progress Graal compiler. Graal is a new dynamic compiler for the HotSpot VM which aims to work well for a diverse set of workloads, including languages other than Java.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

General Terms

Algorithms, Languages, Performance

Keywords

scala, optimization, dynamic compiler, virtual machine, compiler

Scala '13, Montpellier, France

1. INTRODUCTION

The Java Virtual Machine (JVM) was originally developed as the target machine for the Java language. However, it soon became apparent that the virtual machine is useful as a target for other languages as well. Current versions of the JVM specification [8] explicitly mention that Java is not the sole language to be compiled to Java bytecode. Scala [9] and Clojure [3] are examples of language implementations that run directly on top of the JVM without additional infrastructure, while language implementations like JRuby, Jython or Rhino rely on a significant amount of additional infrastructure.

Production-quality virtual machines are heavily optimized pieces of software. Their development and maintenance includes a tremendous amount of effort spent on performance analysis and tuning. This effort, however, focuses almost entirely on the performance of certain types of applications: small number-crunching applications (e.g., SciMark [11]), tools (e.g., DaCapo [1]), enterprise-level server applications (e.g., SPECjbb [15]), or combinations thereof (e.g., SPECjvm2008 [16]). All of these benchmarks almost exclusively run code written in Java.

While users of JVMs want to know the ideal JVM configuration for their application, a compiler engineer needs to understand which parts of the compiler, i. e. which optimizations, contribute most to the overall performance of an application. While there are attempts to analyze the influence of optimizations on Java code, there are no detailed analyses about the performance impact of optimizations on non-Java application code. Our work addresses this deficiency by providing a thorough analysis of the performance impact of a selected suite of optimizations. This analysis was performed on both the Java DaCapo [1] (version 9.12 "Bach") and the Scala DaCapo [12] (version 0.1.0 2012-02-16) benchmark suites, in order to compare the characteristics of Java- and Non-Java code. The fact that both benchmark suites use the same underlying infrastructure helps minimize outside effects on the results.

This paper provides detailed accounts of the results of the analysis, along with a description of how to replicate them. It discusses the results for the Scala DaCapo benchmarks in detail. It is also intended to solicit input from the Scala community on the types of Scala workloads the Graal compiler should be optimized for.

2. SYSTEM OVERVIEW

We performed our experiments on top of the Graal VM,

which is developed as part of the Graal OpenJDK project [2, 10, 14]. The Graal VM is a modification of the HotSpot VM that replaces the dynamic compilers (the HotSpot client and server compilers) and the compilation queue with implementations written in Java, while the underlying virtual machine infrastructure remains mostly unchanged. The meta-circular nature of Graal increases its approachability and malleability, so that it can be adapted to new technologies quicker and more safely. While most parts of Graal are still under heavy development, it already shows promising results, especially in terms of peak performance, when compared to the HotSpot client and server compilers.

The Graal VM uses HotSpot's profiling interpreter to collect information about the application code before compiling methods. This information includes branch probabilities, type profiles, exception probabilities and method invocation counters. The compiler can use these measurements to optimize the code it produces towards a specific application's behavior.

Graal does not, at the moment, employ a multi-tiered optimization scheme like the one used by the HotSpot server compiler. This means that there is only one optimization level, and thus that the first compilation of a method will already perform all optimizations.

When an assumption about the behavior of an application fails, and therefore the machine code is invalidated, the system continues execution in the interpreter. This transition from optimized code to the interpreter is referred to as *deoptimization* [4], and is a very important part of Graal's design. The interpreter updates the profile, so that the next time the code is compiled the new behavior is incorporated in the generated machine code.

Since the Graal compiler itself is written in Java, it will at the beginning only execute in the interpreter. In its current version, the Graal VM performs a bootstrapping phase during VM startup, in which the compiler compiles its own methods. At the beginning of the bootstrapping phase a few methods are manually added to the compilation queue. While compiling these methods other methods of the compiler will reach their compilation threshold, and will therefore be added to the compilation queue. This is repeated until the system stabilizes, which means that the compilation queue empties and no more methods are added to it. Work is currently in progress to use ahead-of-time compilation to make the bootstrap phase unnecessary.

3. EXPERIMENTAL SETUP

For our experiments, we executed the Scala DaCapo[1] and Java DaCapo[2] benchmarks with the HotSpot server and client compilers, with the Graal compiler in the default configuration, and with the Graal compiler in 16 special configurations. Each of the special configurations disables specific optimizations (see Section 3.2), represented by one or more options passed to Graal. We did not include a run with all optimizations disabled because some benchmarks take an excessive amount of time to finish in that case.

Each benchmark within the two benchmark suites (26 in total) was executed 10 times. Each execution consisted of a number of benchmark iterations that allows the compiler to warm up the application. The number of iterations needed to stabilize the results depends on the size of the benchmarks and their inputs. Table 1 lists the number of iterations used in our experiment, which allow the benchmarks to stabilize for all configurations we used. The result for one benchmark execution is the best time achieved by any of the iterations. In this way we measured the peak performance of the compiler.

Benchmark	Iterations
actors	10
apparat	10
factorie	10
kiama	25
scalac	20
scaladoc	25
scalap	40
scalariform	40
scalatest	12
scalaxb	40
specs	20
tmt	15

(a) Scala DaCapo benchmarks

Benchmark	Iterations
avrora	13
batik	20
eclipse	16
fop	30
h2	8
jython	13
luindex	10
lusearch	8
pmd	13
sunflow	15
tomcat	15
tradebeans	13
tradesoap	15
xalan	18

(b) Java DaCapo benchmarks

Table 1: Number of iterations per benchmark run.

This whole procedure was performed for each of the 19 different configurations, resulting in $19 \times 26 \times 10 = 4940$ benchmark runs with 90,060 benchmark iterations. The result for one benchmark and configuration is calculated as the arithmetic mean of all 10 executions. The standard deviation of the results is very small ($< 2\%$).

All runs were executed on an Intel Core i7-3770 quad core CPU running at 3.4 GHz, with 16 GB of RAM. The version of Graal used is available from the Graal OpenJDK repository, revision `221ef4b022c5`. The measurements can be replicated using the "mx" script included in the repository:

- Build Graal accorting to the instructions available in the OpenJDK wiki page[3].
- `mx scaladacapo <number of iterations> <benchmark> <options>` to run one of the Scala DaCapo benchmarks
- `mx dacapo <number of iterations> <benchmark> <options>` to run one of the Java DaCapo benchmarks
- Options can be specified using `-G:+OptionName` and `-G:-OptionName` to enable and disable them.

Table 2 lists the optimizations and the associated command line options.

3.1 Scala DaCapo Benchmarks

The Scala DaCapo benchmarks are described and caracterized in details by Sewe et al. [12]. A short summary is given in Table 3. It is worth noting that scalatest spends more than half of its time in Java code from the Java runtime. The kiama, scaladoc and specs benchmarks also spend a siginificant amout of time (about one third) in Java code.

[1]Version 0.1.0 (2012-02-16), which is based on Scala 2.8
[2]Version 9.12-bach
[3]`http://wiki.openjdk.java.net/display/Graal/Instructions`

Optimization	Options
Profiling Information	`UseProfilingInformation`
Assumptions	`OptAssumptions`
Type-Checked Inlining	`InlineMonomorphicCalls`, `InlinePolymorphicCalls`
Polymorphic Inlining	`InlinePolymorphicCalls`
Intrinsify ArrayCopy	`IntrinsifyArrayCopy`
Intrinsify System	`IntrinsifySystemMethods`, `IntrinsifyArrayCopy`
Intrinsify ObjectClass	`IntrinsifyObjectMethods`, `IntrinsifyClassMethods`
Intrinsify MathThread	`IntrinsifyMathMethods`, `IntrinsifyThreadMethods`
Read Optimization	`OptReadElimination`, `OptEarlyReadElimination`, `OptFloatingReads`
Loop Refactoring	`LoopPeeling`, `LoopUnswitch`
Loop Unrolling	`FullUnroll`
Partial Escape Analysis	`PartialEscapeAnalysis`
Tail Duplication	`OptTailDuplication`
Graph Caching	`CacheGraphs`
Use Exception Probability	`UseExceptionProbability`
Conditional Elimination	`ConditionalElimination`

Table 2: Graal options that correspond to one specific optimization.

Benchmark	Description
actors	Trading sample with Scala and Akka actors.
apparat	Framework to optimize ABC, SWC, and SWF files.
factorie	Toolkit for deployable probabilistic modeling.
kiama	Library for language processing.
scalac	Compiler for the Scala 2 language.
scaladoc	Scala documentation tool.
scalap	Scala classfile decoder.
scalariform	Code formatter for Scala.
scalatest	Testing toolkit for Scala and Java programmers.
scalaxb	XML data-binding tool.
specs	Behaviour-driven design framework.
tmt	Stanford Topic Modeling Toolbox.

Table 3: Description of the Scala DaCapo benchmarks.

3.2 Optimizations

In this paper we look at 16 different types of optimizations:

Profiling Information Profiling information is collected by the interpreter and can be used by Graal to create code that is tailored towards the most likely execution paths. For example, type checks that always encountered the same dynamic type at runtime can be replaced with a more efficient check and code that has never been executed in the interpreter can be omitted entirely.

Assumptions Due to dynamic class loading, the state of the class hierarchy can change at any time during the execution of an application. In practice, however, most applications' class hierarchy stabilizes very quickly. This can be exploited by the compiler, e.g., by assuming that the number of non-abstract implementation classes of an interface doesn't change. Such *static* assumptions need to be registered with the runtime system and may be invalidated in case of additional class loading.

Type-Checked Inlining In complex type hierarchies it is often impossible to determine the method that will be called by a call site statically. However, the profiling performed by the interpreter can provide the compiler with information about the types that were encountered at a call site during interpretation. The compiler can then insert a guard and inline the most likely methods. *Type-Checked Inlining* in general refers to inlining both monomorphic and polymorphic call sites.

Polymorphic Inlining Based on profiling information, a compiler might decide that inlining the code for multiple subclasses will increase performance. The implementations for the different subclasses will be guarded by a switch over the actual type, so that at runtime, the correct one is executed. The default policy will allow up to eight implementations to be inlined, but the actual decision of how many implementations to include depends on factors such as the size of the current method, the size of the inlined methods and the expected execution frequency of the call site.

Intrinsification Some methods that are frequently used in applications, like `System.arraycopy`, are implemented in the JDK as `native` methods. This means that calling them incurs a significant overhead, because these calls will perform JNI call marshalling and because the compiler cannot inline these calls.

For methods within the class library, the compiler can replace the calls with code that mimics the behavior of the original native implementation. Since this code is included in the compilation, it does not incur the call and marshalling overhead, and it can also be optimized further using information about the current compilation context (e.g., known types of parameters).

This process of replacing calls to known methods with an implementation is called *intrinsification*. It is an important optimization for certain groups of methods:

Intrinsify ArrayCopy The `System.arraycopy` method takes two `java.lang.Object` parameters for the source and the destination array. Apart form the native call overhead, this method suffers from having very generic parameter types. If the compiler encounters a call to System.arraycopy, then it can create an inlined version of the arraycopy logic that is tailored towards the type information the compiler has about the parameters.

Intrinsify System Intrinsifying the methods in the `System` class (e.g., `arraycopy`, `currentTimeMillis` and `identityHashCode`) is important for some applications. Disabling this optimization also disables the aforementioned `arraycopy` optimization.

Intrinsify ObjectClass `Object` and `Class` contain several methods that are frequently used. Examples are `Object.getClass` and `Class.isInstance`.

Intrinsify MathThread Both the `Math` and `Thread` classes also contain several methods that are used frequently in some applications. Examples include `Math.sqrt` and `Thread.isInterrupted`.

Read Optimization The Java Memory Model has a precise definition that allows the compiler to perform extensive reordering and restructuring of memory accesses. The compiler can use these opportunities to remove redundant reads and to move reads to more favorable locations. For example, some reads can be moved outside of loops, which will allow the compiler to move other computations out of loops.

Loop Refactoring Loop peeling and loop unswitching both restructure loops so that they can be executed or compiled more efficiently. Loop peeling pulls one loop iteration out in front of the loop. This allows loop invariant checks and calculations to be reused within the loop. Loop unswitching pulls "`if`" statements from within the loop to the outside, so that they only need to be evaluated once.

Loop Unrolling Full loop unrolling detects small loops with a small bounded trip count and replaces them with multiple copies of the loop body. This causes all calculations depending on the loop counter to fold away and removes most control flow.

Partial Escape Analysis Partial escape analysis allows the compiler to remove allocations or pull them into unlikely branches. In Graal partial escape analysis also performs extensive analysis that determines when boxing operations (e.g., `Integer.valueOf` and `Integer.intValue`) can be eliminated.

Tail Duplication Duplicating the tail after a merge in the control-flow graph means that a certain amount of code following the merge is copied into all branches preceding the merge. This is advantageous if the duplicated code contains references to values computed in those branches, which can then be replaced with the concrete value for each branch.

Graph Caching Graph caching during compilation reuses graphs of methods that are inlined multiple times. This happens quite frequently, with cache hit rates often above 90% [14]. This optimization is not expected to have a positive influence on the peak performance. It has been included in our experiment as a verification that it does not have a negative influence on performance.

Use Exception Probability The profiling information gathered by the interpreter tells the compiler at which call sites it encountered runtime exceptions. If the compiler doesn't use this information it has to create an order of magnitude more exception edges, which leads to much more generated code and an increase in compilation time.

Conditional Elimination The number of conditional expressions (e.g., switches, if-statements, guards or null checks) can be reduced by doing a data-flow analysis over the compiler graph that prunes conditionals that can be proven to be true or false. This, for example, removes or simplifies checkcasts within an if-statement that checks for the same type.

3.3 Optimization Dependencies

Compiler optimizations are not independent algorithms. Turning off one of them can adversely affect, or completely disable, other optimizations. Most optimizations have at least some effect on other optimizations. These are the most important dependencies:

- Without *Type-Checked Inlining* there is no *Polymorphic Inlining*.
- Disabling *Intrinsify System* also disables *Intrinsify ArrayCopy*.
- Turning off *Profiling Information* completely disables *Use Exception Probability* and makes *Type-Checked Inlining* and *Polymorphic Inlining* significantly less effective. *Loop Refactoring* will also make less informed decisions.
- Both *Type-Checked Inlining* and *Polymorphic Inlining* affect almost all other optimizations, because they enlarge the compilation scope. Without this larger compilation scope, optimizations like *Partial Escape Analysis* find less opportunities for optimization.

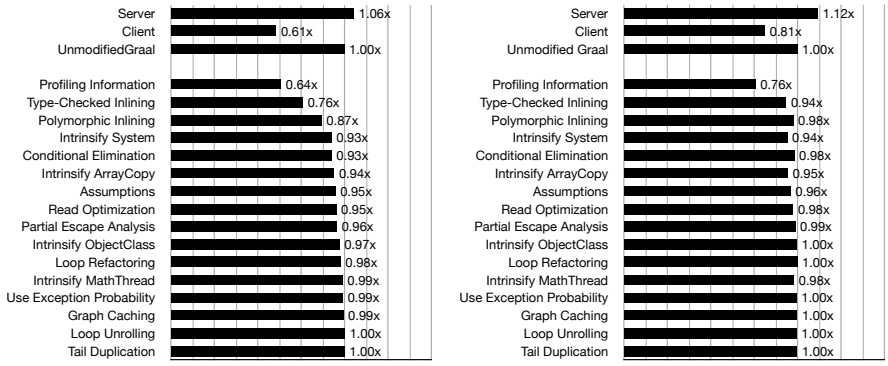
4. RESULTS

Figure 1 contains the detailed results for all Scala DaCapo benchmarks, and Figure 2 shows the average results for the Scala Dacapo and Java DaCapo benchmark suites. Each graph shows the performance of all 19 configurations in relation to the unmodified Graal configuration. All values are unbiased (starting at zero), larger-is-better performance numbers.

The server and client rows relate Graal's performance to the HotSpot server and client compilers. The other rows show the effects of disabling specific optimizations. They are ordered according to their average influence on the Scala DaCapo benchmarks, in order to list the most important ones first. A "Profiling Information" value of 0.57x, for example, means that disabling profiling information on this benchmark leads to a 43% decrease in performance.

Even though Graal is still work in progress, it shows good peak performance on most of the benchmarks, sometimes

Figure 1: Performance of the Scala DaCapo benchmarks for different configurations, on an unbiased, larger-is-better scale.



(a) ScalaDaCapo average (b) DaCapo average

Figure 2: Arithmetic mean (calculated from the performance numbers, not from the underlying benchmark run time) of all Scala DaCapo and Java DaCapo benchmarks. The values shown are performance relative to the unmodified Graal compiler, on an unbiased, larger-is-better scale.

even outperforming the server compiler. It is not, however, the goal of this paper to provide a statistically sound comparison of compiler performance. The server and client numbers have been included mainly to provide an estimate of the optimization potential that each benchmarks presents.

The following sections discuss the results of the Scala DaCapo benchmarks in detail.

4.1 General Observations

The *difference between client and server compiler* performance is a measure of the benchmarks "optimization potential". Benchmarks with a large difference show a bigger performance loss when optimizations are disabled.

The Scala DaCapo benchmarks are *not uniform*, some benchmarks like specs and scalatest show almost no optimization opportunities, while others like factorie and apparat have a huge optimization potential.

4.2 Influence of Optimizations

- *Profiling Information* is always the optimization with the biggest performance impact, because it is required for many of the other optimizations (Polymorphic Inlining, Type-Checked Inlining, Use Exception Probability, amongst others). Disabling it has a performance impact of up to 65% (factorie).

- *Type-Checked Inlining* is necessary for any compiler that wants to achieve good performance on Scala code. Disabling *Type-Checked Inlining* also disables *Polymorphic Inlining*, so the effect of speculatively inlining monomorphic call sites is visible as the difference between the Type-Checked Inlining and Polymorphic Inlining numbers. Polymorphic inlining, with an average impact of 13%, is even more important than monomorphic inlining at 11%.

  Speculative inlining of both monomorphic and polymorphic call sites is important for all benchmarks except for scalatest and specs. For certain benchmarks (factorie, tmt, apparat) the inlining of single-receiver calls is particularly important.

- Disabling *Intrinsify System* also disabled *Intrinsify ArrayCopy*, so it has, apart from measurement errors, at least the performance impact of arraycopy. The factorie results suggest that this benchmark uses many object identity operations, e.g., using `IdentityHashMap`.

- *Conditional Elimination* has a positive influence on many benchmarks, scalaxb and tmt in particular seem to have many redundant checks on conditions.

- *Intrinsify ArrayCopy* influences almost every benchmark. Arraycopy is used extensively in string operations, therefore benchmarks performing many string operations are affected most (kiama, scalaxb, scalariform, scaladoc).

- *Assumptions* have less influence on the overall performance than expected. This can be attributed to the fact that Scala uses interfaces extensively and that Scala applications are (statically) very polymorphic, i.e. the compiler less often has a chance to statically prove that a call site has only one possible receiver type.

- *Read Optimization* works well on benchmarks that use loops, because this optimization includes hoisting reads out of loops. This seems to be the case for example for the tmt benchmark which executes a rather small amount of numerical-oriented code and is also sensitive to loop optimizations. Otherwise, they have hardly any impact, because they are not sophisticated enough to provide large gains on modern CPUs.

- The impact of *Partial Escape Analysis* depends on the number of object allocations and the objects' lifetime. The factorie benchmark contains many allocations and is therefore hugely affected by our sophisticated partial escape analysis algorithm, which lets Graal outperform the server compiler on this benchmark. In their study, Sewe et al. [13] have indeed shown that factorie allocates at least two orders of magnitude more than most other benchmarks (from either Scala DaCapo or Java DaCapo). They also showed that most of the object allocated in this benchmark die relatively close to their creation point.

- *Intrinsify ObjectClass* is mostly relevant for factorie, scalariform and apparat. They use operations such as

`Object.getClass` and `Class.getComponentType` in hot parts of their code.

- *Loop Refactoring* is important for benchmarks with many small loops in the hot parts of their code. The impact of this optimization is a good measure for how important medium-to-small loops are in a benchmark.

- actors uses the methods of the `Thread` class frequently (e.g., `isInterrupted`), therefore *Intrinsify MathThread* is important for the performance of this benchmark. None of the other benchmarks is susceptible to this optimization.

- The additional code generated for exception handlers when *Use Exception Probability* is disabled is placed out-of-line, so it has little effect on normal execution. The apparat benchmark, however, is affected by this optimization, which means that the increase in compilation time and code size becomes a problem in this case.

- *Graph Caching* has a negligible influence on performance, which is to be expected since these measurements are peak-performance numbers. But the measurements also verify the assumption that the graph cache has no negative influence on peak performance.

- *Loop Unrolling* only works on small, fixed length loops. The Scala DaCapo benchmarks do not contain significant amounts of these loops.

- *Tail Duplication* does not have any influence on the Scala DaCapo benchmarks. This is most likely the result of the tail duplication not being "clever" enough, because the current implementation works only locally and does not have enough information to see when tail duplication is beneficial in a global scope.

4.3 Difference Scala DaCapo - Java DaCapo

Compared to other compilers, Graal is doing significantly better on Scala DaCapo than on Java DaCapo. This has two reasons: Scala code contains more opportunities for optimization, on which Graal excels, and the Java DaCapo benchmark suite has been the target of significant performance tuning on the client and server compilers.

In Scala DaCapo, type checked inlining is much more important than intrinsifying certain methods, while for Java DaCapo the intrinsics are more important. We plan to further investigate if Graal does intrinsify all the methods that are important for Scala, or if it misses important methods not present in the Java benchmarks.

In general, the Java DaCapo benchmarks show much less susceptibility to disabling optimizations. Many of the applications in this suite have been meticulously optimized by hand to run well on compilers that do only few optimizations.

One notable outlier in the Java DaCapo suite is the jython benchmark, which shows a behavior similar to the Scala DaCapo benchmarks. This is a hint that the results of this experiment can, to a certain degree, be generalized to other languages that compile to bytecode.

5. FUTURE WORK

We intend to perform the same studies on a larger set of benchmarks. This needs to include code that is compiled using more recent versions of Scala, and ideally results in a DaCapo-style benchmark suite representative of current Scala application code.

We also would like to perform a more thorough study that concentrates on a specific benchmark, to identify the specific challenges in terms of code patterns a compiler needs to recognize. In this context we should extend the configurations to incorporate different values for the heuristic settings (e.g., inlining limits or code size vs. speed) that the compiler takes.

6. RELATED WORK

The influence of different optimizations has been studied for static as well as for Just In Time (JIT) compilation. Hoste and Eeckhout [5] have researched Pareto-optimal sets of optimizations to satisfy multiple criteria such as compilation time and code quality or code size and code quality. Their work gives hints at the influence of different optimizations for statically compiled languages.

In the context of JIT compilation, the influence of optimizations has been studied to improve startup performance by selectively disabling optimizations in order to reduce compilation time. Jantz and Kulkarni [6] have extended this research to improve peak performance for Java. Their study was based on the server compiler and focuses on a small number of selected methods from SPECjvm98 and DaCapo 2006. Even if the set of optimizations that they selectively disable is different from ours, their results are similar for the ones that are equivalent.

A lot of work in the field of optimization selection uses machine learning methods, which can give results for the performance goals, but give little or no insight on why a particular optimization or a combination of optimizations performs well. However, as shown by Kulkarni et al. [7], the generation of decision trees using machine learning can lead to interesting insights into why and where optimizations are efficient or not.

The work of Sewe et al. [12] is useful to understand what the different Scala DaCapo benchmarks are doing and compares them to the Java DaCapo benchmarks. This can give an idea about what optimizations can be more important for Scala in comparison to Java. Sewe et al. [13] have also studied the different behavior of Scala and Java programs on the JVM. Their work focuses on memory management aspects of the JVM and also shows that programs from the Java DaCapo and Scala DaCapo benchmarks exhibit different characteristic behavior.

7. SUMMARY

Graal works well for Scala in terms of peak performance: it is only 6% below the server compiler on the Scala DaCapo benchmarks, while it is 12% below on the Java DaCapo benchmarks. Its performance advantage over the client compiler is also much larger for the Scala DaCapo benchmarks (39% vs. 19%).

Our observations have shown that Scala DaCapo presents the compiler with much more potential for optimization than Java DaCapo. Exploiting profiling information is particularly important, because Scala code contains more polymor-

phic call sites in hot code parts.

Some Scala DaCapo benchmarks show up to a 4x difference between the slowest and the fastest compiler. This is another indication that these benchmarks present many opportunities to an optimizing compiler.

We are happy to receive suggestions for additional Scala benchmarks from the Scala community. We can incorporate those benchmarks into our performance measurement tools and therefore ensure that we keep Scala in mind when adding further optimizations to Graal.

Acknowledgements

We would like to thank the VM research group at Oracle and everyone working on the Graal OpenJDK project for their contributions to the Graal effort, and the reviewers for their thorough and constructive reviews.

References

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190. ACM Press, Oct. 2006. doi: 10.1145/1167473.1167488.

[2] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, and C. Wimmer. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[3] R. Hickey. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*. ACM Press, 2008.

[4] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992. ISBN 0-89791-475-9. doi: 10.1145/143095.143114.

[5] K. Hoste and L. Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 165–174. ACM Press, 2008. doi: 10.1145/1356058.1356080.

[6] M. R. Jantz and P. A. Kulkarni. Performance potential of optimization phase selection during dynamic JIT compilation. In *Proceedings of the ACM/USENIX International Conference on Virtual Execution Environments*, pages 131–142. ACM Press, 2013. doi: 10.1145/2451512.2451539.

[7] S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2013. doi: 10.1109/CGO.2013.6495004.

[8] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java^{TM} Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley, 2013.

[9] M. Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[10] OpenJDK Project. Graal Project. URL `http://openjdk.java.net/projects/graal/`.

[11] R. Pozo and B. Miller. SciMark 2.0. URL `http://www.spec.org/jbb2005/`.

[12] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da Capo con Scala: design and analysis of a scala benchmark suite for the java virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 657–676. ACM Press, 2011.

[13] A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, and S. Z. Guyer. new Scala() instance of Java: a comparison of the memory behaviour of Java and Scala programs. In *Proceedings of the International Symposium on Memory Management*, pages 97–108. ACM Press, 2012. doi: 10.1145/2258996.2259010.

[14] L. Stadler, G. Duboscq, H. Mössenböck, and T. Würthinger. Compilation queuing and graph caching for dynamic compilers. In *Proceedings of the ACM workshop on Virtual Machines and Intermediate Languages*, pages 49–58. ACM Press, 2012. doi: 10.1145/2414740.2414750.

[15] Standard Performance Evaluation Corporation. SPECjbb2005, . URL `http://www.spec.org/jbb2005/`.

[16] Standard Performance Evaluation Corporation. SPECjvm2008, . URL `http://www.spec.org/jvm2008/`.